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

2009–12–16, 08.00–13.00

---

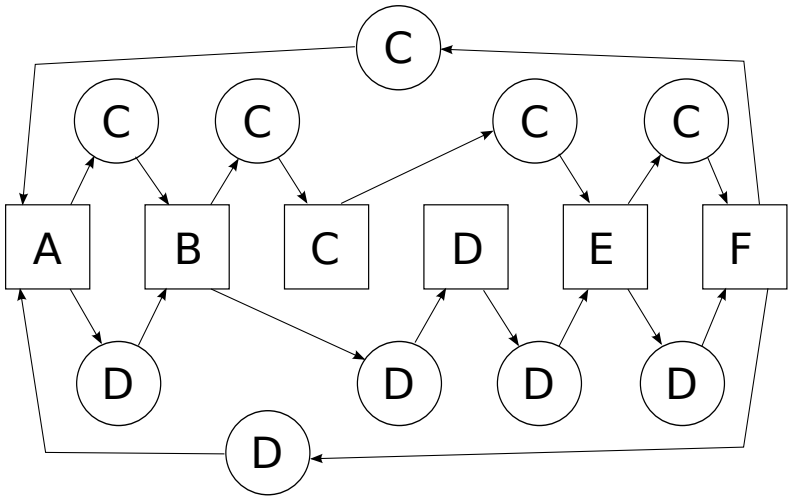
1. a) Pre-emption is required since otherwise there could be some other thread (not part of your design) that continues to run without giving up the CPU (by any call resulting in rescheduling, such as wait, sleep, yield, etc.), and hence your deadlines would be missed.

   b) To ensire consistency of your software, reflecting the fact that threads are collaborating activites (although competing for utilization of some resources such as CPU tme), a locked resource/-monitor may only be unlocked by the locking thread. Thus, we should have no pre-emption on resources.

   c) Schedulability analysis assumes strict priorities, which therefore is required. A real-time application needs to run on an real-time operating system.

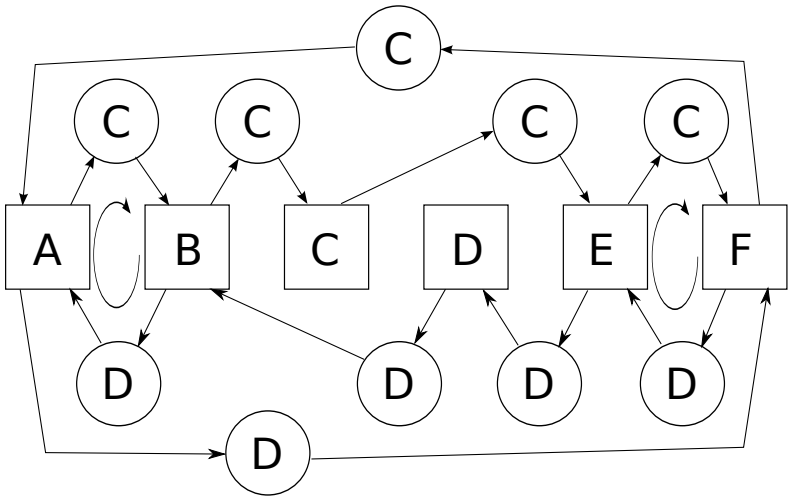
   d) Priority inheritance is required to avoid priority inversion, i.e. to get bounded respose time for your high-priority threads (that otherwise could be blocked on lower priority threads that cannot continue due to some medium-priority thread).

2. a) There are L trains and R trains that go left and righ. Here we call these D and C for track-D-train and track-C-train respectively. The resource allocation graph is shown in the following figure.

