Exam
Concurrent and Real-Time Programming

2012–01–09, 08.00–13.00

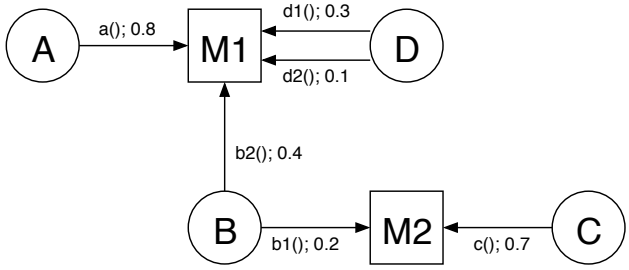
You are allowed to use the Java quick reference and a calculator.
Also dictionaries for English (and the native language for each student) is allowed.

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 (or excellent on the other problems) is required for highest grade (5).

---

1. A core component of multi-threaded systems is the scheduler.
   a) Explain the job of a scheduler. *(1p)*
   b) Explain the following scheduler states; running, ready, blocked. When do state transitions occur? *(1p)*

2. A real-time system consists of four threads named A, B, C and D that communicate through two monitors M1 and M2. The graph below shows the monitor methods called from each thread, annotated with the worst case execution time per method. The table shows the period (T) and worst case execution time (C) for each thread. The system is scheduled using RMS and utilizes the basic inheritance protocol. All times are given in milliseconds.



	T	C
A	2	1
B	5	2
C	17	1
D	25	1

   a) Calculate blocking delays. *(2p)*
   b) Determine if the system is schedulable. Your answer must be motivated. *(2p)*

   Recall:

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

3. In real-time systems, the semaphore is often used in two synchronization use cases, called signalling and mutual exclusion.

   a) Explain each case. *(1p)*

   b) Sometimes it is useful to provide a special mutual exclusion semaphore. Motivate why this could be the case. *(1p)*

4. The code below is a badly written solution to the Dining Philosophers problem. Only the run method is erroneous.

   a) What is the problem with the code? Draw a resource allocation graph illustrating the problem. *(1p)*

   b) Correct the code. *(2p)*

```
import ...

class Philosopher extends Thread {
  static Semaphore[] fork;
  int i;

  Philosopher (int i) {
    this.i=i;
  }

  public void run () {
    while ( true ) {
      think();
      fork[i].take();
      fork[(i+1)%4].take();
      eat();
      fork[i].give();
      fork[(i+1)%4].give();
    }
  }

  private void think() { ... }

  private void eat() { ... }

  public static void main(String[] args) {
    fork = new MutexSem[4];
    for (int i=0; i<4; i++) fork[i] = new MutexSem();
    for (int i=0; i<4; i++) (new Philosopher(i)).start();
  }
}
```

5. A real-time system with strict priorities is to be scheduled with fixed-priority scheduling, with priorities assigned according to the rate-monotonic principle. A previous pessimistic analysis has arrived at blocking delays for the threads. The table below gives partial information about the system:

Thread	C (ms)	T (ms)	B (ms)
A	2	7	4
B	2		4
C	2		4
D			0

Thread D represents a sporadic server to handle external events (running at lowest priority). To respond well there should be 40% free cpu utilization bandwidth (to be used by thread D). Design a schedulable system so that T_B is between 8 and 10 ms and T_C is between 8 and 30 ms, preferably as low as possible. The threads should be prioritized from A (highest) to D (lowest). Select T_D as low as possible and determine the maximum available C_D that still makes the system schedulable. You need to show that your system is schedulable. *(4p)*

Recall:

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

$\frac{1}{7} \leq 0.15$ $\frac{1}{8} \leq 0.13$ $\frac{1}{9} \leq 0.12$ $\frac{1}{10} \leq 0.10$ $\frac{1}{11} \leq 0.091$ $\frac{1}{12} \leq 0.084$ $\frac{1}{13} \leq 0.077$
$\frac{1}{14} \leq 0.072$ $\frac{1}{15} \leq 0.067$ $\frac{1}{16} \leq 0.063$ $\frac{1}{17} \leq 0.059$ $\frac{1}{18} \leq 0.056$ $\frac{1}{19} \leq 0.053$ $\frac{1}{20} \leq 0.050$

6. The following is about using knowledge from the course for understanding some typical related information as obtained from the web.

Background extracted from Internet: A *quantum* is a term used in many real-time systems. It represents a standard amount of cpu time (measured in number of cpu *time slices*) a thread is allotted before it is forced to yield the cpu to another thread. The default quantum may depend on thread type and/or system configuration. During execution the system typically keeps track of the remaining quantum at any given time.

