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

2011–10–22, 08.00–13.00

1. a) The running thread continues until it voluntarily releases the CPU. Pro: potentially faster context switching since the switch occurs at known points (with a known context). Con: context switching need to be taken care of manually.
   b) A context switch can only occur at certain points defined by language or runtime system. Pro: same as above. Con: context switching depends on the language and/or runtime system.
   c) Hardware interrupts initiate context switching. Pro: gives best timing guarantees of the three pre-emption methods. Con: potentially slower context switching since it is not known when a switch will occur (all context must be saved all the time).

2. Calculation of blocking times:
   - A can only be blocked by C via M1, $B_A = max(c1, c2) = 0.2$
   - B can be blocked by C via M1 and by D via M2, $B_B = max(c1, c2) + d = 0.2 + 0.7 = 0.9$
   - C can be blocked by D via M2, $B_C = 0.7$
   - D cannot be blocked, $B_D = 0$

   Calculation of worst case response times:
   - $R_A = C_A + B_A = 1 + 0.2 = 1.2$
   - $R_B = 3.9$

$$\begin{aligned}
R_B^1 &= C_B + B_B = 1 + 0.9 = 1.9 \\
R_B^2 &= C_B + B_B + \left\lceil \frac{R_B^1}{T_A} \right\rceil C_A = 1 + 0.9 + \left\lceil \frac{1.9}{2} \right\rceil 1 = 2.9 \\
R_B^3 &= 1 + 0.9 + \left\lceil \frac{2.9}{2} \right\rceil 1 = 3.9 \\
R_B^4 &= 1 + 0.9 + \left\lceil \frac{3.9}{2} \right\rceil 1 = 3.9
\end{aligned}$$

   - $R_C = 9.7$

$$\begin{aligned}
R_C^1 &= 2 + 0.7 = 2.7 \\
R_C^2 &= 2.7 + \left\lceil \frac{2.7}{2} \right\rceil 1 + \left\lceil \frac{2.7}{5} \right\rceil 1 = 5.7 \\
R_C^3 &= 2.7 + \left\lceil \frac{5.7}{2} \right\rceil 1 + \left\lceil \frac{5.7}{5} \right\rceil 1 = 7.7 \\
R_C^4 &= 2.7 + \left\lceil \frac{7.7}{2} \right\rceil 1 + \left\lceil \frac{7.7}{5} \right\rceil 1 = 8.7 \\
R_C^5 &= 2.7 + \left\lceil \frac{8.7}{2} \right\rceil 1 + \left\lceil \frac{8.7}{5} \right\rceil 1 = 9.7 \\
R_C^6 &= 2.7 + \left\lceil \frac{9.7}{2} \right\rceil 1 + \left\lceil \frac{9.7}{5} \right\rceil 1 = 9.7
\end{aligned}$$

   - $R_D = 14$

$$\begin{aligned}
R_D^1 &= 2 \\
R_D^2 &= 2 + \left\lceil \frac{2}{2} \right\rceil 1 + \left\lceil \frac{2}{5} \right\rceil 1 + \left\lceil \frac{2}{15} \right\rceil 2 = 6 \\
R_D^3 &= 2 + \left\lceil \frac{6}{2} \right\rceil 1 + \left\lceil \frac{6}{5} \right\rceil 1 + \left\lceil \frac{6}{15} \right\rceil 2 = 9 \\
R_D^4 &= 2 + \left\lceil \frac{9}{2} \right\rceil 1 + \left\lceil \frac{9}{5} \right\rceil 1 + \left\lceil \frac{9}{15} \right\rceil 2 = 11 \\
R_D^5 &= 2 + \left\lceil \frac{11}{2} \right\rceil 1 + \left\lceil \frac{11}{5} \right\rceil 1 + \left\lceil \frac{11}{15} \right\rceil 2 = 13 \\
R_D^6 &= 2 + \left\lceil \frac{13}{2} \right\rceil 1 + \left\lceil \frac{13}{5} \right\rceil 1 + \left\lceil \frac{13}{15} \right\rceil 2 = 14 \\
R_D^7 &= 2 + \left\lceil \frac{14}{2} \right\rceil 1 + \left\lceil \frac{14}{5} \right\rceil 1 + \left\lceil \frac{14}{15} \right\rceil 2 = 14
\end{aligned}$$

3. a) Less complex code since memory management is handled automatically. No memory leaks.

   b) It is not easy to predict WCET for garbage collection, since it depends on the allocation history of the program. In realtime systems, GC work should be avoided when high-priority threads execute. This can be achieved, for instance by viewing memory as a resource, and performing GC work in the pauses between high-priority invocations using a low-priority thread with minimum time spent locking the memory resource. Of course this depends on enough CPU low-priority bandwidth being available in the system.

