

Lösningförslag — Realtidsprogrammering — Java 1999-12-17

Uppgifter

1. Vilken/vilka av programkonstruktionerna semaforen, monitorer och meddelanden är mest kraftfull beträffande vad som kan uttryckas? Observera att det inte gäller enkelhet eller vad som är mest praktiskt i olika situationer, utan bara uttryckskraften. (1 p)

Eftersom vilken som helst av konstruktionerna kan uttryckas/implementeras med vilken som helst av de övriga, så har de **samma uttryckskraft**. Med andra ord, ett realtidsprogram som implementerats med hjälp av en av konstruktionerna kan alltid skrivas om baserat på någon av de andra konstruktionerna. (Detta kan dock kan var opraktiskt, men det tillhör inte denna fråga.)

2. I ett lexikon översätts den engelska termen pre-emption med 'förköpsrätt'. Inom realtidsprogrammering används ordet i två sammanhang:

a) Pre-emptive multi-threading/tasking/processing, vilket vi kallar påtvingad tidsdelning, och

b) Resource pre-emption, vilket ingår i bland villkoren för dödläge (eng: dead-lock).

Förklara vad de båda begreppen innebär. Förklara också varför man vill ha, eller inte ha, dessa egenskaper i realtidssystem. (2 p)

a) Påtvingad tidsdelning innebär att exekveringen av en körande aktivitet (OS-process eller tråd) kan avbrytas utan att den avbrutna aktiviteten har medgivit eller initierat detta. Man kan säga att aktiviteter med högre prioritet så att säga har förköpsrätt/förtur till varan/resursen CPU-tid.

b) För övriga gemensamma resurser (dvs inte tiden enligt **a**) inför vi ömsesidig uteslutning för att vi trots parallellitet skall erhålla korrekta resultat. Detta innebär att resurser kan reserveras exklusivt för en aktivitet, exempelvis med hjälp av en monitor vilket bygger på **no resource pre-emption**. Detta innebär att då en aktivitet reserverat resursen har **ingen** annan aktivitet (oberoende av prioritet) förköpsrätt/förtur till resursen, utan innehavaren av en exempelvis en monitor måste själv lämna den ifrån sig genom återhopp från `synchronized`-metoden eller genom anrop av `wait`.

3. För att vänta på att ett villkor skall bli uppfyllt innan en monitoroperation kan fullbordas använder vi metoden `wait()`;

a) Varför kan `wait` endast användas i monitorer, dvs i metoder som deklarerats som `synchronized`?

b) Antag att vi har ett program med bara två trådar. Vilken typ av fel kan generellt sett inträffa om vi omger `wait` med `if` i stället för `while`? Med andra ord, vad kan gå fel om man skriver `if (cond) wait();` i stället för `while (cond) wait();`? (3 p)

a) Visserligen kan `wait` alltid anropas eftersom den ärvs från klassen `Object`, men det är endast innanför `synchronized` som tråden är i en monitor och kan blockeras vilket innebär att tråden ställer sig i en väntekö tillhörande monitorn. Utanför `synchronized` saknar runtime-systemet referens till monitorn och vi får istället ett `MonitorStateException`.

b) Antag att den ena tråden exekverat `if (cond) wait();` och blockerats eftersom `cond` var `true`. Ett av följande tre exempel på fel räcker som svar:

- Efter att `cond` blivit `false` och `notify()` anropats i en annan monitorermetod av den andra tråden så kan denna tråd p.g.a. högre prioritet fortsätta med andra monitoranrop som kanske sätter `cond` till `true` igen innan den första tråden får fortsätta, och därefter kan den första tråden då fortsätta sin operation trots att villkoret inte är uppfyllt.
- Den andra tråden kan anropa `notify` i en annan monitorermetod utan att ha ändrat `cond` (som kanske inte ens är relevant för metoden).

- Även om `cond` är korrekt hanterad/skyddad och det bara finns en metod i vilken `notify` anropas så kan osnygg eller illasinnad kod inte förhindras att var som helst där man har en referens till objektet (som vi kan kalla `obj`) kasta in ett extra `notify` genom att


```
synchronized(obj){notify();}
```

 (Detta fall ingår inte i kursen men kan vara bra att känna till.)

4. En klass har två attribut och fyra metoder Vad innebär nyckelordet `synchronized` som används för metoden `setValue`? Antag att det finns två trådar `T1` och `T2` som båda har referenser till två instanser av `Complex`, `C1` och `C2`. Svara genom att ange vilken/vilka av följande påståenden som är sanna:

1. Då `T1` utför `C1.setValue(1.0,1.1)`; kommer ett anrop i `T2` av `C2.setValue(2.0,2.2)`; att blockeras tills `C1.setValue` exekverats klart av `T1`.

