

## **Tentamensskrivning — Realtidsprogrammering** **2002-08-22, kl 8.00-13.00**

### **Anvisningar**

Tillåtna hjälpmedel: inga utöver Java snabbreferens.

För godkänt betyg krävs att större delen av de 8 första uppgifterna samt en av uppgifterna 9 eller 10 behandlas nöjaktigt. För högre betyg krävs naturligtvis mer, så gör så många uppgifter du kan. **Svar måste motiveras** om inget annat anges.

Senast 2002-08-29 anslås på institutionens anslagstavla samt på kursens hemsida vilka som deltagit i tentamen men som, enligt institutionens noteringar, ännu inte redovisat övnin-garna, laborationerna eller projektet. Deras skrivning rättas inte (och kan på sikt komma att annulleras) såvida inte rättelse skett eller dispens erhållits från ansvarig lärare. Rättelse skall göras senast 2002-09-05.

### **Uppgifter**

**1.** Vid rapportering av fel och brister beträffande programvaror så pratar man ofta om *reproducerbarhet*. Exempelvis finns det till Bugzilla för Mozilla ([www.mozilla.org](http://www.mozilla.org)) följande alternativ för reproducerbarhet (How often can you reproduce the problem?): **alltid** (Every time), **ibland men inte alltid** (Sometimes, but not always), har inte försökt reproducera det (Haven't tried to reproduce it), **försökt men kunde inte reproducera** (Tried, but could not reproduce it). Skillnader i indata (musklick, filinnehåll, etc.) kan givetvis ge skillnader i beteendet även för rent sekventiella program, men här antar vi att upprepade tester/försök göres med samma indata. I detta fall finns inga speciella tidskrav på programvaran, som dock innehåller ett flertal jämlöpande aktiviteter.

**a)** Vilken sorts fel kan det vara om felet inträffar *alltid*? D.v.s., kan vi veta om det är ett logiskt fel (i de sekventiella delarna) eller om det är ett jämlöpandefel? (1p)

**b)** Dito, i fallet *ibland men inte alltid*? (1p)

**c)** I fallet *försökt men kunde inte reproducera* kan det vara mycket svårt att hitta felet med vanlig test och felsökning. Skall man tro på att det verkligen finns ett fel? Kan det finnas (eller komma till) kunder som råkar ut för probelemet mera ofta? Vilken sorts programmeringsfel kan man leta efter i källkoden? (1p)

**2.** I UNIX kan program (OS-processer) kopplas efter varandra via s.k. pipes (anges med '|') i vilka utdata (som en sekvens av bytes) från programmet före '|' skickas som indata till programmet efter '|'. Exempelvis med "prog1|prog2" får prog2 sina indata från prog1.

På motsvarande sätt kan indata skickas till eller från fil genom att ange '>' respective '<' mellan program och filnamn. Exempelvis gör "prog1>fil" att utdata från prog1 skrivs (som en sekvens av bytes) till filen fil.

Körande program är aktiviteter eller aktiva objekt. Filer däremot är passiva genom att de bara lagrar data utan att själva driva någon exekvering.

**a)** På motsvarande sätt (inuti OS-processer) jobbar vi med aktiva och passiva Java-objekt. Hur skiljer sig dessa objekttyper åt beträffande basklasser och skapande? (1p)

**b)** Varför kan man i UNIX (om infil inte är exekverbar) inte skriva infil>utfil? D.v.s. varför kan man inte flytta innehållet i infil till utfil på detta sätt? (1p)

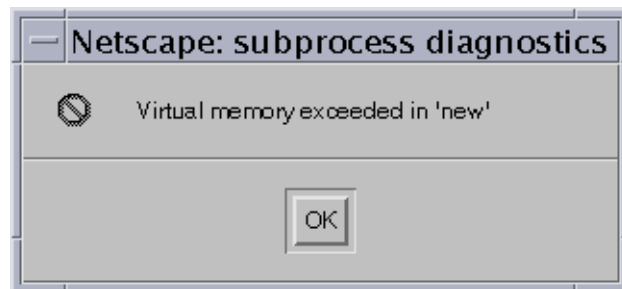
**3.** Varför vill man ibland använda prioriteringsprotokoll såsom *priority ceiling* eller *immediate inheritance*? Med andra ord, nämn en nackdel med det vanliga *priority inheritance* protokollet. (Max 2 meningar) (1p)