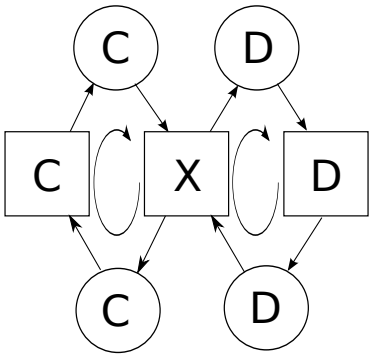


   The graph tells us that with five trains of either type C or D implies a risk for deadlock, and it is easy to see how that can happen. With a mix of train types it is basically the same, except that a single C-train on track C or a single D-train on track D will not create an additional hold-wait loop. However, in such a case that train cannot leave its track (where it would be live-locked). In conclusion, maximum four trains can traffic the tracks.

   b) There are four loops in the allocation graph, but with only one train of each type only the B-A, E-F, and F-A (not marked in figure) loops apply. Hence there are three ways they can deadlook (when meeting on those pairs of tracks).



c) Lump resources A, B, E, and F into a resource X, which is to be locked before use. As clear from the graph, a risk of deadlock requires at least two instances of one type of train. Thus one train of each type and (atomic) allocation of the lumped resource X will be deadlock free.



3. There are two types of blocking that we will have to consider – direct blocking and push-through blocking. They can occur alone or in tandem. We analyze the threads one by one:

Thread T1
Thread T1 is the highest priority thread and can therefore only encounter direct blocking. The only possible scenario is that T3 has locked monitor M1 by calling `c()`. Thanks to priority inheritance, the blocking time for T1 can be at most 0.2 ms in this case. Therefore, the maximum blocking time for T1 is *0.2 ms*.

Thread T2
Thread T2 can, in the same way as T1 above, encounter direct blocking from T4 when trying to enter M2. The maximum blocking time is here 0.1 ms (method `d()`). However, T2 can in addition encounter push-through blocking if T3 has locked M1 while T1 tries to enter it. If that happens, T3 will temporarity inherit the priority of T1 and block T2 due to its elevated priority. T3 can run with an elevated priority for at most 0.2 ms (method `c()`). To summarize, T2 can be blocked *both* by direct blocking by T4 and by push-through blocking by T3. The maximum blocking time for T2 is thus 0.1 + 0.2 = *0.3 ms*.

Thread T3
Thread T3 can only encounter push-through blocking since it does not directly communicate with any other thread through a monitor. In this case it is T4 that causes the push-through blocking when it has locked monitor M2 (method `d()`) while T2 tries to enter it. The maximum time that T4 can run with the priority of T2 is 0.1 ms. Therefore, the maximum blocking time for T3 is *0.1 ms*.

Thread T4
Since T4 is the lowest priority thread in the system, there are no lower priority threads that can block it. The maximum blocking time for T4 is thus trivially *0 ms*.

4. First, we need to determine the priority order for the threads. Since rate monotonic scheduling states that priority should be assigned according to period, we get the following priority order (from highest to lowest): B – D – A – C.

   Calculating the worst-case response times simply involves iteratively evaluating the given formula until it stabilizes for each thread. We use $R_i^0 = 0$ as a starting value for each iteration.

   Thread B

   $R_B^0 = 0$
   $R_B^1 = 1 + 0.5 = 1.5$
   $R_B^2 = 1 + 0.5 = 1.5$

   The worst-case response time for thread B is 1.5 ms. Since 1.5<4, thread B always meets its deadline.

   Thread D

   $R_D^0 = 0$
   $R_D^1 = 1 + 1 + \lceil \frac{0}{4} \rceil 1 = 2$
   $R_D^2 = 1 + 1 + \lceil \frac{2}{4} \rceil 1 = 3$
   $R_D^3 = 1 + 1 + \lceil \frac{3}{4} \rceil 1 = 3$

   The worst-case response time for thread D is 3 ms. Since 3<6, thread D always meets its deadline.

   Thread A

   $R_A^0 = 0$
   $R_A^1 = 2 + 1 + \lceil \frac{0}{4} \rceil 1 + \lceil \frac{0}{6} \rceil 1 = 3$
   $R_A^2 = 2 + 1 + \lceil \frac{3}{4} \rceil 1 + \lceil \frac{3}{6} \rceil 1 = 5$
   $R_A^3 = 2 + 1 + \lceil \frac{5}{4} \rceil 1 + \lceil \frac{5}{6} \rceil 1 = 6$
   $R_A^4 = 2 + 1 + \lceil \frac{6}{4} \rceil 1 + \lceil \frac{6}{6} \rceil 1 = 6$

   The worst-case response time for thread A is 6 ms. Since 6<10, thread A always meets its deadline.