Falskt. Eftersom `C1` och `C2` är **olika objekt** (och `setValue` inte är `static`) så är de inte ömsesidigt uteslutande.

2. Då `T1` utför `C1.setValue(1.0,1.1)`; kommer ett anrop i `T2` av `C1.setValue(2.0,2.2)`; att blockeras tills `C1.setValue` exekverats klart av `T1`.

Sant. Det är **samma objekt** och `setValue` är `synchronized`.

3. Under förutsättning att en variabel av typen `float` tilldelas/returneras odelbart så kan man utan att påverka programmets korrekthet (som en sorts handoptimering vilket dock inte rekommenderas) utelämna `synchronized` (såsom gjorts ovan) framför operationerna `getRe` **och** `getIm`.

Sant. Läsning av odelbart värde kan göras. Givetvis kan vi inte anta att två på varandra direkt följande anrop av `getRe` och `getIm` returnerar delarna från samma tal.

4. Under förutsättning att en variabel av typen `float` tilldelas/returneras odelbart så kan man utan att påverka programmets korrekthet, som en sorts handoptimering, utelämna `synchronized` (såsom gjorts ovan) framför operationen `getAbs`.

Falskt. På detta sätt **kan** `getAbs` **avbryta** `setValue`, exempelvis efter att realdelen tilldelats men med imaginärdelen kvar från föregående tal.

5. Eftersom `setValue` är `synchronized` så kan `T2` inte exekvera (någon kod) under tiden som `T1` utför ett anrop av `setValue`.

Falskt. `synchronized` innebär att `T1` har reserverat en resurs (ett objekt) men inte processorn, så `T2` kan köra annan kod.

6. Eftersom `setValue` är `synchronized` så kan `T2` inte exekvera metoden `getAbs` under tiden som `T1` anropar `setValue`.

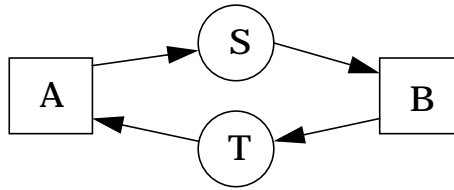
Falskt. Även om det rör sig om samma objekt så är `getAbs` inte `synchronized` och `T2` kan därför komma att schemaläggas till att köra samtidigt som `setValue` utförs av `T1`.

(3 p)

5. Betrakta följande system som kommunicerar via brevlådor (av typen `RTEventBuffer`) för att trådarna senare skall kunna distribueras till olika datorer: brevlådorna `A`, `B` och `C` är på förhand instantierade.

a) Rita en resursallokeringsgraf och förklara varför och hur det kan bli 'deadlock'

Det förekommer tre meddelandeköer. Av dessa representerar `C` ett rent dataflöde med `R` och `S` som producenter och `T` som konsument. Detta kan vi bortse från i uppgiftern eftersom det inte inte utgör `hold-wait`. Kvar har vi då `R` och `S` med resurserna `A` och `B` som vid start (enl separat information som gavs under tentamen) innehåller vars ett meddelande. För `A` och `B` blir då `doFetch` ekvivalent med `take` och `doPost` ekvivalent med `give`, och vi ehåller enkelt grafen:



vilken visar cirkulärt beroende, d.v.s. risk för dödläge.

b) Ändra programmet så att dödläge inte kan uppstå.

Vi byter allokeringsordning. I detta fall lämpligast genom att byta ordning på de två första raderna i R:s while-loop. Detta förlänger inga blockeringstider eftersom inga anrop av (ev tid-skrävande) operationer sker mellan dessa båda rader.

(2 p)

6. Vilka längsta svarstider erhålls enligt Deadline Monotonic Scheduling?

(2 p)

Prioriteterna blir enligt DMS i ordningen BACD, säg från 1 till 4. Längsta svarstiderna får genom exakt schemaläggning från läget att alla trådar vill starta samtidigt. Detta ger tabellen

Thread	Period	Exec.time	Deadline	Priority	Response time
A	12	3	6	2	4
B	18	1	4	1	1
C	8	4	10	3	8
D	12	2	12	4	22

där vi ser att tråd D inte klarar sina tidskrav, vilket dock inte efterfrågades.

7. Vad är och vad utförs vid ett kontextbyte (eng: context switch)?

(1p)

Byte av exekveringsomgivning, vilket innebär att tillståndet för körande aktivitet sparas undan och sedan hämtas tidigare sparade tillstånd för den aktivitet som skall köras. För trådar innehåller ett 'kontext', d.v.s. en omgivning, innehållet i aktuell programräknare, stack-pekare samt övriga hårdvaruregister. (För OS-processer ingår även öppna filer, sockets, användarrättigheter, minnesynder, etc.)

