Exam
Concurrent and Real-Time Programming

2008–12–16, 8.00–13.00

You are allowed to use the Java quick reference and a calculator. To pass the exam, grade 3, you have to solve most of the first 7 problems (which are more of a theoretical type), and you have to develop an acceptable solution to problem 8 (programming). At least a partial solution to problem 9 is required for grade 5.

---

1. Realtime-Richard has written the following lines of code in Java in order to achieve mutual exclusion over the call to `doSomething()`.

```
while (mutex==1) { /* Do nothing */ }
mutex = 1;
doSomething();
mutex = 0;
```

   The `mutex` variable is declared "`volatile int mutex = 0;`" (declaring the variable `volatile` eliminates all possible problems with compiler optimizations etc).

   a) What do we usually call the method Richard use to wait for the shared resource to become available? What disadvantage does this method have?

   (1p)

   b) Does Richard's solution guarantee mutual exclusion? Motivate your answer!

   (1p)

2. According to Wikipedia (which agrees with our course) a computer program or routine is described as ***reentrant*** if it can be safely executed concurrently; that is, the routine can be re-entered while it is already running. To be reentrant, a function:

   - Must hold no static (global) non-constant data.
   - Must not return the address to static (global) non-constant data.
   - Must work only on the data provided to it by the caller.
   - Must not rely on locks to singleton resources.
   - Must not call non-reentrant functions.

   This implies that a reentrant function is thread safe. That is, a reentrant function like `Math.sin(double x)` can be used by multiple threads at the same time. How are then the argument `x` and any internal data (as the summation variable for a series expansion of the sine function) allocated, and why does that result in the thread safety?

   (1p)

3. Many modern software applications, such as Microsoft/Star/Open/Symphony Office programs, include concurrent behavior. For instance, the following is managed in parallel while editing a WYSIWYG (what you see is what you get) text document including layout: text input, spell/grammar checking, online help, and layout of pages/paragraphs. The latter may take longer time than writing several words, and hence the activities have to be handled by several threads with the document as a shared resource.

   Together with the software there are some means of reporting bugs/errors into some bug-tracking system (e.g. based on Bugzilla). An error is then classified into categories such as *Confirmed* (yes this is a bug that should be fixed), *Assigned* (somone is working on it) and *Fixed* (done, from version x.y).

   Typically there is also a classification *Not reproducible*, which means that the programmer about to investigate the problem cannot reproduce the error on his/her computer.

   a) What reason, related to concurrent programming techniques, could there be that an error is not reproducible?

   (1p)

   b) What do we call this type of problem?

   (1p)

   c) What reliability can be expected from a multithreaded software with developers ignoring all non-reproducible errors? How would you improve quality?

   (1p)

4. In the package `se.lth.cs.realtime.semaphore` we have an `interface Semaphore` declaring the methods `take` and `give`, which then are implemented by classes such as `CountingSem` and `MutexSem`. Mention two reasons for having those two separate classes (rather than only having just one class for both types of semaphores).

   (2p)

5. Consider a real-time system consisting of three independent, periodically executing, threads with characteristica according to the table below (C = worst-case execution time, D = deadline, T = period).

Tråd	C (ms)	D (ms)	T (ms)
A	3	4	10
B	1	8	8
C	2	6	6

   a) Suggest a scheduling principle, known from the course, based on *dynamic scheduling and fixed priorities* which can guarantee that the system described above is always schedulable (i.e., every thread will always meet its deadline).

   (1p)

   b) What will the worst-case response time be for each of the three threads above if the we use the scheduling principle you suggested above?

   (2p)

6. A Java program consists of three concurrently executing threads, T1, T2, and T3. They communicate with each other using four shared objects, A, B, C, and D. The shared objects contains methods declared `synchronized` as well as other, non-synchronized, methods (such as `s()` below). Relevant parts of the program and the design of the monitors are shown below (subset of the total program only). Each thread and shared object is assumed to have references to the (other) shared objects A, B, C, and D named a, b, c, and d respectively.

```
class T1
    extends Thread {
    void run() {
        b.z();
        a.x();
    }
}
```

```
class T2
    extends Thread {
    void run() {
        b.r();
        c.y();
        b.w();
    }
}
```

```
class T3
    extends Thread {
    void run() {
        d.u();
    }
}
```

```
class A {
    synchronized void x() {
        c.y();
    }

    void s() {  // NOT synchronized
        ...
    }

    synchronized void t() {
        ...
    }
}
```

```
class B {
    synchronized void z() {
        ...
    }

    synchronized void r() {
        a.s();
    }

    synchronized void w() {
        d.v();
    }
}
```

```
class C {
    synchronized void y() {
        b.z();
    }
}
```

```
class D {
    synchronized void u() {
        a.t();
    }

    synchronized void v() {
        ...
    }
}
```

