Tentamen
Realtidsprogrammering

2008–12–16, 8.00–13.00

Hjälpmedel: inga utöver Java snabbreferens och miniräknare.
DATE14: En uppdelning av betyg 4 för G respektive VG kommer att ske, för övrigt se EDA040.
EDA040: För *godkänt betyg* krävs att större delen av de 7 första uppgifterna (teori) *samt uppgiften 8 (programmering) behandlas nöjaktigt*. Poäng från lösning av uppgift 9 krävs för betyget 5.
OBS: Tentamina framöver ges endast på engelska (kan självklart besvaras på svenska).

---

1. Realtids-Rikard har skrivit följande programrader i Java för att åstadkomma ömsesidig uteslutning över anropet av `doSomething()`.

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

   Variabeln `mutex` är deklarerad "`volatile int mutex = 0;`"(genom att deklarera variabeln `volatile` undviker vi alla problem skapade av kompilatoroptimeringar etc).

   a) Vad brukar man kalla metoden Rikard använde för att vänta på att den delade resursen ska bli ledig och vilken nackdel har denna metod?

   (1p)

   b) Garanterar Rikards lösning ömsesidig uteslutning? Motivera ditt svar!

   (1p)

2. Enligt Wikipedia (som håller med om vad som påstås i vår kurs) beskrivs ett datorprogram eller funktion som ***reentrant*** om det/den kan exekveras säkert av flera parallella trådar; dvs den kan anropas på nytt under tiden som den håller på att exekveras. För att en funktion ska vara rr-eentrantmåste den:

   - Inte använda sig av statisk (global) icke-konstant data.
   - Inte returnera adressen till statisk (global) icke-konstant data.
   - Enbart arbeta med data tillhandållen via anroparen.
   - Inte vara beroende av lås av resurser som bara finns i en upplaga (singleton resources).
   - Inte anropa funktioner som inte är reentrant.

   Ovanstående innebär att en funktion som är reentrant också är trådsäker, dvs en funktion som t.ex. `Math.sin(double x)` kan anropas av flera trådar parallellt. Hur måste i så fall argumentet `x` och alla eventuella interna data (som t.ex. summeringsvariabeln i en serieutveckling för beräkningen av sinus) lagras och varför påverkar valet av lagringsutrymme trådsäkerheten?

   (1p)

3. Många moderna datorapplikationer, som Microsoft/Star/Open/Symphony Office, inkluderar parallellt beteende. Följande behandlas t.ex. parallellt när att man editerar ett WYSIWYG-dokument (what you see is what you get): inmatning av text, stavnings- och grammatikkontroll, online-hjälp och layout av text/stycken. Det senare kan ta längre tid än det tar att skriva in flera nya ord och måste därför hanteras av olika trådar med dokumentet som en delad resurs.

   I många fall finns det ett system för att rapportera buggar/fel i programmet. Ett sådant system kan t.ex. vara baserat på Bugzilla, som är ett populärt system för att hålla reda på buggar och buggfixar. I ett sådant system klassificeras en bugg i olika kategorier såsom *Confirmed* (ja, det är en bugg som bör åtgärdas), *Assigned* (någon arbetar på felrapporten) och *Fixed* (åtgärdat, från version x.y).

   Typiskt finns det även en klassifikation *Not reproducible*, vilket innebär att programmeraren som undersöker rapporten inte kan återskapa felet på sin dator.

   a) Vad kan orsaken, relaterad till multitrådad programmering, vara till att en bugg inte är reproducerbar?

   (1p)

   b) Vad brukar vi använda för namn på ovanstående typ av problem?

   (1p)

   c) Vilken robusthet och pålitlighet kan förväntas från ett multitrådat program vars utvecklare ignorerar alla icke-reproducerbara fel? Hur skulle du vilja förbättra kvaliteten?

   (1p)

4. I paketet `se.lth.cs.realtime.semaphore` finns det ett `interface Semaphore` som deklarerar metoderna `take` och `give`, som sedan implementeras av klasser såsom `CountingSem` och `MutexSem`. Nämn två orsaker till att vi kan vilja ha två separata semaforklasser enligt ovan (i stället för att ha bara en klass för båda typerna av semaforer).

   (2p)

5. Betrakta ett system bestående av tre oberoende periodiskt exekverande trådar med nedanstående karaktäristika (C = värstafallsexekveringstid, D = deadline, T = period).

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

   a) Föreslå en från kursen känd schemaläggningsprincip för *dynamisk schemaläggning med fixa prioriteter* för trådarna som gör att systemet ovan är schemaläggningsbart (dvs att alla trådar garanterat klarar sina deadlines).

   (1p)

   b) Vad blir värstafallssvarstiderna för de tre olika trådarna om man tillämpar schemläggningsprincipen du föreslog ovan?