8. I den inledande beskrivningen till uppgift 11 nedan förekommer en känd synkroniseringsprincip som implementerats med två semaforer. Vad kallas denna princip/mekanism?

(1 p)

Rendez- vous, vilket framgår av OH-bilderna gällande synkronisering (1999: bild F2-14).

9. Ett av de vanligaste realtidsoperativsystemen för större inbyggda system är VxWorks vars innersta del består av en s.k. mikrokärna som tillhandahåller realtidsprimitiverna. I dokumentationen (och vagt i beskrivningen på <http://www.wrs.com/products/html/vxwks52.html>) kan man notera att det förekommer tre typer av semaforer: **a) Binär semafor utan prioritetsärvning, **b) Räkande** semafor utan prioritetsärvning och **c) 'Mutual-exclusion'** semafor med prioritetsärvning.**

Beskriv med programkod en lämplig användning av var och en av dessa semafortyper!

Information: En binär semafor innebär att den interna räknare som finns i kursens räkande semafor bytts ut mot en boelsk variabel så flera anrop av give (utan take emellan) ger ingen ytterligare verkan.

(3p)

Semaforer utan prioritetsarv lämpar sig inte för skydd av gemensamma resurser, däremot duger det för synkronisering och då ger avsaknad av prioritetsarv ger en viss prestandahöjning. Med en binär semafor signaleras alla väntande trådar samtidigt. Kodexempel (de olika kolumnerna med kod körs av olika trådar):

a) En tråd släpper vidare ett okänt antal väntande trådar:

```
// Main tread allocating IO-ports:           // In all control threads, do:
ioStruct = IO.AllocateAllPorts();           // Wait for IO to be initiated:
// Let all control treads continue:         ioSem.take();
ioSem.give();
```

Alternativt **signalerar** en tråd till en annan som ifall den blir efter mer än ett steg så skall den **inte köra ikapp**:

```
// New target available, unprocessed       // Wait for new target:
// tagrets are to be skipped:             newTarget.take();
newTarget.give();                         // Obtain current taget data:
                                           target = supervisor.obtainTarget();
```

b) En tråd som tillåts köra i förväg ett antal steg signalerar till en annan tråd att fortsätta:

```
// Allow player to advance one tick:       // Wait for tick to be available:
tick.give();                               tick.take();
// Player may have to catch up...         step = queue.getNextStep();
```

c) Skyddande av gemensam resurs med krav på begränsad blockeringstid för trådar med högre prioritet:

```
// Put in salary:                          // Get some cash:
account.deposit(amount);                  amount = account.withdraw();
```

10 Parkeringshus

I ett parkeringshus Detaljbeskrivning och specifikationer:

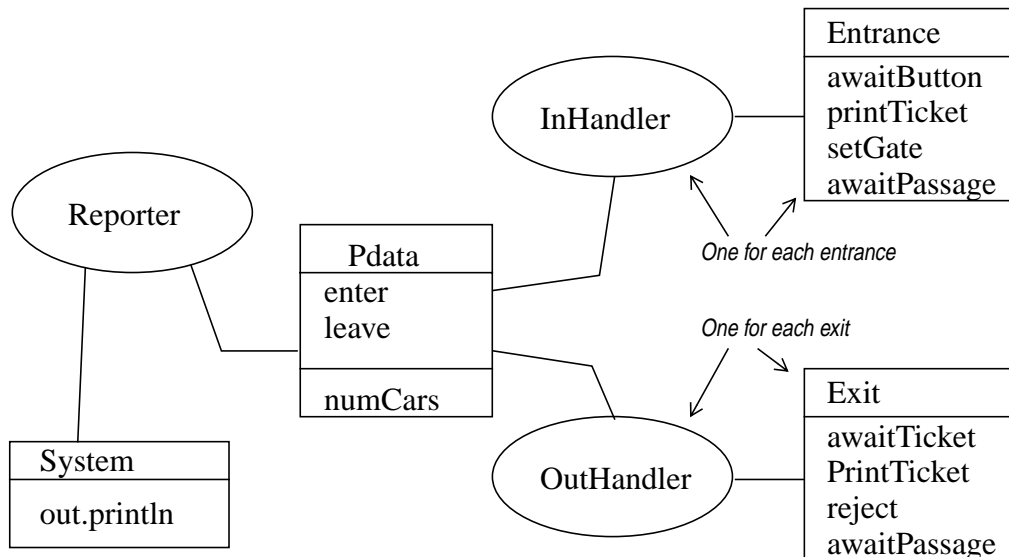
1. *Din programvara startas från konstruktorn i klassen `Parking` som skall utvecklas utifrån koden nedan. Argumentlistan bör vara självförklarande.*
2. *Då får anta att det finns just två ingångar (`Entrance-objekt`) och två utgångar (`Exit-objekt`).*
3. *Vid varje ingång och vid varje utgång finns en grind/bom som öppnas endast vid in-/utpassage av bilar.*
4. *Vid varje ingång finns en knapp som anländ kund trycker på för att få en biljett utskriven med ankomsttiden noterad. Efter att kunden tagit sin biljett skall grinden öppnas, och hållas öppen tills bilen passerat en ljusbom efter grinden. Anrop av metoden `awaitPassage` blockerar tills ljusbomen först brutits och sedan slutits igen, dvs tills bilen passerat helt.*
5. *Då parkeringen är full skall knappen för ny biljett inte resultera i någonting. Kunden får vänta. Om det finns väntande kunder vid två grindar och det blir en plats ledig så skall den som först trycker igen för biljett få komma in först.*
6. *Vid varje utgång kan kunden sätta in sin biljett för att få grinden öppnad. Den tid till vilken betalning erlagts returneras då av metoden `awaitTicket`. Om klockan då inte passerat betalningstiden `+allowedTimeToExitAfterPayment` så skall grinden öppnas och sedan stängas igen efter passage av ljusbom. Om betalning skett alltför långt tid i förväg, eller om betalning inte alls skett vilket gör att `awaitTicket` returnerar noll, så skall `reject` anropas vilket informerar kunden.*
7. *Var 5:te minut skall aktuell beläggning (hur många procent av platserna som används) skrivas ut (på `System.out`). Vid stängningsdags skall antalet kvarvarande (för natten inlästa) bilar skrivas ut.*