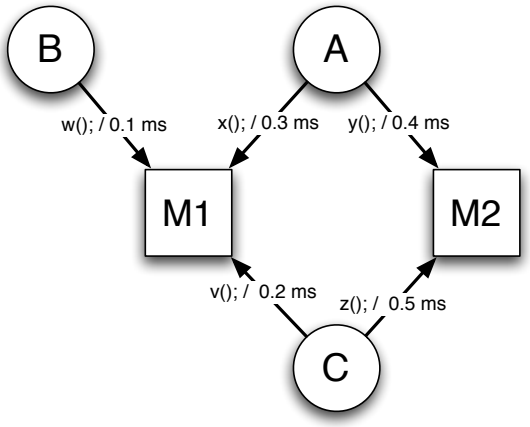
a) Draw a resource allocation graph for the system above.

(3p)

b) Can the system experience deadlock? Motivate your answer.

(1p)

7. The three threads with highest priority in a real-time system (A, B, and C) communicates with each other using two monitors (M1 and M2) according to the figure below:



The monitor operations `v`, `w`, `x`, `y`, and `z` are called by thread A, B, and C as illustrated in the figure once (and only once) each time respective thread is invoked. The calls are not nested, i.e., one monitor operation is not called from within another monitor operation. For each monitor operation, the maximum required execution time is shown in the figure.

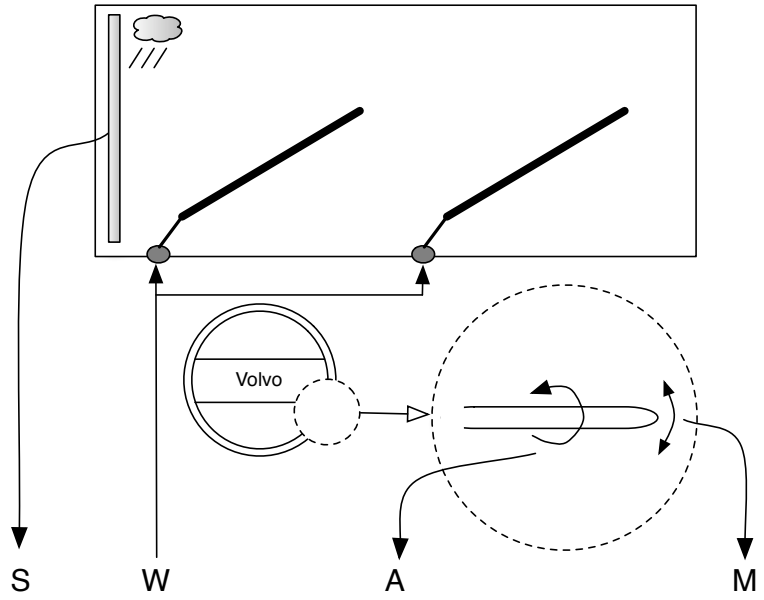
Thread A has the highest priority and C has the lowest priority. The threads do not communicate or interact with lower priority threads other than as illustrated in the figure.

What will the maximum blocking time be for each of the three threads (A, B, and C), i.e., what is the maximum time each of the three threads can be blocked by lower priority threads during one invocation assuming basic priority inheritance is employed?

(3p)

8. **Automatic Wipers**

Many modern cars are equipped with sensing of rain drops on the windscreen, which is then used to automatically control the wipers as needed. Since the sensing is not perfect in terms of how the rain effects the sight, and user preferences differ, there is an adjustable value (`float A`) that is set by the user by twisting the wiper control handle. The same handle is turned to set the mode (`int M`) of wiping, as shown in the figure. The modes are: OFF, AUTomatic, INTerval, SLOW (continuous wiping at low speed), and FAST (continuous wiping at high speed). In automatic mode the sensor (`float S`) is used to determine the wiping. In mode INT the adjusted value instead determines the time between wipes, which are done with speed SLOW. The motor is commanded by an output W (`int W`) that controls the wiper speed.