   (2p)

6. Ett javaprogram innehåller tre stycken parallellt exekverande trådar, T1, T2 och T3. De kommunicerar med varandra via fyra gemensamma objekt, A, B, C och D som bland annat innehåller metoder deklarerade `synchronized` (men även andra, ickesynkroniserade, metoder såsom metoden `s()` nedan). Relevanta delar av trådarnas och monitorernas design visas nedan (utdrag ur den totala koden som behövs för att programmet ska bli komplett). Varje tråd och delat objekt antas ha referenser till de (övriga) delade objekten A, B, C och D kallade a, b, c respektive d.

```
class T1                class T2                class T3
    extends Thread {        extends Thread {        extends Thread {
    void run() {            void run() {            void run() {
        b.z();                  b.r();                  d.u();
        a.x();                  c.y();              }
    }                           b.w();          }
}                           }
                        }
```

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

    void s() {   // NOT synchronized    synchronized void r() {
        ...                                 a.s();
    }                                   }

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

```
class C {                           class D {
    synchronized void y() {             synchronized void u() {
        b.z();                              a.t();
    }                                   }
}
                                        synchronized void v() {
                                            ...
                                        }
                                    }
```

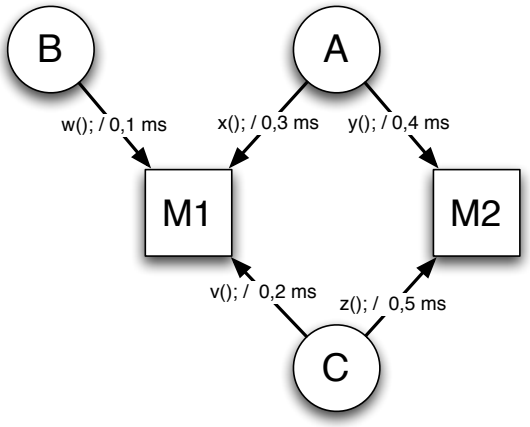
a) Rita en resursallokeringsgraf för systemet ovan.

(3p)

b) Finns det risk för dödläge i systemet? Motivera ditt svar.

(1p)

7. De tre högst prioriterade trådarna (A, B och C) i ett realtidssystem kommunicerar med varandra via två monitorer (M1 och M2) enligt nedanstående figur:



Monitoroperationerna `v`, `w`, `x`, `y` och `z` anropas av trådarna A, B och C enligt pilarna i figuren en (och endast en gång) varje period för respektive tråd. Anropen är ej nästlade, dvs man anropar inte en monitoroperation inuti en annan monitoroperation. För varje operation anges i figuren den maximala tid det kan ta att exekvera operationen i fråga.

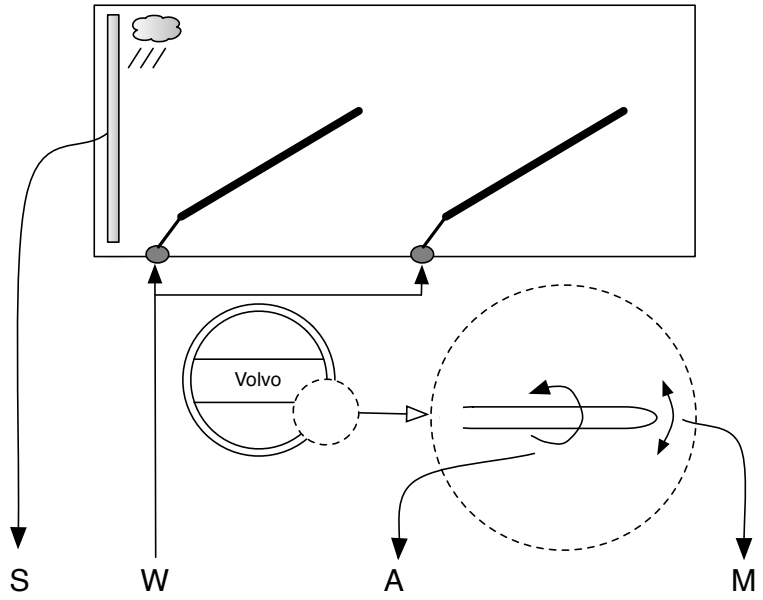
Tråden A har högst prioritet och C har lägst prioritet. Trådarna kommunicerar inte med andra (lägre prioriterade) trådar annat än vad som visas i figuren.

Vad blir den maximala tiden som var och en av trådarna A, B och C kan blockeras av lägre prioriterade trådar under en period under förutsättning att dynamisk prioritetsärvning (basic priority inheritance) används?

(3p)

8. Automatiska vindrutetorkare

Många moderna bilar är utrustade med sensorer i vindrutan som känner av om regndroppar träffar vindrutan, vilket används för att automatiskt aktivera vindrutetorkarna vid behov. Eftersom den automatiska avkänningen inte är perfekt och olika förare har olika önskemål om hur ofta vindrutan ska torkas av finns det ett justerbart värde (`float A` – adjust) som kan ställas in av föraren genom att vrida på kontrollspaken för vindrutetorkarna. Samma kontrollspak kan även föras upp och ner för att aktivera olika funktionslägen (`int M` – mode), såsom visas i figuren. Lägena är: OFF, AUTomatic, INTerval, SLOW (kontinuerlig rörelse med låg hastighet) och FAST (kontinuerlig rörelse med hög hastighet). I automatiskt läge används regnsensorn (`float S` – sensor) för att styra torkningen. I läget INT används det på spaken inställda värdet i stället för att bestämma tiden mellan två avtorkningar, vilka utförs med samma hastighet som i läge SLOW. Vindrutetorkarnas motorer styrs genom en utsignal W (`int W` – wipe) som kontrollerar motorernas hastighet.