Samtliga tider har antagits räknade i millisekunder enligt `System.currentTimeMillis` (dvs räknat från kl 00.00 den 1/1 1970). (8 10 p)

Det önskade systemets parallellitet består i att samtidigt hantera ingrindar och utgrindar, samt tidsperiodisk rapportering. Som alltid skall samtliga ingående trådar vara blockerade då

inget händer och det inte är dags att göra något. Baserat på de externa oberoende händelser och blockerande anrop vi har att ta hand om så finner vi att (enl ELM i föreläsningsbilderna, 1999:F5) varje ingrind skall hanteras av en tråd, varje utgrind av en tråd, samt rapporteringen av en tråd.

Den periodiska rapportaktiviteten behöver data som beror av samtliga in- och ut-portar. En medelandedbaserad lösning skulle innebära att samtliga in- och ut-passeringar skulle skickas till rapportaktiviteten, vilket känns onödigt/onaturligt. Aktuell beläggning är en gemensam storhet av intresse för samtliga ingående trådar. Lämpligen bygger vi därför systemet kring en monitor i vilken antalet bilar kapslas. Denna design åskådliggörs i följande figur.



Monitorn `Pdata` förenklas lämpligen p.g.a. krav #5: Det finns ingen anledning att låta operationen `enter` vara blockerande då det är fullt; programmet blir enklare om vi återår till att invänta knapptryckning som ju skall avgöra vem som får komma in. (Vi antar att det vid två grindar inte trycks så samtidigt att det behövs någon speciell hantering at turordningen annat än vilken tråd som kommer först in i monitorn.) Följande implementering föreslås:

```

public class Parking {

    /**
     * Constructor for the entire parking-house application which is
     * created once for every period that the parking is open
     */
    public Parking(Entrance[] ingate, Exit[] outgate,
                  long openingAt, long closingAt,
                  int maxNumberOfCars,
                  long allowedTimeToExitAfterPayment) {
        // Solution to exercise starts running from here ....
        int i;

        // Even if it is not time to open yet, start all threads which then
        // will be idle but ready to serve at time openingAt.

        // The threads are all operating on ....
        Pdata monitor = new Pdata(maxNumberOfCars);

        Reporter reporter = new Reporter(monitor, openingAt, closingAt);
        reporter.start();

        OutHandler[] outlet = new OutHandler[outgate.length];
        for (i=0; i<outlet.length; i++) {
            outlet[i] = new OutHandler(outgate[i], monitor,

