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

2008–12–16, 8.00–13.00

---

1. a) We call the method Richard used *busy-wait*. The disadvantage is that it consumes a lot of CPU time, potentially all CPU time, which leads to bad performance and possibly starvation for other threads.

   b) No, it does not guarantee mutual exclusion. Assume two threads are attempting to execute the given code simultaneously. Then the following order of execution allows both threads to enter the critical region:

   1) Thread 1 executes the `while` statement and discovers that the resource is free (`mutex=0`). It exits from the loop.
   2) A context switch occurs.
   3) Thread 2 also enters the `while` statement and since `mutex` is still 0 it also exits the loop.
   4) Thread 2 sets `mutex` to 1 and continues into the critical region.
   5) A context switch occurs.
   6) Thread 1 continues where it was interrupted, also sets `mutex` to 1, and continues into the critical region.

2. The argument `x` and any internal data must be stored in variables local to the method, i.e. on the stack belonging to the thread. It results in thread safety since any other simultaneous call to the method stores its data in separate memory locations.

3. a) A possible reason for the error not being reproducible is that it is related to timing and how the threads happen to be scheduled on different machines.

   b) We usually say we have a *race condition* (Swedish: emphkapplöpning).

   c) Very little (none) reliability can be expected from such an application. We need to change the development process in order to handle the problem. Developers might need training and code review should probably be used to catch the errors.

4. `CountingSem` is a general semaphore class which can be used both for mutual exclusion and signaling. However, by introducing a special semaphore class for mutual exclusion we get several advantages. For example, the semaphore can check that it is used correctly (`give()` is not called without a previous `take()` from the same thread). The system can also differentiate between signaling and mutual exclusion which is necessary when using priority inheritance protocols.

5. a) First we identify possible scheduling candidates from the course: static scheduling, RMS, DMS, EDF. Static scheduling and EDF can be ruled out since they are not based on dynamic scheduling and fixed priorities. Thus, we only have to investigate RMS (rate monotonic scheduling) and DMS (deadline monotonic scheduling). The easiest way to do this is to simply draw a scheduling diagram for the two cases showing how the threads behave at the critical instant, i.e. when all threads are released simultaneously. The diagrams are trivial for both RMS and DMS so we leave them out here.

For RMS we get: $R_A = 6ms$, $R_B = 3ms$, and $R_C = 2ms$. We see that thread A misses its deadline and the system is thus not schedulable using RMS.

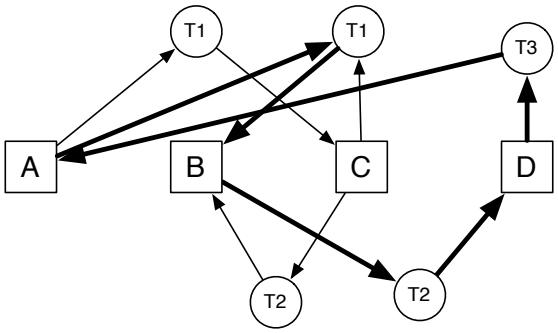
For DMS we get: $R_A = 3ms$, $R_B = 6ms$, and $R_C = 5ms$. All threads meet their deadlines so the system is thus schedulable using DMS.

Answer: DMS – Deadline Monotonic Scheduling

b) The worst-case response times follows from the discussion above.
Answer: $R_A = 3ms$, $R_B = 6ms$, and $R_C = 5ms$

6. a) Here we must start by realizing that entering a block which is `synchronized` corresponds to calling `take()` on a mutex semaphore. A hold-wait situation thus occurs when we call a `synchronized` method from within another `synchronized` method. We should also be aware that not all methods are `synchronized`, so calling these does not mean that we enter a `synchronized` block.

The graph will look like this:



b) Yes, the system may deadlock since there are a circular dependency in the graph (marked by thicker arrows) involving at most one instance of each thread.

7. **Thread A:** Thread A (highest priority thread) can potentially be blocked twice as it enters monitors M1 and M2. If thread C has locked monitor M2 and thread B has locked monitor M1 the total blocking time will be 0.5+0.1=0.6 ms. Another possibility is that C has locked monitor M1, but in that case monitor M2 must be free. So the total blocking time for this latter case must be 0.2 ms. The maximum blocking time thus becomes max(0.6,0.2) = 0.6 ms.