In Java there is a method called *yield()* used to manually give up execution to another thread. It is rarely used because the semantics varies between systems and typical yield situations can be performed by other means. For instance (from `http://www.javamex.com/tutorials/threads/yield.shtml`):

> **When to use yield()?**
> I would say practically never. Its behaviour isn't standardly defined and there are generally better ways to perform the tasks that you might want to perform with yield():
>
> - if you're trying to use only a portion of the CPU, you can do this in a more controllable way by estimating how much CPU the thread has used in its last chunk of processing, then sleeping for some amount of time to compensate: see the sleep() method;
> - if you're waiting for a process or resource to complete or become available, there are more efficient ways to accomplish this, such as by using join() to wait for another thread to complete, using the wait/notify mechanism to allow one thread to signal to another that a task is complete, or ideally by using one of the Java concurrency constructs such as a Semaphore or blocking queue.

Problems: Comment on the following by answering the questions in the final paragraph of subproblem **a** to **c**:

a) Yield behavior on windows (also from the same url as above):

> **Windows**
> In the Hotspot implementation, the way that Thread.yield() works has changed between Java 5 and Java 6.
>
> In Java 5, `Thread.yield()` calls the Windows API call `Sleep(0)`. This has the special effect of clearing the current thread's quantum and putting it to the end of the queue for its priority level. In other words, all runnable threads of the same priority (and those of greater priority) will get a chance to run before the yielded thread is next given CPU time. When it is eventually re-scheduled, it will come back with a full quantum, but doesn't "carry over" any of the remaining quantum from the time of yielding. This behaviour is a little different from a non-zero sleep where the sleeping thread generally loses 1 quantum value (in effect, 1/3 of a 10 or 15ms tick).
>
> In Java 6, this behaviour was changed. The Hotspot VM implements `Thread.yield()` using the Windows `SwitchToThread()` API call. This call makes the current thread give up its current timeslice, but not its entire quantum. This means that depending on the priorities of other threads, the yielding thread can be scheduled back in one interrupt period later. (See the section on thread scheduling for more information on timeslices.)

Assume there are some high-priority threads executing in a high-priority process. Also assume that according to scheduling analysis each thread use only a small part of the standard quantum. Motivate why the Java6 behavior is preferable compared to the Java5 one. Are there situations when the Java5 implementation is better? *(1p)*

b) Yield behavior on Linux (from the same url):

> **Linux**
> Under Linux, Hotspot simply calls `sched_yield()`. The consequences of this call are a little different, and possibly more severe than under Windows:
>
> - a yielded thread will not get another slice of CPU until all other threads have had a slice of CPU;
> - (at least in kernel 2.6.8 onwards), the fact that the thread has yielded is implicitly taken into account by the scheduler's heuristics on its recent CPU allocation. Thus, implicitly, a thread that has yielded could be given more CPU when scheduled in the future.

There are other version of the Linux kernel with different types of scheduling, but according to this description even if you assume strict priorities, is the Linux kernel (default scheduler) suitable for real-time? Motivate your answer. *(1p)*

c) In case of non-preemptive scheduling, what would you use the yield method for? *(1p)*

7. Monitor Timing

Background

Typically threads interact via shared resources that often are in the form of monitors, and the maximum blocking times will depend on the execution times for the monitor methods. However, after estimating those WCETs, how do we (when running the system) check that we never encounter anything worse than the (assumed) worst case? For each deadline we could check that the deadline is not passed. We would then have both the thread itself doing the test, and also a supervising thread with highest priority since the thread missing its deadline could be starving. This problem is about managing execution times for monitor methods, and taking actions when deadlines are not met.

Introduction

As indicated in problem 6, some scheduling techniques require the run-time system to keep track of the execution time for each thread. We would then need be able to obtain the execution time for the current thread (e.g.,starting at zero when the thread is started, and time not progressing when the thread is blocked), but that would require the OS kernel and scheduler to be instrumented accordingly. Now, instead, obtaining the execution time is simply a native method reading the CPU clock (i.e., execution is from the hardware point of view), and for any usage we have to be aware about preemption by other threads.

Obtaining a high-resolution clock is not available for some operating systems so for portability it is not part of the standard Java classes. Using the keyword native, however, the implemented method is

```
public native long executionTimeMicros()
```

which is part of class `se.lth.cs.realtime.RTSystem`. In this problem you can assume the time never flips over. As mentioned, this execution time is based on the built-in CPU high-resolution clock that is available for all modern CPUs, and the time is scaled to the returned number in microseconds.

The assumed WCET could be given statically by annotations or dynamically by properties assigned by the class-loader, but here we simply have them given in the code of each method. To explore the principles, we here simply consider one specific monitor.

For reading the system time the standard `System.currentTimeMillis()` method call is available. This system clock time is given in milliseconds. Time resolution is here assumed to be 1 ms.

Thread priorities can be read by the `int Thread.getPriority()` method call. Furthermore, the `static Thread Thread.currentThread()` method returns the currently executing thread as a Thread.