```

```

        openingAt, closingAt,
        allowedTimeToExitAfterPayment);
    outlet[i].start();
}

InHandler[] inlet = new InHandler[ingate.length];
for (i=0; i<inlet.length; i++) {
    inlet[i] = new InHandler(ingate[i], monitor,
        openingAt, closingAt);
    inlet[i].start();
}

// Comment: Generally, after allocation of monitors (passive
// resources), I first create active objects that are
// data-flow independent (the periodic statistic output here),
// then constructing threads that represent data sinks
// (outgates here), going up-streams creating data converters
// (none here), and finally creating the data sources. This
// minimizes the risk for (timing/OS dependent)
// NullPointerExceptions during startup of the system.

// System is up and running. Wait for final report to complete and
// then terminate application by returning to main program.

try {
    reporter.join(); // Wait until it is time to quit.
    for (i=0; i<outlet.length; i++) outlet[i].interrupt();
    for (i=0; i<inlet.length; i++) inlet[i].interrupt();
    Thread.sleep(1000); // Give In/Out-Handlers time to terminate.
} catch (InterruptedException exc) {}
return;
}
}

public class Reporter extends Thread {
    protected long t0, tf; // start and final time.
    protected Pdata data;
    final int h = 2*60*1000; // sampling period.

    public Reporter( Pdata sharedData, long openingAt, long closingAt) {
        data = sharedData;
        t0 = openingAt;
        tf = closingAt;
    }

    public void run() {
        long t, tn; // time now and next sample time.
        System.out.println("Occupation at minute after opening.");
        tn = t0 + h;
        try {
            while (!interrupted()) {
                t = System.currentTimeMillis();
                if (tn-t > 0) sleep(tn-t);
                int percentage = data.nCarsNow()/data.nCarsMax()*100;
                System.out.println(" @: " + (tn-t)/60000 +
                    " %: " + percentage);
                tn += h;
            }
        }
        catch (InterruptedException exc) {return;} // Terminate.
    }
}
}

```

```

public class InHandler extends Thread {
    protected Entrance gate;
    protected Pdata parking;
    protected long t0, tf;        // start and final time.

    public InHandler(Entrance mygate, Pdata sharedData,
                    long openingAt, long closingAt) {
        gate = mygate;
        parking = sharedData;
        t0 = openingAt; tf = closingAt;
    }

    public void run() {
        try {
            sleep(t0-System.currentTimeMillis());
            while (!interrupted() && System.currentTimeMillis()<=tf) {
                do {
                    gate.awaitButton();
                } while(!parking.enter());
                gate.setGate(true);
                gate.awaitPassage();
                gate.setGate(false);
            }
        }
        catch (InterruptedException exc) {return;} // Terminate.
    }
}

public class OutHandler extends Thread {
    protected Exit gate;
    protected Pdata parking;
    protected long t0, tf;        // start and final time.
    protected long tt;           // time tolerance

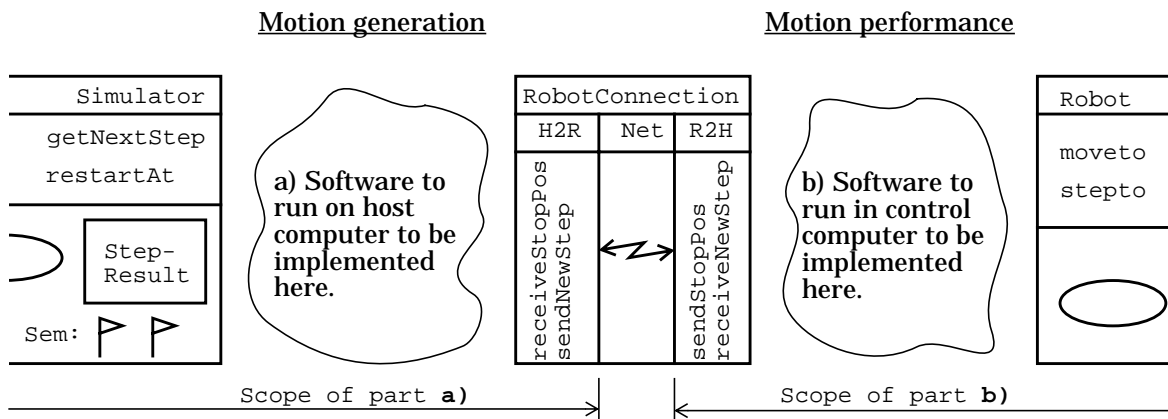
    public OutHandler(Exit mygate, Pdata sharedData,
                    long openingAt, long closingAt,
                    long allowedTimeToExitAfterPayment) {
        gate = mygate;
        parking = sharedData;
        t0 = openingAt; tf = closingAt;
        tt = allowedTimeToExitAfterPayment;
    }

    public void run() {
        long tp;                 // time of payment
        boolean ok;              // payment OK
        try {
            sleep(t0-System.currentTimeMillis());
            while (!interrupted() && System.currentTimeMillis()<=tf) {
                do {
                    tp = gate.awaitTicket();
                    ok = System.currentTimeMillis()<(tp+tt);
                    if (!ok) gate.reject();
                } while(!ok);
                gate.setGate(true);
                gate.awaitPassage();
                gate.setGate(false);
            }
        }
        catch (InterruptedException exc) {return;} // Terminate.
    }
}