**4.** Ett realtidssystem är implementerat med hjälp av fyra trådar, A, B, C och D. Trådarna är periodiska, helt oberoende av varandra och schemaläggs enligt earliest deadline first scheduling (EDF). Maximal exekveringstid (C), periodtid (T) och deadline (D) för varje tråd framgår av följande tabell:

Tråd	C (ms)	T (ms)	D (ms)
A	5	32	18
B	3	12	5
C	2	8	8
D	4	20	10

- a)** Vad blir svarstiden för var och en av de fyra trådarna i värsta fall? Vissa formler och antaganden (för periodiska fixprioritetsprinciper) gäller inte för EDF. Svara därför med ett uppriktat schema för de första 32 ms där samtliga trådar är redo vid tiden noll. (2p)
- b)** Är systemet schemalägningsbart? Om ej, ange vad problemet är. (1p)
- c)** Nämn en fördel och en nackdel med EDF. (1p)

**5.** Många programvaror kan få slut på minne p.g.a. att objekt inte avallokeras trots att de inte används mera, s.k. minnesläckage. Det kan då för användaren se ut enligt vidstående bild, vilket inte vållar några större besvär för vanliga skrivbordstillämpningar. I fallet inbyggda datorer, som styr fordon och annan viktig utrustning, kan en minnesläcka vara mycket problematisk.



- a)** I Java (och vissa andra språk) tar man inte bort objekten manuellt, utan detta sker automatiskt. Vad kallas den del i run-time-systemet som sköter minneshantering/städningen? (1p)
- b)** Nämn ett problem med automatisk minneshantering för realtid, och en teknik för att hantera problemet. (1p)

**6.**

- a)** Nämn två typer av programmeringsfel som gör att en metod inte blir *reentrant* (dvs inte trådsäker eller återanropningsbar). (2p)
- b)** Varför behöver exempelvis funktionerna/metoderna `sin` och `cos` i standardklassen `Math` inte vara `synchronized`, trots att de kan anropas från många trådar parallellt? (1p)

**7.** Hur går ett kontextbyte (context switch) mellan två trådar till? (1p)

**8.** Beroende på OS och JVM-version så följs trådarnas prioriteter antingen strikt (mera hög-prioriterad får alltid köra före en med lägre prioritet) eller inte (för att interaktiva system skall bete sig i någon mening bättre). Varför vill man för inbyggda programvaror normalt ha strikta prioriteter? Varför behöver man kunna sätta tråders prioritet? (2p)

## 9. Tidskontroll av metदानrop

Sune Sanntid leder utvecklingen av ett företags realtidsprogramvaror. Han vet från projektledning att uppkomna förseningar hanteras bättre ju tidigare de upptäcks. Harald Hacker har skrivit en regulator i form av en Java-tråd som periodiskt anropar fem olika metoder enligt följande

```
public class Regul extends RTThread
{
    OneClass obj1 = new OneClass();
    AnotherClass obj2 = new AnotherClass();
    int period;
    long time;

    public void run()
    {
        time = System.currentTimeMillis();
        try {
            while (!interrupted()) {
                obj1.q();
                obj2.y();
                obj2.x();
                obj2.z();
                obj1.p();
                time += period;
                sleepUntil(time);
            }
        }
        catch (RTDelayed exc) {
            Supervisor.GracefulShutdown();
            return;
        }
    }
}
```

