Exam – Solutions
EDA040/DATE14
Concurrent and Real-Time Programming

2012–01–09, 08.00–13.00

---

1. a) The job of a scheduler is to choose which thread should run.

   b) The scheduler marks all threads it handles. Running: the thread is currently running. Ready: the thread is announced to the scheduler as ready to run, but has not yet been selected for running. Blocked: the thread is not ready for running and will therefore not be selected by the scheduler.

2. Calculation of blocking times:

   - A can be blocked by B and D via M1, $B_A = max(b2, d1, d2) = 0.4$
   - B can be blocked by C via M2 and by D via M1, $B_B = c + max(d1, d2) = 0.7 + 0.3 = 1.0$
   - C can be blocked by D via M1, $B_C = max(d1, d2) = 0.3$
   - D cannot be blocked, $B_D = 0$

   Calculation of worst case response times:

   - $R_A = C_A + B_A = 1 + 0.4 = 1.4$
   - $R_B = 6 > T_B$, the system is not schedulable.

$$
\begin{aligned}
R_B^1 &= C_B + B_B = 2 + 1 = 3 \\
R_B^2 &= C_B + B_B + \left\lceil \frac{R_B^1}{T_A} \right\rceil C_A = 2 + 1 + \lceil \tfrac{3}{2} \rceil 1 = 5 \\
R_B^3 &= 2 + 1 + \lceil \tfrac{5}{2} \rceil 1 = 6 \\
R_B^4 &= 2 + 1 + \lceil \tfrac{6}{2} \rceil 1 = 6
\end{aligned}
$$

3. a) Signalling: a thread passes a flag (signal) to another thread. Mutual exclusion: a thread holds the flag access to a resource. All threads accessing the resource need to hold the flag, but only one thread at a time can do it.

   b) Extra checks could be made to ensure that the same thread takes and gives back the flag, to catch misuse.

4. a) Deadlock.

   b) One left handed philosopher or only three chairs.

   Refer to lecture slides for the graph and suitable code corrections.

5. Start by calculating response times:

   - $R_A = C_A + B_A = 2 + 4 = 6 < T_A$
   - $R_B = 10$

     $$\begin{aligned} R_B^1 &= C_B + B_B = 2 + 4 = 6 \\ R_B^2 &= C_B + B_B + \left\lceil \frac{R_B^1}{T_A} \right\rceil C_A = 2 + 4 + \lceil \tfrac{6}{7} \rceil 2 = 8 \\ R_B^3 &= 2 + 4 + \lceil \tfrac{8}{7} \rceil 2 = 10 \\ R_B^4 &= 2 + 4 + \left\lceil \frac{10}{7} \right\rceil 2 = 10 \end{aligned}$$

     Since we are must choose T_B in the range 8-10, this means $T_B = 10$.

   - $R_C = 14$

     $$\begin{aligned} R_C^1 &= C_C + B_C = 2 + 4 = 6 \\ R_C^2 &= C_C + B_C + \left\lceil \frac{R_C^1}{T_A} \right\rceil C_A + \left\lceil \frac{R_C^1}{T_B} \right\rceil C_B = 2 + 4 + \lceil \tfrac{6}{7} \rceil 2 + \lceil \tfrac{6}{10} \rceil 2 = 10 \\ R_C^3 &= 2 + 4 + \left\lceil \frac{10}{7} \right\rceil 2 + \left\lceil \frac{10}{10} \right\rceil 2 = 12 \\ R_C^4 &= 2 + 4 + \left\lceil \frac{12}{7} \right\rceil 2 + \left\lceil \frac{12}{10} \right\rceil 2 = 14 \\ R_C^5 &= 2 + 4 + \left\lceil \frac{14}{7} \right\rceil 2 + \left\lceil \frac{14}{10} \right\rceil 2 = 14 \end{aligned}$$

     So $T_C \geq 14$.

   Next look at cpu utilization. The utilization for threads A-C must be below 60% for the bandwidth requirement of D to be fulfilled. Using the provided table to calculate utilization we get:

   $T_C = 14$: $U_{A-C} = \frac{2}{7} + \frac{2}{10} + \frac{2}{14} = 2 * 0.15 + 0.2 + 2 * 0.072 = 0.644 > 0.6$

   Successively testing larger numbers for T_C gives the smallest value below or equal to 60%:

   $T_C = 20$: $U_{A-C} = \frac{2}{7} + \frac{2}{10} + \frac{2}{20} = 2 * 0.14 + 0.2 + 2 * 0.050 = 0.6$

   Note: if calculating more exact (not using the fraction table), T_C can be lowered to 18.

   Choose $T_D = 21$ and calculate maximum C_D by assuming $R_D = T_D$:

   $$R_D = C_D + B_D + \left\lceil \frac{R_D}{T_A} \right\rceil C_A + \left\lceil \frac{R_D}{T_B} \right\rceil C_B + \left\lceil \frac{R_D}{T_C} \right\rceil C_C$$

   $$21 = C_D + \left\lceil \frac{21}{7} \right\rceil 2 + \left\lceil \frac{21}{10} \right\rceil 2 + \left\lceil \frac{21}{20} \right\rceil 2$$

   $$C_D = 5$$

6. a) J6 gives a more fine-grained switching between (yielding) threads of the same priority. J5 doeas not let a very time-consuming thread take as much CPU time, which in some application might be preferable.

   b) No, this is more of Round-Robin scheduling (FIFO-like queueing), which is not real-time suitable. (For real-time, other versions exist.)

   c) To tell where a context switch is suitable or even desirable. (By having the highest priority threads short and without yield, and the less critical threads calling yield after each (short) part of their work, a program can be portable between preemptive and non-preemptive scheduling.)