```

11. Robotstyrning

I en simulator för dynamiska system sker numerisk integration av en robotarms rörelsedynamik baserat på differentialekvationerna för mekanik och reglering. Tillsammans med grafik utgör detta en virtuell robot. Genom lämpliga val av datorkraft, integrationsmetod, algoritmens steglängd och erforderlig noggrannhet klarar den virtuella roboten för det mesta av att röra sig åtminstone lika snabbt som motsvarande fysiska robot som vi i denna uppgift vill styra.Systemet och dess deluppgifter illustreras i följande figur:



a) Utveckla ett program som anropar `getNextStep` och skickar erhållna steg vidare till en industrirobot som kommunicerar via ett nätverk som nås genom en klass som implementerar ...`Host2Robot`.... Klassen `PosEvent` är definierad

Du skall utveckla övriga delar av värddatorprogrammet så att följande specifikationer uppfylls:

1. Vid start, efter att `getNextStep` anropats första gången, avläses realtidsklockan (dvs systemtiden via `System.currentTimeMillis()`). På så vis kan simulerad tid och realtid relateras till varandra.
2. Tiden i erhållet `StepResult` skall användas som tidstämpel i det `PosEvent` som skickas till roboten (för att vi skall uppnå en korrekt rörelse i uppgift b).
3. Informationen i erhållet `StepResult` skickas snarast med metoden `sendNewStep` som dock kan blockera viss tid då nätverk eller mottagare inte hinner med.
4. För att klara av tillfällena då simulerad tid kan gå långsammare än realtid (vid hög CPU-last) så vill vi då så medges (vid mindre CPU-last) simulera lite i förväg genom att anropa `getNextStep` och lagra resultatet i väntan på skickning. Simulerad tid får dock aldrig ligga mer än tre sekunder före realtiden.
5. Vid val av prioriteter gäller att om vi varken blockerats av simulator (`getNextStep`) eller nätverk (`sendNewStep`) så är det viktigare att skicka redan erhållna steg än att beräkna nya.
6. Allra viktigast är att ta emot ev felmeddelanden som erhålls genom anrop av metoden `receiveStopPos` som är blockerande. Returnerat `PosEvent` anger tid och plats för stoppad robotrörelse. Simulatorens skall då startas om genom anrop av `restartAt` med denna tid och plats som argument.

Parallelliteten i problemet innebär att vi behöver olika trådar för

- anrop av simulatorens¹ och placering av steg i buffert,
- skickning av buffrade steg via nätverket, och
- mottagning av ev. stoppmeddelanden via det blockerande anropet.

vilket innebär tre trådar för genereringen och överföring av robotrörelsedata. Designval:

1. Även om simulatorens i sig drivs av en egen tråd eller process så måste vi hämta ut resultatet. Hade vi istället haft en simulator till vilken man kunde registrera sitt objekt som `StepResultListener` så hade vi sluppit en tråd. Å andra sidan kan vi med nuvarande *rendez-vous* ha simulatorens i en egen OS-process eller körande på annan CPU, utan att detta påverkar val av trådar i övriga delar av tillämpningen.