där vi antar att `sleepUntil` är sådan att `RTDelayed` (subklass av `Error`, dvs kan fångas men krävs ej) kastas om svarstiden för samplet överskrider samplingsperioden. För att renodla tidsaspekterna så antar vi att metoderna i `obj1` och `obj2` saknar argument, d.v.s.

```
public class OneClass {
    public void p() {...};
    public static final int p_WCET = 6;

    public void q() {...};
    public static final int q_WCET = 11;
}

public class AnotherClass
{
    public void x() {...};
    public static final int x_WCET = 14;

    public void y() {...};
    public static final int y_WCET = 9;

    public void z() {...};
    public static final int z_WCET = 1;
}
```

där konstanterna `_WCET` anger värstafalls exekveringstid (Worst Case Execution Time) i ms för respektive metod.

För att upptäcka om någon metod inte uppfyller sin WCET (direkt efter anropet) så implementerar Harald kapslande klasser i vilka tiderna kontrolleras och `RTDelayed` kastas om angiven WCET inte uppfylls. I anropande tråd ändras bara instantieringen av `obj1` och `obj2` enligt

```
OneClassProxy obj1 = new OneClassProxy(new OneClass());
AnotherClassProxy obj2 = new AnotherClassProxy(new AnotherClass());
```

och `Proxy`-klasserna som kapslar de ursprungliga klasserna utformats enligt följande:

```

public class OneClassProxy {
    OneClass target;

    OneClassProxy(OneClass target){
        this.target = target;
    }

    public void p() {
        long time = System.currentTimeMillis();
        target.p();
        int lag =
            (int)(System.currentTimeMillis()
                - time - target.p_WCET);
        if (lag > 0) {
            throw new RTDelayed(
                time+target.p_WCET, lag);
        }
    }

    public void q() {
        long time = System.currentTimeMillis();
        target.q();
        int lag = // etc. etc. as for p ....
    }
}

public class AnotherClassProxy
{
    AnotherClass target;

    AnotherClassProxy(AnotherClass target) {
        this.target = target;
    }

    public void x() { // As for p.
        long time = System.currentTimeMillis();
        target.x();
        int lag =
            (int)(System.currentTimeMillis()
                - time - target.x_WCET);
        if (lag > 0) {
            throw new RTDelayed(
                time+target.x_WCET, lag);
        }
    }

    public void y() { // Dito. ... }
    public void z() { // Dito. ... }
}

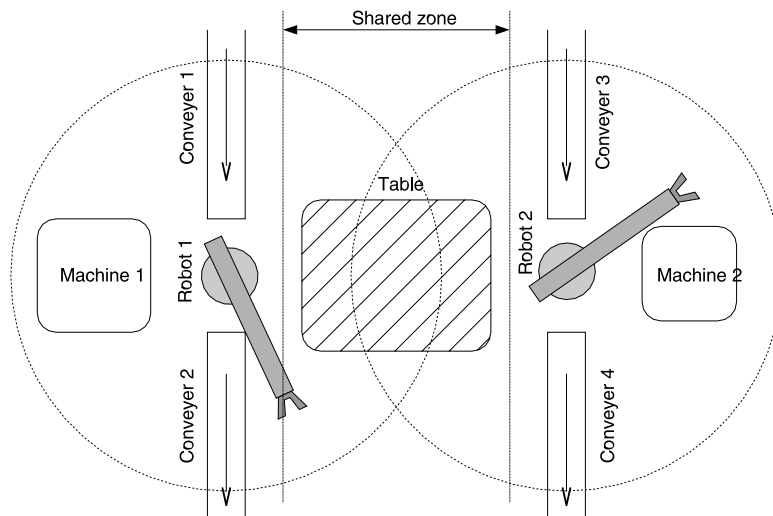
```

Efter denna förbättring trodde Harald Hacker att Sune Sanntid skulle vara helt nöjd, men så var icke fallet. Om någon av de anropade metoderna skulle ta mycket längre tid än vad som är acceptabelt (exempelvis p.g.a. en iteration som baseras på ett mätvärden som kanske kan göra att iterationen inte avslutas) så kommer inte förseningen att upptäckas förrän metoden kört klart. Sune kräver att **omedelbart efter att tiden för ett anrop är ute så skall ett RTDelayed kastas till anroparen**, som direkt efter felhantering skall kunna fortsätta regleringen med fortsatt fungerande tidsövervakning. Byt ut Proxy-klasserna mot något som uppfyller Sunes krav! Du får ändra i anropande tråd-klass och lägga till nya klasser, men inte ändra i OneClass eller i AnotherClass. Ev. begränsningar och designval skall kommenteras.