7. The following implementation is not perfect (and not really tested), in particular not methods x and s, but it shows one approach to a solution.

```
public class M {
  // Some attributes, see usage further below.
  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();}

// Helper methods omitted for brevity.

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

/** See doc/spec in exam. */
public void x() {
  long t, t0, t1, t2, t3, t4;
  synchronized (xLock) {
    synchronized (this) {
      t0 = RTSystem.executionTimeMicros();
      x0();
      t1 = RTSystem.executionTimeMicros();
      while (RTSystem.executionTimeMicros() - t1 < 800)
        await(1);
      t2 = RTSystem.executionTimeMicros();
      if (t2 - t1 > 2000)
        xLate++;
      x1();
      t3 = RTSystem.executionTimeMicros();
      x1();
      while (RTSystem.executionTimeMicros() - t3 < 600)
        await(1);
      t4 = RTSystem.executionTimeMicros();
      x2();
      if (t4 - t3 > 2000)
        xLate++;
      if (RTSystem.executionTimeMicros() - t0 > WCET_X)
        xLate++;
      if (Thread.currentThread().getPriority() < Thread.MAX_PRIORITY - 1)
        xLate = 0;
      else if (xLate > 0)
        notifyAll();
      while (xLate > 0)
        await();
    }
  }
}

private void x0() { /* No need to know the internals: */ }
private void x1() { /* No need to know the internals: */ }
private void x2() { /* No need to know the internals: */ }

// ------------------------------------------------------------- Method y:

// Type def of AnsY (static class AnsY) and attribute for storage (theAnsY)
// kept (omitted for brevity), just as methods yWork and yQuick.

/** See doc/spec in exam. */
void w() {
  AnsY ansY;
  w1();
  do {
    ansY = yWork();
  } while (w2(ansY) == 0);
```

```
  // Deadline missed; wait for termination..
}

private synchronized void w1() {
  while (worker != null)
    await();
  worker = Thread.currentThread();
  while (!todo)
    await();
}

private synchronized int w2(AnsY yArg) {
  theAnsY = yArg;
  todo = false;
  notifyAll();
  return yLate;
}

/** See doc/spec in exam. */
public synchronized boolean y() {
  long t0 = System.currentTimeMillis();
  if (yLate > 0) {
    yQuick();
    return false;
  }
  if (worker == null)
    throw new RTError("No worker ready");
  todo = true;
  notifyAll();
  while (todo && System.currentTimeMillis() < t0 + WCET_Y / 1000)
    await(t0 + WCET_Y / 1000 - System.currentTimeMillis());
  if (todo) {
    yLate++;
    worker.setPriority(Thread.MIN_PRIORITY);
    worker.interrupt();
    worker = null;
    yQuick();
    notifyAll();
    return false;
  } else {
    return true;
  }
}

// ------------------------------------------------------------- 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() {
  long zTime = System.currentTimeMillis();
  zControl();
  if (System.currentTimeMillis() - zTime > WCET_Z / 1000) {
    zLate++;
    notifyAll();
  }
}

private void zControl() { /* 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.
   *
   * @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() {
    String ans = "";
    if (Thread.currentThread().getPriority() < Thread.MAX_PRIORITY) ans = "p";
    sTime = System.currentTimeMillis();
    do {
      sTime += 1000;
      while (xLate == 0 && yLate == yLateOld && zLate == zLateOld) {
        await(sTime - System.currentTimeMillis());
      }
      if (xLate > 0) {
        ans = "x";
        xLate = 0;
        notifyAll();
      }
      ;
      if (yLate != yLateOld) {
        ans = "y";
        yLateOld = yLate;
        notifyAll();
      }
      if (zLate != zLateOld) {
        ans = "z";
        yLateOld = zLate;
        notifyAll();
      }
      if (zTime < sTime - 200) {
        throw new RTError("Stall alarm");
      }
    } while (ans == "");
    return ans;
  }
}
```

8. No solution provided, but in principle a design for message-based communication need to treat high-priority activities as special cases, e.g. by picking those messages before posting and putting them into some type of high-priority buffer. That is, even if threads are scheduled based on priorities, buffering messages imposes a strict FIFO order (with high-priority messages being behind messages from lower-priority threads), and hence the priorities are effectively neglected.

   To manage missed deadlines (the returned event and then throwing the Error as stated in the problem), and for waiting on the return value, the calling thread has to be blocked after posting (like wait in putEvent) the message, and then upon notification the Error has to be thrown to the calling thread. (Recall, catching exceptions pops the call-stack of a thread, and hence exceptions do not work between threads.)

   The credits depends on how these issues are handled, and how well a figure (and documentation) explains a suitable design.