Hårdvaran beskrivs i termer av IO-metoder enligt klassen `WiperIO` nedan. Notera att `awaitHandle` blockerar tills föraren har gjort någon förändring på kontrollspaken och att `sampleSensor` samplar utan att blockera. Sampling behövs dock bara i automatiskt läge. Metoden `commandWiper` exekverar också utan blockering, men det finns några saker att notera:

1. När en hastighet skild från noll (SLOW eller FAST) har angivits rör sig torkarna fram och tillbaka kontinuerligt helt automatiskt tack vare torkarnas mekanik utan att några särskilda ytterligare kommandon behöver ges.
2. När stopp beordras under en rörelse fortsätter torkarna automatiskt fram till viloläget.
3. När en hastighet skild från noll anges efter ett stopp, i syfte att starta en enstaka torkning av rutan i samband med lägena INT eller AUT (dvs när varje enskild torkning startas separat), måste utsignalen hållas satt i 0,8s (för att torkarna ska hinna röra sig ur viloläget).

```
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() {...}
}
```

Med ovanstående kunskap om applikationen och styrgränssnittet är din uppgift att designa och implementera mjukvaran som kontrollerar torkarna. Ta hänsyn till följande extra förutsättningar nedan.

Högnivådesign

1. Baserat på tidigare projekt där buffring av data resulterade i fördröjningar som var svåra att förutsäga/testa finns det ett ledningsbeslut om att buffring endast är tillåten för att jämna ut lasten vid tunga beräkningar eller för nätverkskomminikation.
2. I t.ex. läge INT genereras en utsignal under 0,8s för att starta torkarna och det är praktiskt att generera denna fördröjning med sekventiell kod i en enda tråd. Parallellt med detta kan föraren ge nya input till systemet via kontrollspaken, vilka ska registreras (men inte buffras) och behöver inte tas om hand förrän startpulsen är avslutad.
3. Sampling av regnsensorns värde görs periodiskt med en period på 100 ms i automatiskt läge. I andra funktionslägen ska sampling inte utföras.

Visa i termer av en beskrivande figur hur du vill designa systemet. Notera att krav 1 ovan tvingar fram en monitorbaserad lösning. Varför är det vanligtvis en god idé att ha få stora monitorer i stället för många små och varför är det vanligtvis *inte* mera kostsamt i termer av exekvereingstid eller fördröjningar?

Följande utgör ett förslag på ett lämpligt monitorgränssnitt.

```
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)

Implementering och detaljerad design

Med relevanta motiveringar kan du, om du vill, förbättra monitorförslaget ovan i termer av argument eller features. På grund av säkerhetskraven i bilbranschen får trådar bara startas när applikationen startar. Implementera monitorn tillsammans med run-/performmetoderna i de nödvändiga trådarna.

(8p)

9. Android

För att skapa en öppen och fri programvaruplattform för mobila tillämpningar introducerade Google nyligen Android (`http://code.google.com/android`). Följande beskrivning är baserad på utdrag från den nätplatsen, där man också kan läsa att Linux är den huvudsakliga målsystemplattformen, att systemet är Java-baserat (program skrivs, kompileras och kan köras som Java, men den korskompilerade koden körs inte på en JVM). Ansatsen ligger således i linje med den som gäller för Lund-varianten, men eftersom tillämpningsområdet gäller mobila interaktiva program snarare än inbyggda styrsystem så skiljer det en hel del vad gäller struktur och tillgängliga klasspaket.

Din uppgift är att beskriva hur en plattform liknande Android kan konstrueras med hjälp av de Java-klasser och den funktionalitet för jämlöpande processer som du känner till från kursen. Ditt svar ges i termer av:

- En figur (eller några figurer) som visar de involverade klasserna/objekten, samt hur dessa relaterar till varandra i termer av synkronisering och signalering.
- Klargörande av (i figuren och/eller i text) hur de (semi-)jämlöpande aktiviteterna i Google:s design kan åstadkommas genom att använda ordinarie Java-trådar.
- Det skall vara tydligt hur processer och trådar (i Android-mening) relaterar eller åstadkommes med dina normala Java-trådar, och hur meddelandebaserad kommunikation med eventobjekt hanteras. Kodsnuttar kan vara förtydligande.

Du kan anta en generell tillämpning eller en specifik exempeltillämpning (beroende på hur du bäst beskriver din lösning). Som specifikation gäller följande utdrag från Google:

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 hierarchybased 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)*