   Thread C

   $R_C^0 = 0$
   $R_C^1 = 2 + 0 + \lceil \frac{0}{4} \rceil 1 + \lceil \frac{0}{6} \rceil 1 + \lceil \frac{0}{10} \rceil 2 = 2$
   $R_C^2 = 2 + 0 + \lceil \frac{2}{4} \rceil 1 + \lceil \frac{2}{6} \rceil 1 + \lceil \frac{2}{10} \rceil 2 = 6$
   $R_C^3 = 2 + 0 + \lceil \frac{6}{4} \rceil 1 + \lceil \frac{6}{6} \rceil 1 + \lceil \frac{6}{10} \rceil 2 = 7$
   $R_C^4 = 2 + 0 + \lceil \frac{7}{4} \rceil 1 + \lceil \frac{7}{6} \rceil 1 + \lceil \frac{7}{10} \rceil 2 = 8$
   $R_C^5 = 2 + 0 + \lceil \frac{8}{4} \rceil 1 + \lceil \frac{8}{6} \rceil 1 + \lceil \frac{8}{10} \rceil 2 = 8$

   The worst-case response time for thread C is 8 ms. Since 8<20, thread D always meets its deadline.

5. The run method of a thread object should not be synchronized, because a thread is expected to be able to call sleep for waiting on time without then looking its communication with other threads. This is assumed to be standard and need not be mentioned. More centrally, a monitor only works if the locking protocol (such as declaring methods synchronized) is followed by all (mutually exclusive) methods, and when those methods are actually used for accessing the data. For such a proper way of access to be checked, both encapsulated data and any non-synchronized method should be private or protected, so the only way to acess shared data is via public och package-visible methods. By separating active objects (threads) and shared data (the monitors) we gen help from the compiler to check that data is only acessed properly.

   a) To avoid accidential access of shared data from the run method; To get assistance from the compiler for checking access to shared data.

   b) To prevent illegal access of shared data from other methods (in other classes, possibly within the same package).

   c) A sub.class does not inherit synchronized, and most likely looking the object is still necessary since the base-class (shared) data is still there.

   d) With wait you temporarily leave the monitor, and the (actually co-designed) monitor methods of the base class may depend on that the base-class method does not result in that the monitor is left. By declaring it final, alternative sub-class implementations are prevented (in the cases where this is crucial).

6. Synchronized Traffic Light Control

a)

The interfaces for awaiting vehicles and for awaiting commmands are both blocking and need to be served by one thread each, which we name `StreetHandler` and `CommandHandler` respectively. The latter will additionally block inside the monitor since it has to wait for some servicing thread to acknowledge the command. Since the input threads are blocking they are not suitable for directly controlling the lights, which instead is done by the service thread.

We call the service thread `LightHandler`, whch either has to obtain light-change requests as return values from the monitor, or the monitor method (named `service`) can set thelights directly from within the monitor. We choose the latter, for simplicity since the `setColor` method is non-blocking. That is, the `service()` method needs no arguments and no return value. Actually, it never needs to return (and we do not even need a `while`-loop in the `LightHandler` thread as can be seen below).

Note that for ensurance of the light sequencing, it would be a very bad design to have more than one service thread (like one for each street, or one for each mode). Compare with the sequencing of the elevator lab, where we also learned that it is better with a few large monitor methods that make sense from an application point of view. That way the concurrency issues are better managed within the monitor (using `wait` and notify, and proper logic), instead of dealing with method call sequences that are subject to change (and scheduling) elsewhere.

There is much timing involved but it is all about minimum times so we can use `wait(timeout)` in the monitor, which is (simpler than the) equivalent solution of having an external timer thread that calls notify when time is out. Note that while (extra) calls of `notifyAll` are nothing problematic in most monitors, we have to take some extra care when we use `wait(timeout)` for sleeping, as encoded in the `await` methods below.

Since the service thread will maintain the light state in terms of it's excution state (where in the sequences of `setColor` it is), the only needed monitor state in terms of attributes is the mode (current, previous, and next) and the current direction (that has green, or red, depending on implementation), plus some car counting values if statistics och fairness are to be implemented.