The hardware in terms of IO methods is available according to class `WiperIO`. Note that `awaitHandle` blocks until the user has changed anything, and `sampleSensor` samples without blocking but sampling is only needed while in automatic mode. The `commandWiper` also works without blocking, but there are some things to know:

1. When a non-zero (slow or fast) speed has been set, the wipers move (due to the mechanics) back and forth without more commands needed.
2. When stop has been commanded during a motion, the wipers automatically continues to the resting position.
3. When a non-zero speed is commanded after a stop, for the purpose of interval or automatic wiping (that is, when each turn is to be explicitly commanded), that speed has to be set during 0.8s (to move the wipers out of the stopping position).

```
class WiperIO {

    /** Values for mode M input: */
    final static int MODE_OFF = 0;
    final static int MODE_INT = 1;
    final static int MODE_AUT = 2;
    final static int MODE_SLOW = 3;
    final static int MODE_FAST = 4;
```

```
    /** Values for wiper motor W output: */
    final static int WIPE_STOP = 0;
    final static int WIPE_SLOW = 1;
    final static int WIPE_FAST = 2;

    /** Return type for reading of handle. */
    static class UserEvent {
        UserEvent(int mode, float adjust) {
            this.mode = mode; this.adjust = adjust;
        }
        int mode;       // M
        float adjust;   // A
    }

    /** Block caller until there is a change in mode or adjustment */
    UserEvent awaitHandle() {...}

    /**
     * Sample the rain-sensor value.
     * @return the sensed value between 0.0 (dry) and 1.0 (max).
     */
    float sampleSensor() {...}  // S

    /**
     * Set the speed of the wiper motors.
     * @param wipe the value according to WIPE_* constants.
     */
    void commandWiper(int wipe) {...}  // W

    /** Obtain the interface instance to use. */
    static WiperIO getInstance() {...}
}
```

Using this knowledge about the application and the control interface, your task is to design and implement the software that controls the wipers according to the above specifications.

Overall design

1. Based on experiences from an earlier project where data buffering resulted in latencies/delays that were hard to predict/test, there is a management decision that buffering is only permitted for smoothing computational load (running heavy computation in background, with order queue input) or for network interfaces.
2. During for instance the INT mode, the motor interface is given a SLOW speed during 0.8 s to get started, and such a pulse is desirable to generate sequentially within one thread. Concurrently during that time, user inputs should be taken care of (registered but not buffered).
3. Sensing is done periodically with period 100 ms during automatic mode, but during other modes the sampling of the sensor should not be carried out.

Explain the design of your software system in terms of an annotated figure. Requirement 1 leads to a monitor-based design. Why is it typically a good idea to have a few larger monitor methods instead of several small ones, and why is that usually *not* more costly in terms of processing time or delays?

The following code is a suggested monitor interface, where the comments are part of the system specifications.

```
public class Monitor {

    // Attributes to be defined by the programmer...
    int mode;
    ...
```

```
    /**
     * Provide sensor value for automatic mode (MODE_AUT). If that mode
     * is not active, block until that is the case and then continue as
     * if maximum rain is the case. In automatic mode, filter sensed
     * value (99% of old value and 1% of new), and if the filtered value
     * is above the level A (adjustment via user knob): notify the control
     * and wait (by blocking) for wiping to be ordered. Then continue as
     * if dry and let sensing increase rain level.
     *
     * @param sens the new sensor value between 0.0 (dry) and 1.0 (wet).
     */
    synchronized void putSample(float sens) {...}

    /**
     * Determine what wiper speed to use now, depending on modes:
     * MODE_OFF: STOP
     * MODE_SLOW: SLOW
     * MODE_FAST: FAST
     * MODE_INT: STOP then SLOW
     * MODE_AUT: STOP until rain indicated, then SLOW (or fast if S>0.9)
     *
     * For interval or automatic, call blocks until conditions met.
     *
     * @return the desired wiper speed according to WiperIO.WIPE_*
     */
    synchronized int getControl() {...}

    /**
     * Provide any changed user input, due to changed mode or adjusted
     * tuning A of interval or rain sensitivity. For interval wiping,
     * the period varies from 2s to 10s depending on user knob A.
     *
     * @param command is the new input values.
     */
    synchronized void putUserEvent(WiperIO.UserEvent command) {...}
}
```

(3p)

Implementation and detailed design

With relevant motivations, you may improve these monitor methods in terms of arguments and features. Due to the safety traditions in the car industry. threads are only created when the program starts. Implement the monitor and the run/perform methods of the involved threads.