Thread B: Thread B has medium priority and we want to find the maximum time thread C can block it. We have two cases to consider. Case one is that thread C has locked monitor M1 in which case the maximum blocking time will be 0.2 ms. Case two is that thread C has locked monitor M2. Thread B might in such a case experience push-through blocking when C inherits the priority of A to finish executing method `z()`. This might take 0.5 ms. The maximum blocking time will thus be max(0.2,0.5) = 0.5 ms.

Thread C: There are no lower priority threads that can block thread C and the blocking time will thus be 0 ms.
Answer: $B_A = 0.6ms$, $B_B = 0.5ms$, and $B_C = 0ms$

8. Automatic Wipers

a) As the suggested monitor more or less postulates, we need (apart from the monitor and the main program) three threads that take care of the concurrency. Specifically we need the following threads:

- `Handler` is a thread that handles the user input by reading via `WiperIO.awaitHandle` and putting those event to the monitor via `Monitor.putUserEvent`. Since the former is blocking we need a separate thread, and we are free to block (`wait`) in `putUserEvent` if that would simplify/improve the software.
- `Sampler` is a thread that reads the sensor (never blocking) and puts those values into the monitor via `Monitor.putSample`, which is specified to block at certain conditions but we need to make sure it always blocks (at least for the time until next sample), otherwise there will be a busy loop consuming all CPU power. The waiting for next sample can be done by the thread sleeping outside the monitor call, but it is better to do `wait(time)` in the monitor since the combination with other blocking then gets clearer. Technically a system could work even if sleep is called within the monitor, but that is very bad practice since the monitor then is locked during the sleeping time.
- `Commander` is a thread that takes care of the control actions that are obtained from `Monitor.getControl`. In principle we could be without this thread like we had no alarm thread necessary in lab 1, but note 3 with the 800ms motivates an extra thread as indicated by the existence of the getControl method. If we decide to have all `WiperIO` calls outside of the monitor (inside each thread respectively) we need a way to tell the thread to do the 800ms puls (somewhat dirty by the changed speed sign in code below). Alternatively that could be managed by `wait(800)` in the monitor but then the logic gets more complex there.

The main thread would just start the rest. With the design information given, and preferably a simple figure, implementation of the threads are not required. Alternatively the content of the `while run` loop can be given, as part of the optional thread implementations below.

As we learned in lab 2 it is advisable to, if possible, have fewer and bigger monitor operations that each take care of a complete action/transaction. Then each operation is a sequence that can only be interrupted when wait is called, and concurrency problems (and readability of the code) get much easier to handle. There is normally no efficiency problem due to having bigger monitor operations. When other threads are blocked a longer time on entering the synchronized scope, those threads do not take CPU time and the thread in the monitor gets the work done (more efficiently than if the monitor need to be entered multiple times).

b) For completeness the calling threads are first given, but the task simply is to (during some 2-2.5 hours) implement the three suggested monitor methods as shown below.

```
public class Handler extends Thread {
    WiperIO io;
    Monitor mon;
    Handler(WiperIO io, Monitor mon) {this.io=io; this.mon=mon;}
    public void run() {
        while (!interrupted()) {
            mon.putUserEvent(io.awaitHandle());
        }
    }
}

public class Sampler extends Thread {
    WiperIO io;
    Monitor mon;
    Sampler(WiperIO io, Monitor mon) {this.io=io; this.mon=mon;}
    public void run() {
        while (!interrupted()) {
            mon.putSample(io.sampleSensor());
            try {
                sleep(100);
```

```
            } catch (InterruptedException exc) {return;}
        }
    }
}

public class Commander extends Thread {
    WiperIO io;
    Monitor mon;
    Commander(WiperIO io, Monitor mon) {this.io=io; this.mon=mon;}
    public void run() {
        while (!interrupted()) {
            int spd = mon.getControl();
            if (spd<0) { // INT or AUT: pulse output during 0.8s
                io.commandWiper(-spd);
                try {
                    Thread.sleep(800);
                } catch (InterruptedException exc) {
                    io.commandWiper(WiperIO.WIPE_STOP);
                }
            } else io.commandWiper(spd);
        }
    }
}
```