As part of the design, here are the full implementations of the threads (formatted for brevity):

```
class CommandHandler extends Thread {
   Monitor mon;
   CommandHandler(Monitor mon) {this.mon = mon;}
   public void run() {
        char cmd;
        while (!isInterrupted()) {
            cmd = CommandInput.awaitCommand(cmd);
            mon.setMode(cmd);}}}

class StreetHandler extends Thread {
   Monitor mon;
   StreetInput.Value cnt = new StreetInput.Value();
   StreetHandler(Monitor mon) {this.mon = mon;}
   public void run() {
        while (!isInterrupted()) {
            cnt = StreetInput.awaitVehicle(cnt);
            mon.setCount(cnt);}}}

class LightHandler extends Thread {
   Monitor mon;
   LightHandler(Monitor mon) {this.mon = mon;}
   public void run() { mon.service(); }}
```

As intended, threads are shorts and simple, calling a few larger monitor operations that make sense from the application point of view. It was sufficient to scetch the connection between the IO methods with the monitor methods via a figure or similar, and to comment shortly on the design. That is, the thread implementations or the longer explainations above were not required.

b)
The actual monitor then looks like the following, starting with the attributes representing the monitor state, followed by the three methods (that are called from the threads) of which the first two are trivial and the third is the comprises the basic service of the entire application (made trivial by putting the logic in other methods while keeping the top-level state machine in the service method):

```
public class Monitor {

    char mode = 'w';
    char next = 'w';
    char prev = 'w'; // Previous mode to return to after alarm.
    boolean ns;      // north-south should have green.
    StreetInput.Value count;

    synchronized public void setMode(char m) {
        next = m;
        notifyAll();
        try {
            while (mode != next) wait();
        } catch (InterruptedException exc) {//ignore}
    }

    synchronized void setCount(StreetInput.Value cnt) {
        count = cnt;
        if (mode == 'e' || next == 'e') notifyAll();
    }

   // THE SERVICE, A SMALL STATE MACHINE ASSUMING HELPER METHODS:

    synchronized void service() {
        while (!Thread.currentThread().isInterrupted()) {
            switch (next) {
            case 'w':
                warning();
                if (next == 'a') { prev = mode; }
                break;
            case 'a':
                alarm();
                break;
            case 't':
                time();
                if (next == 'a') { prev = mode; }
                break;
            case 'e':
                event();
                if (next == 'a') { prev = mode; }
                break;
            default:
                return;
            }
        }
    }

    // PRIVATE HELPER METHODS:

    private void NS(String ryg) {
        Light.setColor('N', ryg.charAt(0), ryg.charAt(1), ryg.charAt(2));
        Light.setColor('S', ryg.charAt(0), ryg.charAt(1), ryg.charAt(2));
    }
```

```
private void EW(String ryg) {
    Light.setColor('E', ryg.charAt(0), ryg.charAt(1), ryg.charAt(2));
    Light.setColor('W', ryg.charAt(0), ryg.charAt(1), ryg.charAt(2));
}

private void await(long time, char m) {
    long t = System.currentTimeMillis();
    long tf = t + time;
    try {
        while (next == m && t < tf) {
            wait();
            t = System.currentTimeMillis();
        }
    } catch (InterruptedException exc) {// Ignore}
}

private void await(long time) {
    long t = System.currentTimeMillis();
    long tf = t + time;
    try {
        while (t < tf) {
            wait();
            t = System.currentTimeMillis();
        }
    } catch (InterruptedException exc) {// Ignore}
}

private void warning() {
    NS(" Y "); EW(" Y ");
    await(2000);
    mode = next;
    notifyAll();
    while (next == 'w') {
        NS("   "); EW("   ");
        await(500);
        NS(" Y "); EW(" Y ");
        await(500);
    }
}

private void alarm() {
    NS(" Y "); EW(" Y ");
    await(2000);
    NS("R  "); EW("  G");
    ns = false;
    mode = next;   // Accept alarm mode
    notifyAll();
    next = prev;   // Assume we'll resume old mode
    while (mode == 'a' && next != 'a') { // for each 'a' command
        await(90000);
        if (next == 'a') {  // One more alarm; continue
            mode = 'a';