(8p)

9. Android

To create an open and free mobile software platform, Google recently introduced Android (`http://code.google.com/android`). The following description is an enhanced extract from that web site, where you also can read that Linux is the main target platform, and the system is Java-based (written, compilable and runnable as a Java, but cross-compiled code is not running on a JVM). Thus the approach is in line with that of the Lund appraoch, but since the application area is mobile interactive applications rather than real-time control applications, the structure and available packages are quite different.

Your assignment here is to describe how a platform similar to Android could be designed using the Java-classes and concurrent functionality you know from the course. Your answer should primarily be given in terms of:

- One (or a few) figure(s) showing the involved classes/objects, and how they relate in terms of synchronization and signaling.
- Clarification (in the figure and/or in text) how the (semi-concurrent) activities in the Google design can be accomplished by using ordinary Java threads.
- It should also be clear how processes and threads (in the Android sense) are related/accomplished by means of your normal Java threads, and how the message-based communication in terms of events is managed. Possibly some short pieces of code to explain certain solutions.

You may assume a general application or a specific example application, whatever best describes the design of of your version of a new mobile software platform. As a specification of the platform the following descriptions from Google applies:

Introduction

There are four building blocks to an Android application: Activity, Broadcast Receiver, Service, and Content Provider. Not every application needs to have all four, but your application will be written with some combination of these. Referring to these building blocks as described in this section, there is then an application model involving processes and threads as described in the next section.

Activity

Activities are the most common of the four Android building blocks. An activity can be a single screen in your application. Each activity is implemented as a single class that extends the Activity base class. Your class will display a user interface composed of graphical views and *respond to events*.

Applications can also consist of multiple screens. For example, a text messaging application might have one screen that shows a list of contacts to send messages to, a second screen to write the message to the chosen contact, and other screens to review old messages or change settings. Each of these screens would be implemented as an activity. Moving to a new screen is accomplished by a starting a new activity. In some cases an activity may return a value to the previous activity – for example an activity that lets the user pick a photo would return the chosen photo to the caller.

When a new screen opens, the previous screen is paused and put onto a history stack. The user can navigate backward through previously opened screens in the history. Screens can also choose to be removed from the history stack when it would be inappropriate for them to remain. Android retains *history stacks* for each application launched from the home screen.

Service

A Service is code that is long-lived and runs without a UI. A good example of this is a media player playing songs from a play list. In a media player application, there would probably be one or more activities that allow the user to choose songs and start playing them. However, the music playback itself should not be handled by an activity because the user will expect the music to keep playing even after navigating to a new screen. In this case, the media player *activity could start a service* using `Context.startService()` to run in the background to keep the music going. The system will then keep the music playback service running until it has finished.

Broadcast Receiver

You can use a `BroadcastReceiver` when you want code in your application to execute in reaction to an external event, for example, when the phone rings, or when the data network is available, or when it's midnight. BroadcastReceivers do not display a UI, although they may use the NotificationManager to alert the user if something interesting has happened.

Content Providers

If you want to make your data public, you can create (or call) a content provider. This is an object that can store and retrieve data accessible by all applications. This is the only way to share data across packages; there is no common storage area that all packages can share. Android ships with a number of content providers for common data types (audio, video, images, personal contact information, and so on). You can see some of Android's native content providers in the provider package. How a content provider actually stores its data under the covers is up to the implementation of the content provider, but all content providers must implement a common convention to query for data, and a common convention to return results. However, a content provider can implement custom helper functions to make data storage/retrieval simpler for the specific data that it exposes.

Android Application Model: Applications, Tasks, Processes, and Threads

In most operating systems, there is a strong 1-to-1 correlation between the executable image (such as the `.exe` on Windows) that an application lives in, the process it runs in, and the icon and application the user interacts with. In Android these associations are much more fluid. Because of the flexible nature of Android applications, there is some basic terminology involving Tasks, Processes and Threads that needs to be understood when implementing the various pieces of an application.