4. a) Consider the 4ms delay as blocking for all threads, i.e. $B_{A-D} = 4$. The shortest period and deadline for B and C is equal to the response time. Since a desirable priority order is given, we need to make sure $D_A < D_B < D_C$. This is the case.

   - $R_A = 2 + 4 = 6$
   - $T_B = D_B = R_B = 8$

   $$\begin{aligned} R_B^1 &= 2+4=6 \\ R_B^2 &= 6+\left\lceil \tfrac{6}{9} \right\rceil 2 = 8 \\ R_B^3 &= 6+\left\lceil \tfrac{8}{9} \right\rceil 2 = 8 \end{aligned}$$

   - $T_C = D_C = R_C = 14$

   $$\begin{aligned} R_C^1 &= 2+4=6 \\ R_C^2 &= 6+\left\lceil \tfrac{6}{9} \right\rceil 2+\left\lceil \tfrac{6}{8} \right\rceil 2 = 10 \\ R_C^3 &= 6+\left\lceil \tfrac{10}{9} \right\rceil 2+\left\lceil \tfrac{10}{8} \right\rceil 2 = 14 \\ R_C^4 &= 6+\left\lceil \tfrac{14}{9} \right\rceil 2+\left\lceil \tfrac{14}{8} \right\rceil 2 = 14 \end{aligned}$$

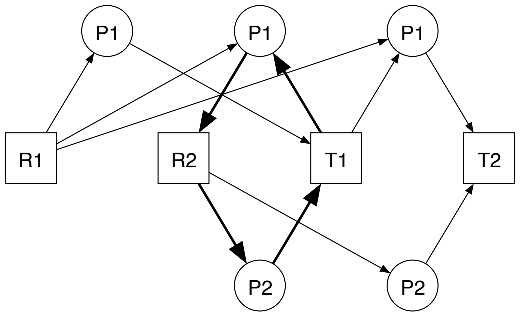
   b) The longest possible C_D occurs when $R_D = T_D = 10T_A = 90$, so

   $$\begin{aligned} R_D &= C_D + B_D + \textstyle\sum_{j\in A,B,C} \left\lceil \frac{R_D}{T_j} \right\rceil C_j \\ 90 &= C_D + 4 + \left\lceil \tfrac{90}{9} \right\rceil 2 + \left\lceil \tfrac{90}{8} \right\rceil 2 + \left\lceil \tfrac{90}{14} \right\rceil 2 \\ 90 &= C_D + 4 + 10\cdot 2 + 12\cdot 2 + 7\cdot 2 \end{aligned}$$

   making $C_D = 28$

5. a) Livelock

   b) Resource allocation graph with potential deadlock marked in bold:



One resolution to the deadlock problem is to lock T1 before R2 in the P2 process. This would however make useT2 dependent on T1 (even though T1 is not required for the operation). Since the useXX methods are independent, a possible improvement of the programs is to minimize the "locking distances". This would ensure minimal interference between the two robots. For instance:

```
P1                                      P2
------                                  ----------
takeR1 // me working                    takeT2
   takeT1                                 takeR2
    takeT2                                useT2
    useW1T1T2                             giveR2
    giveT2                              giveT2
  giveT1                                helpR1 // potential assist
giveR1                                  takeT1
takeR1                                    takeR2
  takeT1                                  useT1
    takeR2                                giveR2
    useW1T1R2 // assisted               giveT1
    giveR2                              helpR1 // potential assist
   giveT1
giveR1 // me done
```