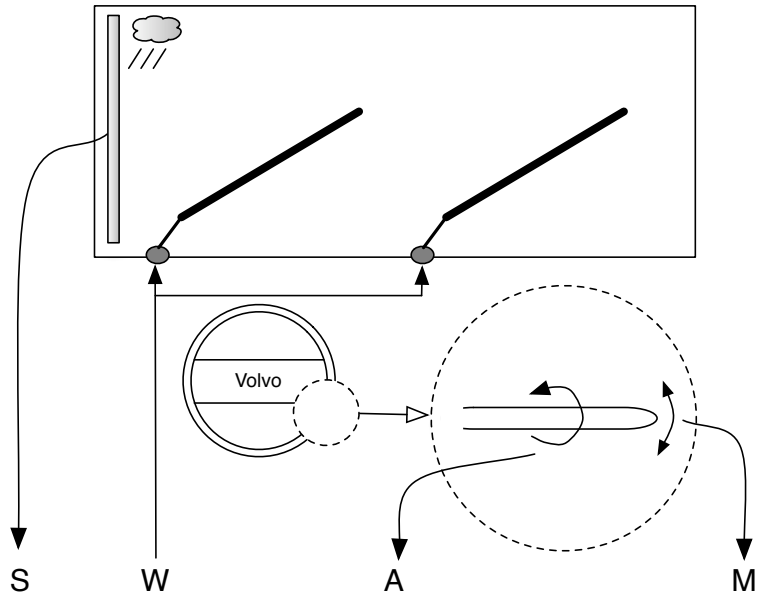
The actual implementation that to some extent is required to pass the exam:

```
public class Monitor {

    // Attributes to be defined by the programmer...
    float rain;
    int mode;
    float tuning;
    long interval;
    long release;
    float level;

    int oldMode = mode;
    long oldInterval = interval;
    float oldLevel = level;

    /**
     * .. according to given text ..
     */
    synchronized void putSample(float sens) {
        try {
            while (mode!=WiperIO.MODE_AUT) {
                wait();
                rain=sens=(float)1.0;
            }
            rain = (float)0.99*rain + (float)0.01*sens;
            if (rain>level) {
                notifyAll();
                while (rain!=(float)0.0) {
                    wait();
                }
            }
        } catch (InterruptedException exc) {rain=(float)0.0;}
    }

    /**
     * ... accoding to given text, but added:
```

```
     * In mode INT and AUT the return value is negated, meaning that
     * motion is only to be started (0.8s pulse) with that speed.
     */
    synchronized int getControl() {
        int ans = WiperIO.WIPE_STOP;
        try {
            while (mode==oldMode) {
                if (mode==WiperIO.MODE_INT &&
                    mode==WiperIO.MODE_INT) break;
                if (interval!=oldInterval || level!=oldLevel)
                    break;
                wait();
            }
            logger.info("new control");
            switch (mode) {
            case WiperIO.MODE_OFF:
                ans = WiperIO.WIPE_STOP; // once again
                break;
            case WiperIO.MODE_INT:
                if (oldMode!=WiperIO.MODE_INT) {
                    release = System.currentTimeMillis();
                }
                release += interval;
                long dt = release-System.currentTimeMillis();
                while (dt>0) {
                    wait(dt);
                    dt=release-System.currentTimeMillis();
                }
                ans = WiperIO.WIPE_SLOW;
                ans = -ans; // Tell caller to pulse output.
                break;
            case WiperIO.MODE_AUT:
                while (mode==WiperIO.MODE_AUT && rain<level) {
                    wait();
                }
                if (mode==WiperIO.MODE_AUT) {
                    ans = (rain > (float)0.9) ?
                        WiperIO.WIPE_FAST :
                        WiperIO.WIPE_SLOW;
                    rain = (float)0.0;
                } else if (mode==WiperIO.MODE_OFF) {
                    ans = WiperIO.WIPE_STOP;
                } else if (mode==WiperIO.MODE_SLOW) {
                    ans = WiperIO.WIPE_SLOW;
                } else if (mode==WiperIO.MODE_FAST) {
                    ans = WiperIO.WIPE_FAST;
                }
                ans = -ans; // Tell caller to pulse output.
                break;
            case WiperIO.MODE_SLOW:
                ans = WiperIO.WIPE_SLOW;
                break;
            case WiperIO.MODE_FAST:
                ans = WiperIO.WIPE_FAST;
                break;
            }
        } catch (InterruptedException exc) {ans = WiperIO.WIPE_STOP;}
        oldMode = mode;
        oldInterval = interval;
        oldLevel = level;
        return ans;
    }
```

```
    /**
     * .. according to given text ..
     */
    synchronized void putUserEvent(WiperIO.UserEvent command) {
        mode = command.mode;
        tuning = command.adjust;
        switch (mode) {
        case WiperIO.MODE_AUT:
            level = tuning;
            break;
        case WiperIO.MODE_INT: // 2 to 12 s depending on user knob
            interval = 2000 + Math.round((float)8000*tuning);
            break;
        }
        notifyAll();
    }
}
```