Problem

Ensuring that deadlines are met within real-time software is preferably done by scheduling analysis at engineering time, such that the real-time scheduling (e.g., priority-based by strict priorities) at run-time results in the timing requirements being fulfilled. In case of uncertainties (such as uncertain convergence of iterative computing or uncertain WCET for certain functions), however, additional run-time checks might be needed, to detect or handle error situations.

Consider a system using a single monitor `M` with three methods `x`, `y` and `z`, all presumably without arguments and return values (since that would not have any influence on our topic). For simplicity we assume all threads are of type `Thread` and scheduled by a priority-based real-time scheduler (strict priorities, basic priority inheritance, etc.). The estimated WCETs for methods `x`, `y` and `z` are given by WCET-constants in the monitor code below.

There is also a method `s` that is not part of the application but added for supervising the timing and for reporting errors. There is a supervisory thread calling method `s`. You are not supposed to write any of the mentioned threads.

In the following the monitor is to be implemented such that the methods fulfill certain requirements on timing checks or error handling. Specifically three methods `x`, `y`, and `z` should have certain properties for the application to work, and an additional method `s` is called by a supervisory thread that should report any timing error. There is also a worked thread calling a method `w`, as explained

below for method y. You do not have to write in of those threads, but the monitor should be developed accordingly. The purpose of the monitor is to prototype different techniques to supervise timing for three types of methods that have the following characteristics:

M.x(): Method x should fulfill strict timing requirements when called by high-priority threads, and inside the monitor is has to wait for external conditions. The x method uses `wait(timeout)`, and does so twice per call to wait for network packets to arrive, but when one thread has entered (and is waiting on wait) other callers may not pass even if such a second calling thread has higher priority (than the one already waiting).

Only when the highest priority thread (except for the supervising checker thread) calls method x we care about the execution time of the method. That is, only when the caller has priority `MAX_PRIORITY-1` the execution time (including blocking times) should be considered (and reported if longer than the WCET).

M.y(): Method y contains an algorithm that should fulfill its WCET but the convergence of the iterations therein is not fully tested, and a reporting a missed deadline should therefore not wait until the call completes. Therefore, the actual call of the algorithmic code needs to be carried out by another thread, which is the one (provided for you) calling method w. Failure to return before the stated WCET should be reported via method s. If there is no worker thread (no caller blocked within method w), the call return after an error being reported via s.

M.z(): Method z is to be called periodically for purposes of feedback control that is critical for keeping the controlled system stable. While the calling thread can check if the response time is acceptable, it should be checked within the monitor that the WCET is not exceeded. If the periodic calls suddenly stops, that should also be detected and reported (via method s) within approximately one second.

The above items list what key properties of each method that will be evaluated. Your task is to develop the monitor and answer questions c-d. That is:

a) Develop the monitor M based on the provided code template below, such that the specifications contained in the comments are met. *(10p)*

b) Why, despite not waiting for any notify, do you need to use wait(timeout) in method x, instead of just calling `Thread.sleep(timeout)`? *(1p)*

c) Why is it necessary to have a worker thread (in some form, here obtained via a method w call) calling the unpredictable algorithms in method y? *(1p)*

d) The supervisory thread (calling s) was needed also for reporting stopped calls of method z, but what is the run-time cost of that? Consider if the period would be 10ms and missing calls should be reported within 20ms. Any ideas how that cost could be reduced (by help from the OS or hardware)? *(1p)*