Anm: Ev. behov av jämlöpande tidmätning (timeout) kan lösas med egen timertråd, men oftast ännu enklare med anrop av `wait(timeout)` i en monitor vilket utnyttjar den timertråd som finns inbyggd i JVM:en. (9p)

## 10. Tillverkningsystem

I en fabrik finns två robotar som var och en har ett transportband in, ett transportband ut, samt en maskin i vilken arbetsstycken sätts för bearbetning. Dessutom finns ett bord som båda robotarna kan nå, för att överlämna arbetsstycket till den andra roboten för bearbetning i den andra maskinen, eller för leverans via det andra transportbandet. Denna s.k. produktionscell (mycket idealiserad) illustreras i vidstående figur.



Bordet befinner sig i ett gemensamt arbetsområde (Shared zone i figuren). I denna uppgift gäller följande krav, förenklingar och antaganden:

- Endast en robotarm åt gången får befinna sig i det gemensamma arbetsområdet.
- Resursallokering med avseende på undvikande av dödläge behöver inte beaktas (utan kan ses som krav på arbetsbeskrivningarna vilka är givna).
- Varje robot kan bara utföra ett antal s.k. enhetsoperationer, som i detta fall inskränker sig till `move`, `pick` och `place` med tillhörande argument.
- En arbetsuppgift består av en sekvens av operationer som skickas till en robot. Denna kan då samverka med den andra roboten, om båda robotarna givits samhörande arbetsuppgifter. Planering av samhörande uppgifter tillhör inte uppgiften; robotarna styrs individuellt (men kollisionsrisken skall enligt ovan beaktas i den gemensamma zonen).

Samtlig utrustning hanteras via färdiga klasser, som dock saknar monitorskydd. Din uppgift är att utveckla styrningen av respektive robotarm så att ömsesidig uteslutning etc. åstadkommes.

Ett omgivande huvudprogram (antas finnas) skapar robotobjekten med referenser till omgivande objekt som argument:

```
/**
 * The robot accepts work descriptions and performing them by calling
 * the MotionControl etc.
 */
class Robot extends RThread {

    MotionControl gripper;

    Robot(Conveyer input, Conveyer output, Machine process, Table table) {
        // ...
    }

    void todo(Task work){ /*....*/ }

    // Fill in the code to make this class complete.....
}
```

Klassen `MotionControl` antas finnas och styr robotarmen till de förutdefinierade positionerna.

```
/**
 * The capabilities of the robot motion controller.
 */
class MotionControl
{
    /** Move arm/gripper to destination. */
    void move(Station dest) { /*....*/ };

    /** Move to Station and pick up whatever object there is. */
    boolean pick(Station where) { /*....*/ };

    /** Move to Station and place the currently held object there. */
    void place(Station where) { /*....*/ };
}
```

I Robot skall således dessa metoder anropas baserat på den arbetsbeskrivning i form av textsträngar som tillhandahålles via anrop av `todo(Task work)` där klassen `Task` mycket förenklat definieras som en lista (array) med operationer:

```
class Task extends REvent
{
    Operation sequence[];
}
```

där `Operation` utgörs av

```

class Operation
{
    public Operation(String what, String where, String with) {
        this.what = what; this.where = where; this.with = with;
    }
    String what, where, with;
}

