Exam – Solutions EDA040 Concurrent and Real-Time Programming

2010–12–18, 08.00–13.00

1. a) As explained on slide 3 of lecture 2, the save-switch-store that forms a context switch that takes place when the system changes running process/thread.In a typical preemptive kernel, switching from one thread to another then consists of (an answer simply lists most of these items):

   - Turn off interrupts.
   - Push PC, CPU registers (Ax, Dx, SR) on stack.
   - Save stack pointer in process record.
   - Get new process record and restore stack pointer from it.
   - Pop CPU registers (SR, Ax, Dx) from stack, and pop PC.
   - Turn on interrupts.

   Hence, each thread has its own stack, allocated at thread creation on the heap.

   b) Perhaps on the limit of the course content, but to promote desired understanding of how the computer/compilers work, and an answer could be based on either of the following two aspects:

   - without preemption the switch locations are reflected in the source/binary code as compiled, and then the context can be smaller since scratch registers do no need to be saved/restored.
   - some types modern CPUs with pipelining and/or cashing might be more efficient when not natively pre-empted.

   c) When a reschedule is desirable in a system without preemption.

   d) For reasons according to a and b, the floating-point registers are part of the context too. With hardware support the computations (using the floating-point registers) are on average faster, but with too frequent context switching can counteract and decrease performance.

2. a) If there is a clock interrupt (increasing the system time) after one sleep and before next is started (e.g. due to other threads using the CPU), there will be a drift that also accumulates. Additionally, sleep is specified to sleep *at least* as long as requested.

   b) Both extra delays due to the CPU-load, and scheduler of the OS, differ. Additionally, Roger might have had strict priorities, which for normal Windows threads are not the case.

   c) See bottom p34.

   d) No, there can still be a clock interrupt between determining the time to sleep and the start on the sleeping as part of the system call, and then the second with be one OS tick to long (typically 1ms, but the next second normally that much shorter). To avoid this, sleepUntil needs to be a system call that is properly integrated in the OS kernel.

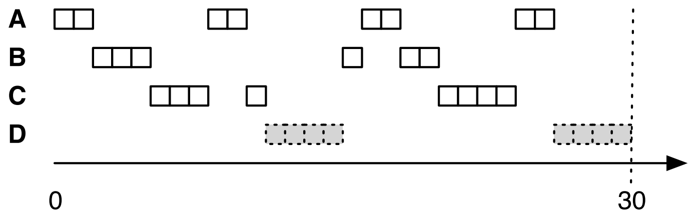
3. a) Dekker's algorithm

   b) Inefficiency, and an increased risk of starving other threads (e.g. if one threads dies and the other keeps looping too long without sleeping).

4. a) Thread B can experience push-through blocking. Assume C is executing and enters the monitor M. While it is executing inside the monitor, B starts executing, preempting C. While B is executing, A preempts B and tries to enter the monitor M. Since the monitor is blocked by C, C inherits the priority of A in effect blocking B. C can execute with elevated priority for at most 0.3 ms before exiting the monitor. Then A can finish executing and execution of B can continue.

   b) As stated above, C can block B for at most 0.3 ms.

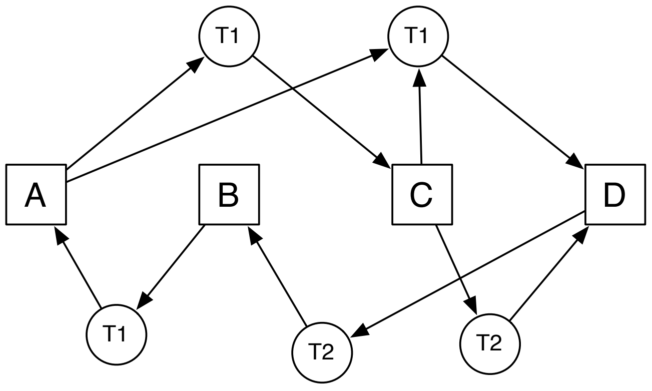
5. a) We draw a scheduling graph for the critical instant for the three given threads A, B, and C. The time not allocated to these three threads in the interval 0–30 is available for executing thread D. By counting the amount of available free time slots we get the maximum possible execution time for thread D:



   Answer: 8 ms

   b) Here, we can apply the scheduling test of Liu and Layland which state that the system can be guaranteed to be schedulable if the total CPU utilization is below $n(2^{1/n} - 1)$, where n is the number of threads (in this case 4).
   According to the given table, $4(4^{1/4} - 1) \approx 0.757$. The CPU utilization of thread D is $C_D/30$.
   We get:
   $0.25 + 0.20 + 0.20 + C_D/30 \leq 0.757$
   $C_D \leq 30 \cdot (0.757 - 025 - 0.2 - 0.2)$
   $C_D \leq 30 \cdot 0.107$
   $C_D \leq 3.21$
   By rounding downwards we get:
   Answer: 3 ms

6. a) We draw the resource allocation graph:



   b) No, deadlock can not occur since the graph shows that at least two instances of T1 is required for deadlock.