Endnote: As an example from an actual car, a picture of the handle for a Volvo V50 is shown below. The scale in the middle, showing an increased value upwards, is fixed in the handle. To the right of that scale there is the knob for adjusting the value A (interval time or rain sensitivity, depending on mode). To the left is the push button for activating `MODE_AUT` (when the handle as such is in zero position). The other values of mode M is obtained by turning/pushing the handle down to `MODE_INT`, `MODE_SLOW` and `MODE_FAST`.



9. **Android**

The Android focus on mobile interactive graphical applications apparently resulted in

- an event-based approach where an event is processed in non-blocking pieces of application code, or there has to be an explicit service that does background processing. We may refer to this as a **Code** view of the platform.
- explicit support for management of the rather small display area, where events from user input goes to the appropriate activities and the applications have a simple way of influencing the displayed views on different screens belonging to different applications. We may refer to this as a **User** view.
- explicit support for managing embedded resources (time and momory), both in overload situations but also to cope with applications that behave badly (looping in event processing, hanging on networking, and the like) due to design or programming errors. This is the **Exec**ution view.

The nice thing with Android appears to be that these three items are all nicely engineered together in a practical way. Other attractive choices are the use of Java as the programming language, and the use of Linux as the default runtime platform.

There are several ways to present a solution to this problem, including UML diagrams showing the relations and interactions between the involved classes and objects. However, only an overall design with emphasis on threads, activities and processes is required, so one possible solution is to simply present a figure with your own notation like the one below coving one example application.