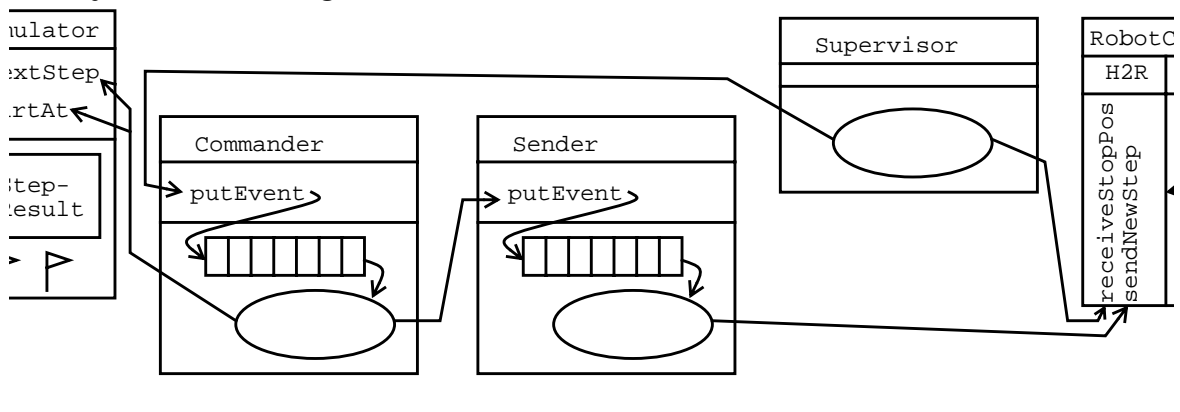
- Den buffert som behövs för beräknade steg kan vara en fristående monitor eller ligga internt i exempelvis en tråd. Här har det senare valts, dvs buffringen är en intern egen-skap hos ett objekt av typen Sender som då lämpligen är en subclass till RTThread vilket innebär att vi anropar Sender.putEvent för buffring.
 - Stegbufferten kan vi antingen göra som en 'tillräckligt stor' RTEventBuffer, eller göra en subclass till RTEventBuffer och implementera själva bufferten med en lista eller en Vector. Det förstnämnda är enklast och rekommenderas för en tentamenslösning, men en större buffert än vad som visar sig behövas slösar med minne. Alternativt kan vi göra en subclass till RTEventBuffer och modifiera doPost så att denna ökar buffertstorleken om bufferten blir full. Detta har gjorts i det följande men krävs som sagt inte.
 - Den modifierade RTEventBufferen kan utgöra en fristående klass, eller läggas internt i den enda klass som (i detta fall) behöver den. Det senare har valts.
 - Det finns också flera acceptabla alternativ vad det gäller hanteringen av stop-eventen. I det följande finns en RTThread Supervisor som tar emot detta från roboten. Man kan sedan **1**) anropa simulatorns restartAt direkt, **2**) skicka vidare eventobjektet till tråden som hämtar stegresultaten (klassen Commander nedan) eller anropa en stophanteringsmetod i Commander men samma effekt, **3**) skicka stoppevent eller göra metodanrop till Sender-objektet som då omedelbart slutar skicka och propagerar stoppet 'motströms' upp till simulator och Commander genom anrop av restartAt etc.
- Av dessa (alla godkända) alternativ så valdes alternativ **2** med vanligt putEvent till en RTThread. Alt 1 är enklast men om man dessutom vill tömma bufferten (ej i spec) så behöver man antingen lite logik till steghanteringen eller ytterligare anrop av övriga object, och dessutom antar man (då man anropar restartAt asynkront med getNextStep) att simulatören är trådsäker vilket man kan göra här men i praktiken kan det vara riskabelt¹. Alt 3 skulle också innebära asynkron hantering av simulatören. Därför valdes alt 2 trots att man då måste 'polla' om stop inträffat men det är acceptabelt då vi har en normal blockering i väntan på nästa steg som utgör det normala dataflödet.

Vi har då följande huvudprogram:

```
public class Master {
    Supervisor supervisor;    Commander commander;    Sender sender;

    public Master(Simulator simRobot, Host2Robot realRobot) {
        (sender = new Sender(realRobot)).start();
        (commander = new Commander(simRobot, sender)).start();
        (supervisor = new Supervisor(realRobot, commander)).start();
    }
}
```

Där följande informella figur visar trädarna och dess relationer:



1. Utanför uppgiften: I detta fall har vi att göra med numerisk programvara som typiskt är implementerad i FORTRAN eller C, eller i vissa fall i C++. Denna typ av programmerare brukar inte tänka på trådsäkerhet (reentrance) utan av effektivitetsskäl (åtminstone på äldre datorer) så allokerar man en statisk arbetsarea (static i file-scope i C/C++ eller common-block i FORTRAN) som man ofta använder från olika metoder. Effektivt då man inte samtidigt anropar flera metoder men blir helt fel om man gör så.

Skickning till robot över nätet implementeras av:

```
import se.lth.cs.realtime.*;
import se.lth.cs.realtime.event.*;

public class Sender extends RTThread {

    Host2Robot target; // Connection to real robot.

    // Mailbox with unlimited size.
    class UnboundedBuffer extends RTEventBuffer {

        public UnboundedBuffer(int size) {super(size);}

        public synchronized void doPost(RTEvent ev) {
            // int incSize = Sender.this.defaultEventBufferSize;
            int incSize = Sender.this.getES();
            if (isFull()) setMaxSize(currentSize()+incSize);
            super.doPost(ev);
        }
    }

    public int getES() { return defaultEventBufferSize;}

    /** Method on thread object; just calls internal monitor/buffer. */
    public void flushEvents() {
        mailbox.flush();
    }

    public Sender(Host2Robot target) {
        super();
        this.target = target;
        defaultEventBufferSize = 256;
        mailbox = new UnboundedBuffer(defaultEventBufferSize);
    }

    public void run() {
        while (!interrupted()) {
            PosEvent ev = (PosEvent) mailbox.doFetch();
            target.sendNewStep(ev);
        }
    }
}
```