            notifyAll();
            next = prev;
        } else {
            mode = prev;    // next might be updated now
        }
    }
}

private void time() {
    NS(" Y ");EW(" Y ");
```

```
    await(2000);
    if (ns) {
        EW("R  "); NS("  G");
    } else {
        NS("R  "); EW("  G");
    }
    mode = next; // Accept time mode
    notifyAll();
    while (next == mode) {
        await(45000, 't');
        if (next == 't') {   // go on, just toggle
            if (ns) {        // is green N-S
                NS(" Y "); await(2000);
                NS("R  "); await(2000);
                EW("RY "); await(2000);
                EW("  G");
            } else {         // is green E-W
                EW(" Y "); await(2000);
                EW("R  "); await(2000);
                NS("RY "); await(2000);
                NS("  G");
            }
            ns = !ns;        // toggle complete
        }
    }
}

private void event() {
    StreetInput.Value mycnt = new StreetInput.Value();
    mycnt.N = count.N; mycnt.S = count.S;
    mycnt.E = count.E; mycnt.W = count.W;
    if (mode != 't') {
        NS(" Y "); EW(" Y ");
        await(2000);
    }
    if (ns) {
        EW("R  "); NS("  G");
    } else {
        NS("R  "); EW("  G");
    }
    mode = next; // Accept event mode
    notifyAll();
    while (next == mode) {
        await(120000, 'e');
        if (next == 'e') { // still event mode
            int nNS = count.N - mycnt.N + count.S - mycnt.S;
            int nEW = count.E - mycnt.E + count.W - mycnt.W;
            if (nNS > 0 && nEW > 0) {
                ns = nNS > nEW;
                if (ns) { // Extra: Serve NS first; most cars there
                    NS("RY "); await(2000);
                    NS(" Y "); await(2000);
                    NS("  G"); await(2000);
                    mycnt.N = count.N; mycnt.S = count.S;
                    NS(" Y "); await(2000);
                    NS("R  "); // No await but seq/order important
                    EW("RY "); await(2000);
                    EW("  G"); await(6000);
                    mycnt.E = count.E; mycnt.W = count.W;
                    EW(" Y "); await(2000);
                    EW("R  "); await(2000);
                } else {
```

```
                        // Exactly the same but starting with EW...
                    }
                } else if (nNS > 0 && nEW == 0) { // only cars NS
                    ns = true;
                    NS("RY "); await(2000);
                    NS(" Y "); await(2000);
                    NS("  G"); await(2000);
                    mycnt.N = count.N; mycnt.S = count.S;
                    NS(" Y "); await(2000);
                    NS("R  ");
                }
                if (nNS == 0 && nEW > 0) {
                   // Exactly the same but serving EW...
                } else { // timeout, serve those plastic vehicles.
                    EW("RY "); await(2000);
                    EW("  G"); await(6000);
                    mycnt.E = count.E; mycnt.W = count.W;
                    EW(" Y "); await(2000);
                    EW("R  "); await(2000);
                    NS("RY "); await(2000);
                    mycnt.N = count.N; mycnt.S = count.S;
                    NS(" Y "); await(2000);
                    NS("  G"); await(2000);
                    NS(" Y "); await(2000);
                    NS("R  ");
                }
            }
        }
    }
}
```

Note that serving the direction with most cars first was not required, and neither was the event mode/method. To pass the exam it was sufficient to implement one of the modes.

7. Asynchronous Traffic-Light Control

The following is not a full solution, but a description of how to make it.

a)

- Move the monitor code inside the service method, and thereby make the public methods private (and remove synchronized).
- Add a putEvent method and a mailbox to your service thread, for instance by changing the thread type to JThread or similar.
- Add a timer thread that receives timer events and returns it to it's source after a time that is specified in that event.
- Update the StreetHandler and the CommandHandler threads such that they call service.putEvent(ev) instead of calling the monitor methods.

b)

With the proposed solution of problem 6, the changes concerning the application state is minor, as just described here in part a.