6.
```
public class M {
  Thread aThread = null;

  // Requirement 7 & 8
  S s;
  H h;

  public M() {
    s = new S();
    h = new H(s);
    h.setPriority(Thread.MAX_PRIORITY);
    h.start();
  }

  // Requirement 7
  public String supervisor() throws InterruptedException {
    return s.supervisor();
  }

  // Requirement 6
  private long bET = 0;

  public void a() {
    final int wcet = 2;
    final int blocking = 6;

    // Requirement 3
    long sysStart = System.currentTimeMillis();
    synchronized (this) {
      long exeStart = RTSystem.executionTimeMicros();

      // Requirement 1
      if (aThread == null)
        aThread = Thread.currentThread();
      else if (aThread != Thread.currentThread()) {
        String err = "Call rejected";
        s.error(err); // Requirement 7
        throw new RTError(err);
      }
      // Requirement 2
      if (Thread.currentThread().getPriority() != Thread.MAX_PRIORITY) {
        String err = "Wrong a priority";
        s.error(err); // Requirement 7
        throw new RTError(err);
      }

      // ... a executing

      // Requirement 3
      long sysEnd = System.currentTimeMillis();
      long exeEnd = RTSystem.executionTimeMicros();
      long et = (exeEnd - exeStart) / 1000;
      long st = sysEnd - sysStart;
      if (et > wcet || st > wcet + blocking) {
        String err = "Method a too late, possibly blocked by b (execution time): " + bET;
        s.error(err); // Requirement 7
        throw new RTError(err);
      }
    }
  }

  public synchronized void b() {
    final int wcet = 4;

    // Requirement 4
    long exeStart = RTSystem.executionTimeMicros();

    // Requirement 2
    if (Thread.currentThread().getPriority() == Thread.MAX_PRIORITY) {
      String err = "Wrong b priority";
      s.error(err); // Requirement 7
      throw new RTError(err);
    }

    // Requirement 8
    s.bStart();

    // ... b executing

    // Requirement 8
    s.bEnd();

    // Requirement 4
    long exeEnd = RTSystem.executionTimeMicros();
    long et = (exeEnd - exeStart) / 1000;
    if (et > wcet) {
      String err = "Method b too late";
      s.error(err); // Requirement 7
      throw new RTError(err);
    }

    // Requirement 6
    bET = et;
  }
  public synchronized void c() throws InterruptedException {
    final int wcet = 12;
    boolean condition = false;

    // Requirement 4
    long exeStart = RTSystem.executionTimeMicros();

    // Requirement 2
    if (Thread.currentThread().getPriority() == Thread.MAX_PRIORITY) {
      String err = "Wrong c priority";
      s.error(err); // Requirement 7
      throw new RTError("Wrong c priority");
    }

    // ... c executing first half

    // Requirement 5
    long wStart = RTSystem.executionTimeMicros();

    // c waiting condition
    while (!condition) {
      wait();
      // Update condition
    }

    // Requirement 5
    long wEnd = RTSystem.executionTimeMicros();
    long delay = (wEnd - wStart) / 1000;

    // ... c executing second half

    // Requirement 4
    long exeEnd = RTSystem.executionTimeMicros();
    long et = (exeEnd - exeStart) / 1000;
    if (et > wcet) {
      String err = "Method c too late, delay " + delay;
      s.error(err); // Requirement 7
      throw new RTError(err);
    }
  }
}

// Requirement 7 & 8
public class S {
  private String err = null;
  private boolean started = false;
  private boolean ended = false;

  public synchronized void error(String msg) {
    err = msg;
    notifyAll();
  }

  public synchronized void bStart() {
    started = true;
    notifyAll();
  }

  public synchronized void bEnd() {
    ended = true;
    notifyAll();
  }

  public synchronized String supervisor() throws InterruptedException {
    while (err == null) wait();
    return err;
  }

  public synchronized void helper() {
    long bR = 14;
    while (true) {
      try {
        while (!started) wait();
        long sysStart = System.currentTimeMillis();
        while (!ended) {
          long dt = System.currentTimeMillis() - sysStart;
          wait(bR - dt);
        }
        long sysEnd = System.currentTimeMillis();
        long st = sysEnd - sysStart;
        if (st > bR) {
          err = "b overrun";
          notifyAll();
        }
        started = false;
        ended = false;
      } catch (InterruptedException e) { return; }
    }
  }
}

//Requirement 8
public class H extends Thread {
  S s;

  public H(S s) {
    this.s = s;
  }

  public void run() {
    s.helper();
  }
}
```

7. No solution provided.