Mottagande av stop

```
public class Supervisor extends RTThread {

    Host2Robot realRobot;    Commander commander;

    public Supervisor(Host2Robot realRobot, Commander commander) {
        super();
        try {setPriority(MAX_PRIORITY);}
        catch (FixedPriorityException exc) {};
        this.realRobot = realRobot;
        this.commander = commander;
    }

    public void run() {
        while (!interrupted()) {
            PosEvent stopEv = realRobot.receiveStopPos();
            commander.putEvent(stopEv);
        }
    }
}
```

```

    }
}

```

Hantering av simulatorn:

```

import se.lth.cs.realtime.*;
import se.lth.cs.realtime.event.*;

public class Commander extends RTThread {

    Simulator simRobot;
    Sender stepBuffer;
    Simulator.StepResult stopState; // Used for restart.
    Simulator.StepResult pos;      // Current simulated position.
    long t0sim, t0real;

    public Commander(Simulator simRobot, Sender stepBuffer) {
        super();
        this.simRobot = simRobot;  this.stepBuffer = stepBuffer;
        stopState = simRobot.new StepResult(); // Strange Java syntax but ..
        StepResult is not a static member so the object, not the class,
        // has to be used, and new has to be done in the context of that obj.
        try {setPriority(MIN_PRIORITY);}
        catch (FixedPriorityException exc) {/*impossible*/}
        defaultEventBufferSize = 64; // Even 1 or 2 should be ok.
    }

    public void run() {
        RTEvent ev;
        PosEvent stop;
        pos = simRobot.getNextStep();
        while (!interrupted()) {
            t0sim = pos.t;
            t0real = System.currentTimeMillis();

            while (!interrupted()) { // Send steps until robot stops...
                // Check and possibly take care of stop event
                ev = mailbox.tryFetch(); // Polling ok; blocking below.
                if (ev instanceof PosEvent) stop = (PosEvent) ev;
                else stop = null;
                if (stop != null) {
                    stopState.t = stop.getTicks();
                    stopState.x = stop.x;
                    stopState.y = stop.y;
                    stopState.z = stop.z;
                    simRobot.restartAt(stopState);
                    // Eat pending poses before restart pos.
                    do {
                        pos = simRobot.getNextStep();
                    } while (pos.t > stopState.t);
                    // Flush old unsent steps (not in spec. but good):
                    stepBuffer.flushEvents();
                    // ----- jump to outer loop ----->>>
                    break;
                }
                // Sleep if we are running too much ahead of time.
                long t = System.currentTimeMillis();
                long ahead = (pos.t - t0sim) - (t - t0real);
                if (ahead > 3000) {
                    try { sleep(ahead - 3000); }
                    catch (InterruptedException exc) { break; }
                }
            }
        }
    }
}

```



```

    public Slave(Robot robot, Robot2Host master) {
        (mover = new Mover(robot, master)).start();
        (receiver = new Receiver(mover, master)).start();
    }
}

```

Mottagartråden:

```

import se.lth.cs.realtime.*;

public class Receiver extends RTThread {

    Robot2Host master; // Connection to master.
    Mover mover;      // Motion performer.

    public Receiver(Mover mover, Robot2Host master) {
        this.mover = mover;  this.master = master;
    }

    public void run() {
        while (!interrupted()) {
            mover.putEvent(master.receiveNewStep());
        }
    }
}

```

Hantering av robotarmen:

```

import se.lth.cs.realtime.*;
import se.lth.cs.realtime.event.*;

public class Mover extends RTThread {

    Robot      arm;
    Robot2Host master;

    class StopEvent extends RTEvent {
        PosEvent posEv;
        StopEvent(PosEvent ev) {super(); posEv = ev;}
    }

    class StepBuffer extends RTEventBuffer {
        int nMoves; // Number of complete but unperformed motions.
        boolean moving; // Motion ongoing but all steps not in buffer.
        PosEvent last; // Last step posted so far.

        StepBuffer(int size) {
            super(size);
        }

        public synchronized void doPost(RTEvent ev) {
            int incSize = 256; //Mover.this.defaultEventBufferSize; // E.g.
            if (isFull()) setMaxSize(currentSize()+incSize);
            if (ev instanceof PosEvent) {
                PosEvent p = (PosEvent) ev;
                boolean zeroSpeed = p.x==last.x && p.y==last.y && p.z==last.z;
                if (zeroSpeed) {
                    moving = false;
                    nMoves++; // Now one if moving was true.
                }
            }
        }
    }
}

```