7. **Timed Event Distribution**

This problem checks the understanding of the central parts of the course, as also represented by the three lab assignments. Specifically, for respective lab:

1. Concurrency accomplished by the separation of (encapsulated) shared data and threads (that are as simple as possible) that operate on that data via well-defined (not too small) operations. This was part of the available design here, but understanding of threads in terms of their execution, objects, blocking and timing is needed.
2. Monitors with conditions in terms of wait-notify is the key object oriented concept for solving the central part of this problem, and the four monitor methods were given to clarify the problem.
3. Message-based communication forms the background and motivation for the need of the handling of timing, blocking and starvation. Understanding of that, the use of events not only for buffering, and the effects of full/blocking buffers was beneficial.

Thus, the solution here is simply to implement the missing four monitor methods, and to add the attributes that are needed. A complete solution with comments and exceptions handling is almost 100 lines as shown below. By covering the central parts of the code half of that (some 50 lines if the similarity between the add and remove methods was pointed out) grade 4 was possible. Grade 3 required some core function to be solved properly. Some comments based on the requirements to be fulfilled now follows as a further description of the solution (i.e., the code) below:

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.*
   See `addListener` below where `super.addListener`is called with the argument being `Thread.currentThread().getPriority()`, which also examplifies how we deal with threads object (with its `static` method `currentThread`) and the thread of execution (that `getPriority` returns the priority for).
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.*
   See first if statement in `putEvent` where a termination event is sent (in case running properly but simply accupied with other processing, and thereafter and possibly much later it will terminate upon processing that event), the thread is then interrupted (for termination if properly programmed), the priority is thereafter set to be the minimum (in case of long computations ongoing as threads are not to be killed), and the reference is set to null (for GC and for request creation of new thread).
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.*
   See first while loop in `putEvent`: The `event` reference (that is used to pass the event from producing threads calling `putEvent` to the distributing thread calling `next`) is checked for `null`, which means previous distribution was finished successfully.
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).*
   See first while loop in `putEvent`: The timeout check needs to be around the same wait as for next event (see previous item) since it cannot be known which one occures first, and time handling is implemented accordingly including the timeout of `wait` and `if` thereafter.
5. *New threads may only be created (and started) upon first initialization, and after a hang according to the previous item.*
   See how attribute `distributor` is used below in methods `putEvent` and `next`.
6. *If a `putEvent` consumer call returns but takes more than 2ms, that delaying listener should be automatically removed from the subscriber list.*
   Since the `Distributer` thread is the caller, is needs to check the time as shown in the available implementation, but it should not call a separate/new monitor method (for reasons like in lab 2 and 3, as here shown by the existence of the argument `tooSlow` to `next`).

7. *Any calls to `addListener` and `removeListener` should be blocked during event distribution (of one event).*
   This is the motivation for the attribute distributing.
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.*
   Feature of fulfilling the other requirements, specifically the first one.
9. *Before starting to distribute another event, all pending calls to `addListener` and `removeListener` should be permitted to complete.*
   This is why there needs to be a counter (a boolean whould only work for one call at a time) that each call of add or remove listener can mark their interest for change by incremetning it, followed by the decrement when done. This is like a counting semaphore but part of the monitor functionality. See attribute `reorganize` and its usage in the `addListener`, `removeListener` and `next` methods.