```

Här anger `what` vilken typ av operation som roboten skall utföra, d.v.s. något av "move", "pick", "place" eller "make". Vidare anger `where` platsen (målpositionen) för rörelsen, och om ett föremål skall gripas så kan man med `with` ange att endast en viss typ av föremål önskas. Antag att föremålen som kommer på transportbandet till roboten är plate och cylinder, att maskinen är drilling, och att bordet fått namnet store. Att ta in en platta, borra denna, och sedan placera den på bordet, samt därefter ta en cylinder, borra den, och placera den på utbandet. Detta kan då anges med (i verkliga system har man speciella robotspråk för detta, men det leder utanför kursen):

```

Task task1 = new Task();
task1.sequence = new Operation[6];
task1.sequence[0] = new Operation("pick", "in", "plate");
task1.sequence[1] = new Operation("make", "drilling", "");
task1.sequence[2] = new Operation("place", "store", "");
task1.sequence[3] = new Operation("pick", "in", "cylinder");
task1.sequence[4] = new Operation("make", "drilling", "");
task1.sequence[5] = new Operation("place", "out", "");

```

Den färdiga basklassen `Station` är gemensam för alla definierade platser i robotens arbetsområde. Som en förenkling använder vi namn direkt och antar att dessa anger kända positioner. Denna klass utgörs av:

```

class Station
{
    String description; // For documentation.
    Station(String description) {this.description = description;}
    // Coordinates etc., but nothing of that needed here.
}

```

Transportbanden hanteras via `Conveyer` som är en färdig monitor, inkl. identifiering av vilken sorts objekt som finns i gripposition.

```

class Conveyer extends Station
{
    Conveyer() {super("");}

    /** Tell the next type of work piece, return null if none available. */
    synchronized WorkPiece isNext() {/*....*/}

    /** Block until work piece available, then grasp it and return type. */
    synchronized WorkPiece takeNext() {/*....*/}

    /** Put piece on conveyer belt which starts automatically. */
    synchronized void deliver(WorkPiece part) {/*....*/}
}

```

Arbetsstycken (hanterade föremål) representeras av klassen `WorkPiece` och har namn som används som tredje argument till operation ovan, d.v.s. för att ange vilken sorts föremål man vill plocka från ingående transportband. Exempelvis kan omgivande system ha skapat följande objekt för två inkommande föremål

```

WorkPiece part1 = new WorkPiece("plate");
WorkPiece part2 = new WorkPiece("cylinder");

```

genom användning av den färdiga klassen:

```
/**
 * Base class representing any manipulated physical object.
 */
class WorkPiece {

    /** CAD data etc. whould go here, but just a name is fine here. */
    String sort;

    public WorkPiece(String kind) {sort = kind;}
}
```

Den maskin som roboten känner till (via sin konstruktör) startas (efter place av arbetsstycket där) genom anrop av process-metoden i

```
class Machine extends Station
{
    Machine(String type) {
        super(type);
    }

    void process(WorkPiece object)
    {
        // ...machine control via static object is here, NOT to implement.
    }
}
```

där vi antar att bearbetningen (initierad av anropet av process) alltid tar 5 sekunder, d.v.s. pick får inte anropas tidigare än så efter anropet av process.

### Uppgift:

Utveckla övriga delar av systemet, vilket innebär

- implementering av klassen Robot, helst så att flera arbetsuppgifter kan buffras,
- styrning av maskinerna så att roboten kan hantera annat arbetsstycke då maskinbearbetning pågår, och så att man väntar på maskinen utan pollning,
- hantering av gemensamt arbetsområde, d.v.s. ömsesidig uteslutning och signalering så att endast en robot åt gången kan genomföra sitt anrop av move, pick eller place med den gemensamma bordet som argument,

samt eventuella övriga hjälpklasser och dokumenterade antaganden. I den sista punkten ingår även utveckling av klassen Table som antas vara subclass till Station. Givetvis skall pick av ett föremål från bordet resultera i blockering av roboten om bordet är tomt.

(9p)