Tasks

A task is generally what the user perceives as an "application" that can be launched: usually a task has an icon in the home screen through which it is accessed, and it is available as a top-level item that can be brought to the foreground in front of other tasks. A key point here is: when the user sees as an "application", what they are actually dealing with is a task. If you just create program with a number of activities, one of which is a top-level entry point, then there will indeed be one task created, and any activities you start from there will also run as part of that task.

A task, then, from the user's perspective your application; and from the application developer's perspective it is one or more activities the user has traversed through in that task and not yet closed.

Processes

In Android, processes are entirely an implementation detail of applications and not something the user is normally aware of. A process is a low-level kernel process in which an application's code is running. Normally all of the code of one application is run in one dedicated process; however, the process tag can be used to modify where that code is run, either for the entire application or for individual activity, receiver, service, or provider, components. The main uses of processes are simply:

- Improving stability or security by putting untrusted or unstable code into another process.
- Reducing overhead by running the code of multiple packages in the same process.
- Helping the system manage resources by putting heavy-weight code in a separate process that can be killed independently of other parts of the application.

In most cases, every Android application runs in its own Linux process. This process is created for the application when some of its code needs to be run, and will remain running until it is no longer needed and the system needs to reclaim its memory for use by other applications.

To determine which processes should be killed when low on memory, Android places each process into an "importance hierarchy" based on the components running in them and the state of those components. These process types are (in order of importance):

1. A *foreground process* is one that is required for what the user is currently doing. Various application components can cause its containing process to be considered foreground in different ways. A process is considered to be in the foreground if any of the following conditions hold:
   - It is running an Activity at the top of the screen that the user is interacting with (its `onResume()` method has been called).
   - It has a BroadcastReceiver that is currently running (its `BroadcastReceiver.onReceive()` method is executing).
   - It has a Service that is currently executing code in one of its callbacks (`Service.onCreate()`, `Service.onStart()`, or `Service.onDestroy()`).

   There will only ever be a few such processes in the system, and these will only be killed as a last resort if memory is so low that not even these processes can continue to run. Generally, at this point, the device has reached a memory paging state, so this action is required in order to keep the user interface responsive.
2. A *visible process* is one holding an Activity that is visible to the user on-screen but not in the foreground (its `onPause()` method has been called). This may occur, for example, if the foreground Activity is displayed as a dialog that allows the previous Activity to be seen behind it. Such a process is considered extremely important and will not be killed unless doing so is required to keep all foreground processes running.
3. A *service process* is one holding a `Service` that has been started with the `startService()` method. Though these processes are not directly visible to the user, they are generally doing things that the user cares about (such as background mp3 playback or background network data upload or download), so the system will always keep such processes running unless there is not enough memory to retain all foreground and visible process.
4. A *background process* is one holding an `Activity` that is not currently visible to the user. These processes have no direct impact on the user experience. Provided they implement their activity life-cycle correctly, the system can kill such processes at any time to reclaim memory for one of the three previous processes types. Usually there are many of these processes running, so they are kept in a list to ensure the process that was most recently seen by the user is the last to be killed when running low on memory.
5. An *empty process* is one that doesn't hold any active application components. The only reason to keep such a process around is as a cache to improve startup time the next time a component of its application needs to run. As such, the system will often kill these processes in order to balance overall system resources between these empty cached processes and the underlying kernel caches.

When deciding how to classify a process, the system will base its decision on the most important level found among all the components currently active in the process.

Threads

Every process has one or more threads running in it. In most situations, Android avoids creating additional threads in a process, keeping an application single-threaded unless it creates its own threads. An important repercussion of this is that all calls to Activity, BroadcastReceiver, and Service instances are made only from the main thread of the process they are running in.

Note that a new thread is not created for each Activity, BroadcastReceiver, Service, or ContentProvider instance: these application components are instantiated in the desired process (all in the same process unless otherwise specified), in the main thread of that process. This means that none of these components (including services) should perform long or blocking operations (such as networking calls or computation loops) when called by the system, since this will block all other components in the process. You can use the standard library Thread class or Android's HandlerThread convenience class to perform long operations on another thread.

(7p)