The code template for the monitor:

```
public class M {
  // Some example attributes, to be used or replaced by your own:
  Object           xLock;
  int              xLate, yLate, yLateOld, zLate, zLateOld;
  long             zTime, sTime;
  Thread           worker;
  boolean          todo;

  // Worst Case Execution Time per method in microseconds:
  static final int WCET_X = 3000;
  static final int WCET_Y = 5000;
  static final int WCET_Z = 4000;

  public M() {
    xLock = new Object(); // In case you want to do things this way.
    // Whatever more that is needed...
```

```
}

/**
 * Helper method to be called from within synchronized methods below,
 * containing the try catch around the wait.
 */
void await() {
  try {
    wait();
  } catch (InterruptedException exc) {
    throw new RTError("interrupted wait()");
  }
}

/**
 * Helper method to be called from within synchronized methods below,
 * containing the try catch around the wait with a timeout argument.
 */
void await(long timeout) {
  try {
    wait(timeout);
  } catch (InterruptedException exc) {
    throw new RTError("interrupted wait(timeout)");
  }
}

// ---------------------------------------------------------------- Method x:

/**
 * Request and obtain one network message, by using the private methods x1
 * and x2 in the following sequence:
 *
 *            x0(); wait(dt1); x1(); wait(dt2); x2();
 *
 * where the dt1 and dt2 times, in microseconds, are dt1=800 and dt2=600.
 * Delays of twice those numbers are fine, but not more. It can be assumed
 * that higher priority threads and the runtime system never takes more
 * than 100 microseconds. The time granularity of the system is 1ms.
 *
 * Thus, since the waiting is specified in microseconds, and there is no
 * wait(millis,nanos) available, the RTSystem.executionTimeMicros has
 * to be used to decide if the next millisecond should be waited for.
 *
 * In total the return of x2 should be within WCET_X microseconds.
 * However, only when the caller has MAX_PRIORITY-1 violation of those
 * timing constraints should be reported (by method s). In that case,
 * this method returns after method s has notified that the error has
 * been taken care of.
 *
 * When one caller has started running this method, within the monitor,
 * there should be no risk for a second caller interfering with the
 * first (for instance when wait is called by the first). A solution
 * with internal synchronization blocks (instead of the method being
 * synchronized) is fully acceptable, and efficiency is encouraged.
 */
public void x() {
  // Your code goes here...
}

private void x0() { // No need to know the internals:
  // Deal with networking hardware; you don't need to know how...
}
```

```
private void x1() { // No need to know the internals:
  // Handle network header response...
}

private void x2() { // No need to know the internals:
  // Handle network packet response...
}

// ------------------------------------------------- Worker and Method y:

static class AnsY { /* Some data... for storing algorithm output... */
}

private AnsY theAnsY; // Storage for computations as requested by method y.

/**
 * Computation to be carried out upon call of method y, but it is uncertain if
 * a results will be delivered on time and hence the handling by methods y and
 * w is necessary.
 */
private AnsY yWork() { // No need to know the internals:
  // Algorithm taking short or long time, hopefully converging...
  return new AnsY();
}

/**
 * Minimal and predictable computation to be carried out upon call of method
 * y. It is guaranteed that this method completes well within WCET_Y.
 */
private AnsY yQuick() { // No need to know the internals:
  // Quick and predictable algorithm...
  return new AnsY();
}

/**
 * Method to be called by worker threads, or at least one worker thread as
 * needed for the timing supervision of method y. Only one thread at a time
 * may act as a worker thread, so additional calls will be blocked. This
 * method (the single thread running it) will work as long as the call of
 * yWork is successful (meeting its deadline WCET_Y), but upon failure to meet
 * the deadline the calling worker thread will be interrupted and then run
 * until termination (or it will run forever but with lowest priority).
 */
void w() { // synchronized or not, depending on your design.
  // Your code here...
}

// Additional methods for the worker, here, if needed in your design.

/**
 * Let a single thread at a time call the private (not monitor protected)
 * yWork method, but if that once results in a missed deadline (result
 * available after WCET_Y microseconds) then always call the corresponding
 * yQuick method (which has an ensured timing that does not need to be
 * checked) instead. The algorithm output (of type AnsY) is taken care of
 * internally in the monitor (simply store it in attribute theAnsY).
 *
 * @return true if yWork was called and completed on time, false otherwise.
 */
public synchronized boolean y() {
  // For you to implement...
```

```
  }

  // ------------------------------------------------------------ Method z:

  /**
   * Perform (one job of) periodic control, by calling the control algorithm
   * once. The control, no matter what preemption or priorities that apply,
   * should be completed within WCET_Z microseconds (checked with the
   * granularity of the 1ms clock tick).
   */
  public synchronized void z() {
    // Also for you to implement
  }

  private void zControl() { // Available method to be called from z.
    // Perform feedback control
  }

  // ------------------------------------------------------------ Supervisory:

  /**
   * Supervise timing and execution, by checking reported delays in methods x, y
   * and z, or in method w for y. Also checks that the caller has MAX_PRIORITY.
   * Returns when an error occurs. Every second, check that z has been called
   * at least once during the last 200ms (the period is never longer than that).
   *
   * @return the error as a string stating the missed deadline. If the periodic
   *         calling of method z stops, an Error is thrown.
   */
  synchronized String s() {
    // Here goes the last part of your implementation...
  }
}
```

8. Timed message dispatching

Consider the problem in previous problem, and suppose you want to re-implement the system using message-based communication only. You make use of the fact that method `x` is only used by the (almost) highest priority thread (so the response time for that thread can be ensured, if just buffering effects can be avoided). For the threads calling methods `y` and `z`, however, you relax the schedulability and instead the run-time checking will detect missed deadlines.

A thread that requests a method call (by sending an event, which is timestamped marking the start time of the call) and then returns after the WCET for that method, should be informed by an event (`RTEvent` back to the source) and by an error (`throw RTDelayed`, which is a `Throwable` of type `Error`), in that order (reply event sent before `throw`, so the `catch` can use the `mailbox`). Describe and illustrate how you would solve the problem, but no implementation is required.

(5p)