Taking the above requirements into account, the simplification mentioned in the problem description, and starting out from the solution outline, the following code is one complete solution (also ignoring the fact that `java.util.EventObject` does not implement `Clonable` and lack of support for explicit change of ownership in `se.lth.cs.realtime.event`, which will be fixed in the next version of the class libraries): *(13p)*

```
public class DeMux extends RTEventDispatcher {

  // Attributes as needed...
  int index = -1;        // indexing the listeners, -1 for inactive
  long since;            // for storing arrival time of producer
  JThread distributor;   // the thread that might need replacement
  boolean distributing;  // flag for ongoing distribution
  int reorganize;        // number of listener-change calls pending
  long tmax = 8;         // the deadline for complete distribution
  RTEvent event;         // the next event to be distributed
  RTEvent harakiri;      // for telling a hanged distributor to give up

  /** For one more way of telling listener threads to give up */
  static class HarakiriEvent extends RTEvent {
    public HarakiriEvent() {owner = null;}
  }

  /** return type for method next below */
  static class OnePutEvent {
    RTEvent message;
    RTEventListener destination;
    long before;
  }

  public DeMux() {
    harakiri = new HarakiriEvent();
  }

  /**
   * 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) {
    if (!distributing || reorganize==0) notifyAll();
    reorganize++;
    while (distributing) {
      try {
        wait();
      } catch (InterruptedException exc) {
```

```
      throw new RTInterrupted();
    }
  }
  super.addListener(dest, Thread.currentThread().getPriority());
  reorganize--;
  notifyAll();
}

/**
 * Remove the listener.
 * @param dest the listener no longer listening
 */
public synchronized void removeListener(RTEventListener dest) {
  if (!distributing || reorganize==0) notifyAll();
  reorganize++;
  while (distributing) {
    try {
      wait();
    } catch (InterruptedException exc) {
      throw new RTInterrupted();
    }
    super.removeListener(dest);
  }
  reorganize--;
  notifyAll();
}

/**
 * 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) {
  long arrival = System.currentTimeMillis();
  long t = arrival;
  // Wait for previous calls to complete...
  while (event!=null && t-arrival<=tmax) {
    try {
      wait(arrival+tmax-t);
      t = System.currentTimeMillis();
    } catch (InterruptedException exc) {
      throw new RTInterrupted();
    }
  }
  // Take care of uncompleted distribution
  if (event!=null || t-arrival>tmax) {
    distributor.putEvent(harakiri);
    distributor.interrupt();
    distributor.setPriority(Thread.MIN_PRIORITY);
    distributor = null;
  }
  // provide next event to be distributed
  if (event==null) {
    event = evt;
    since = arrival;
    index = 0;
  };
  // if first time or hanged distributor thread, create new
  if (distributor == null) {
    distributor = new Distributer(this);
    distributing = false; // to be changed by distributor
    distributor.start();
    };
    notifyAll(); // for earlier running distributor or for reorganization
    if (t>arrival+tmax) return evt; // returning ref to late delivery
    return null;
  }

  /**
   * 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) {
    OnePutEvent ans = new OnePutEvent();
    // acknowledge completion of previous round and signal ready for new
    if (index<0) {
      event = null;
      notifyAll();
    }
    // wait for any work to do
    while (event==null) {
      try {
        wait();
      } catch (InterruptedException exc) {
        throw new RTInterrupted();
      }
    }
    // still this thread working or should I die
    if (distributor != Thread.currentThread()) {
      throw new RTError("harakiri");
    }
    // take care of pending subscription changes
    while (reorganize > 0) {
      try {
        wait();
      } catch (InterruptedException exc) {
        throw new RTInterrupted();
      }
    }
    // remove delaying previous listener
    if (tooSlow != null) {
      removeListener(tooSlow);
      index--; // Compensate for removed listener
    }
    // ok, let's distribute
    if (index < 0) throw new RTError("bad mon impl 1");
    if (index==0) distributing = true;
    if (index < listenerList.length) {
      ans.destination = listenerList[index++].listener;
      ans.message = event.clone();
      ans.before = since + tmax;
    };
    if (index > listenerList.length) throw new RTError("bad mon impl 2");
    if (index == listenerList.length) {
      index = -1; // Got last listener
      event = null;
      notifyAll();
    }
    return ans;
  }
}
```

8. **Networked Ad-hoc Event Distribution**
The problem was to design a networked system similar to the one of the previous problem, but with some difference in distribution (threads are running on different machines) and timing (the timeouts relaxed) and instead it is about periodically probing the subscriberes to detect is the connection if lost due the (as stated) unrelaible network. The following is not a solution, but for each requirement (quoted in italics) it is commented what a solution should show.

   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.*
   Either the thread receiving the producer events, or the calling thread of the producer in case that is on the same machine (the important aspect of threads being distributed is that the consumers should be distributed, whereas it was considered to be acceptable to assume that the producer thread runs in the same program at the dispatching monitor exists in) can then be permitted to call the dispatching, but the `Distributer` thread can also be kept for additional reliability as in the previous problem. Thus, this requirement is more of a simplification, but should be understood.
   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.*
   Since communication is also blocking, it should be shown what additional threads that are needed for the communication, and that is similar to what was needed for camera communication within the course project.
   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.*
   Hence, if the producer thread is in the same program, the monitor functionality is better kept, and there needs to be threads receiving those subscription events (and blocking on the listener calls of the monitor). In case of the monitor and the producer being on different machines (or in different programs), a monitor-serving thread serializes the calls (serving one event at a time), but to serve all pending listener-change calls before next dispatching requires additional functionality in the event-receiving thread.
   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.*
   That is natural to do when the subscription is made, as an attribute in that type of event, and the monitor extended with that information.
   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.*
   It should be detected/managed in the monitor which is the fastest subscriber, and that thread is then requested (by an extra event) to emit ping events (by itself without waiting for ping event from the Distributer). When those events arrive to the dispatching program, reactively by serving those network events without having an extra thread, it is detected as part of the logic in the monitor when it is time to ping each of the other (slower) subscribers.
   6. *Consumers that are not responding properly to the ping messages should be removed.*
   Preferably handled by the thread sending the ping events, but somewhat depending on how the rest of the solution is constructed.

It was also pointed out *"Special emphasis should be put on explaining how lost connections are detected and managed"*. This is related to the second last item, and a description or figure explaining how that is accomplished without an extra thread gives one point extra. A clear and understandable figure (or clear explainations) also gives one point extra. Otherwise, it is half a point per requirement, but some balancing in case it is not clear how the combinations of them are fulfilled. *(5p)*