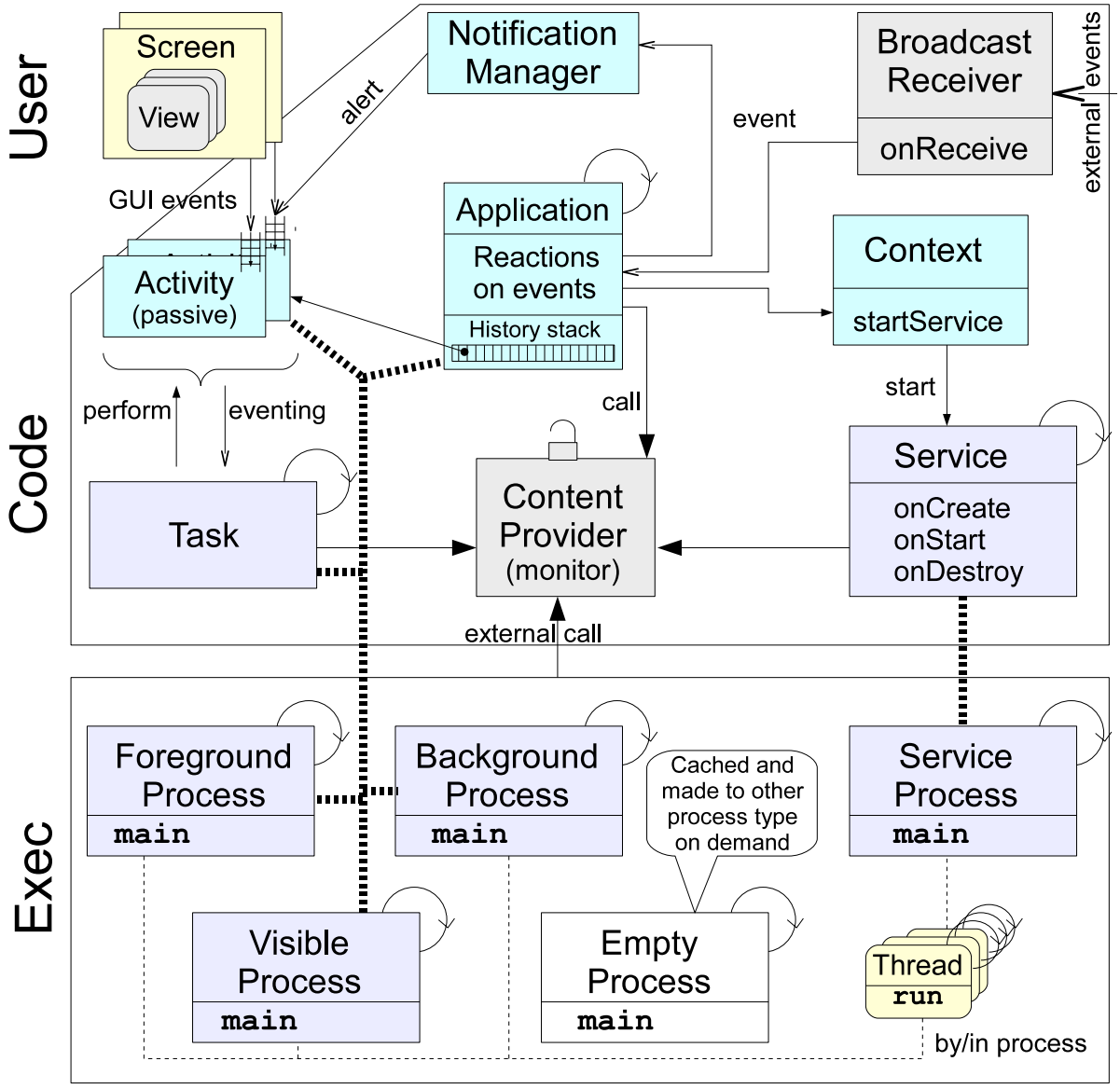
To arrive to a draft solution in a short time (max 1 hour), just read the description while marking/underlining the phrases that have to do with the concurrent design, then review the underlines parts while drawing their relations (such as event flow or calls), and finally put the figures together in one (or two) showing the overall design. Then add text/notes explaining the notions and the things that are not shown in the figure on the next page.

Here the events flows are marked with open arrows, the calls with solid arrows, the mutual exclusive calls (to the monitor) with bigger solid arrows, and relations between processes and the code they execute is denoted with thick dotted lines. The thin dashed lines denote that additional threads (apart from the main thread) run in processes.

A particular aspect of processes is that they have a well-defined resource allocation with respect to the runtime system (OS), which means that processes can be killed when needed. That opposed to threads that are collaborating activities that can be asked to terminate but need to do that by ending their run (or main for the main thread) method. For the purpose of this exam problem, we can model also the processes by threads (expected alternative that works under certain assumptions), thread groups (if you know their use in e.g. applets), or by processes (if java.lang.Process is known to you).

The activities and the applications do event processing, and for the Activity it was explicit that it is a passive type of object. The Application is more unclear. In the figure the Application is active but possible they could be run via a corresponding task. A service is run by a Service Process, the Empty Process is there for performance reasons, and the other three types of processes run the tasks (and the application).

Rather than looking at details, the main issue is that the interpretation of Android in terms of an overall design should make sense from the concurrency point of view according to the course content.


User
Code
Exec
Screen
View
alert
Notification Manager
Broadcast Receiver
onReceive
external events
event
GUI events
Application
Reactions on events
History stack
Context
startService
Activity (passive)
perform
eventing
call
start
Service
onCreate
onStart
onDestroy
Task
Content Provider (monitor)
external call
Foreground Process
main
Background Process
main
Cached and made to other process type on demand
Service Process
main
Visible Process
main
Empty Process
main
Thread
run
by/in process