

Tentamensskrivning i Realtidsprogrammering **1998-12-17, kl 8.00-13.00**

Anvisningar

Tillåtna hjälpmedel: inga utöver Java snabbreferens.

För godkänt betyg krävs att större delen av de 7 första uppgifterna samt en av uppgifterna 8 eller 9 behandlas nöjaktigt. För högre betyg krävs naturligtvis mer, så gör så många uppgifter du kan.

Senast 3:e arbetsdagen efter tentamen anslås på institutionens anslagstavla vilka som deltagit i tentamen men som, enligt institutionens noteringar, ännu inte redovisat någon av övningarna, laborationerna eller projektet. Deras skrivning rättas inte och annulleras den 6:e arbetsdagen efter tentamen, såvida inte rättelse skett dessförinnan eller dispens erhållits från ansvarig lärare.

Uppgifter

1. Vad är skillnaden beträffande villkor för korrekthet på programmering med jämtlöpande aktiviteter (concurrent programming) och realtidsprogrammering (real-time programming)? (Svara med högst två meningar.) (1 p)

2. Programkonstruktionen *semaphor* utgör en tidig och grundläggande primitiv för realtidsprogrammering.

a) Nämn två motiv till att semaforer är vanligt förekommande i industrin idag. (1 p)

b) Implementera en klass Semaphore i Java. (2 p)

3. Vilka fyra villkor måste vara uppfyllda för att dödläge (deadlock) skall kunna uppträda i ett system? Vilket av dessa krav är det som vi vid realtidsprogrammering försöker att inte uppfylla. Varför vill vi inte förbjuda att övriga villkor är uppfyllda? (3 p)

4. Förklara begreppen proiritetsinversion och prioritetärvning. Vilka är konsekvenserna i de båda fallen (inversion resp. ärvning) för högprioriterade trådars förmåga att uppfylla ställda tidskrav? (3 p)

5. Ett naivt sätt (som används av en del programmerare utan kunskaper i realtidsprogrammering) att vänta tills ett villkor är uppfyllt är följande programkonstruktion:

```
while (!flag) {};
```

där `flag` symboliserar villkoret som vi väntar på skall bli `true`.

a) Vad kallas detta sätt att vänta/testa? Nämn två nackdelar med denna konstruktion. (2 p)

b) Hur bör man göra istället? Varför? (Högst fyra meningar som svar.) (2 p)

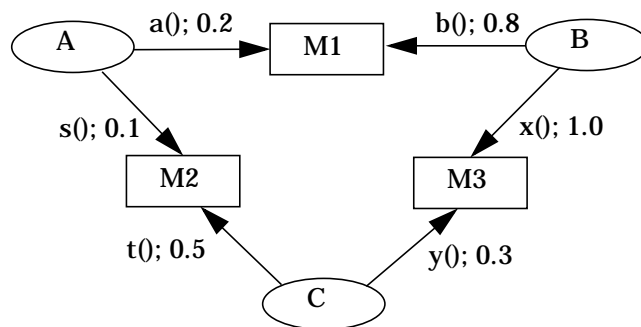
6. Du implementerar ett realtidssystem i vilket följande tre periodiska processer skall köras:

| Thread | Period | Exec. time | Priority |
|--------|--------|------------|----------|
| A | 8 | 4 | ? |
| B | 10 | 2 | ? |
| C | 16 | 3 | ? |

a) Sätt prioriteter på trådarna A, B och C enligt RMS (Rate-Monotonic Scheduling). (1 p)

b) Sätt prioriteterna enligt principen för DMS (Deadline-Monotonic Scheduling). Sätt maximal svarstid till dubbla exekveringstiden. (1 p)

c) Beräkna maximal blockeringstid för den mest högprioriterade tråden med de båda priortets-sättningarna (RMS och DMS enl. a och b) då trådarna A, B och C anropar monitoroperationer i monitorerna M1, M2 och M3 enligt följande: (2 p)



Det numeriska värdet efter varje metदानrop anger den maximala exekveringstiden.

7. Då man i en monitoroperation väntar på att ett villkor skall bli uppfyllt så vet den realtidskunnige att testet skall omges med `while`, d.v.s. man skriver

```
while (!ok) wait();
```

och inte

```
if (!ok) wait();
```

där `ok` skall bli sann för att operationen skall kunna slutföras. Ett triviale fall där detta har betydelse är då flera trådar väntar och då man i operationen som ändrar villkoret använder `notifyAll()` Beskriv (med kod och/eller ord) ett annat scenario (med tre trådar inblandade) där `while` är av avgörande betydelse även om endast `notify()` används. (2 p)

8 Exekvering av robotprogram

I robotstyrssystem finns en programtolk som tolkar instruktioner om vad roboten skall göra, och beroende på typ av instruktion och dess argument anropar avsedd funktion i styrssystemet. I följande mycket förenklade fall så kan roboten endast utföra två olika operationer:

- **move** flyttar robotarmen till en ny position enligt ett argument innehållande koordinaterna för rörelsens slutpunkt.
- **grasp** tar ett argument av typen `boolean`, och om detta är `true` griper robothanden. I annat fall öppnas den.

Låt oss bortse från hur instruktionerna skrivits och givits till programtolken, och istället titta på de metदानrop som tolken gör. Rena beräkningar kan utföras internt i tolk-objektet så det är bara de två rörelseinstruktionerna **move** och **grasp** som vi beaktar. Beordring av rörelser sker via klassen `RobotMotion` enligt följande:

```

public class RobotMotion {

    /**
     * Move the arm to the target position, one small step at a time according
     * to the perform method, and using velocities etc according to prepare.
     */
    public void move(Pose target) {
        Trajectory ref = prepare(target);
        perform(ref);
    }

    /**
     * Close gripper if arg is true, open otherwise.
     */
    public void grasp(boolean on) {
        // Control digital output according to argument...
    }

    //----- Methods not called by interpreter:

    /**
     * Compute motion data according to dynamic properties.
     */
    public Trajectory prepare(Pose goalPosition) {
        Trajectory traj = new Trajectory();
        // Extensive computations go here....
        return traj;
    }

    /**
     * Perform motion by sending motion data to servo control system.
     */
    public void perform(Trajectory setpoint) {
        // Set servo reference each 10 ms until motion completed...
    }
}

```

där följande abstrakta datatyper (färdiga klasser vars innehåll vi inte behöver känna till) används:

Pose: Beskriver rörelsens slutpunkt (position och handorientering).
Trajectory: Beskriver varje robotleds rörelse som funktion av tiden från start till slut.

Programtolken som utför robotinstruktionerna anropar således metoderna `grasp` och `move`, och `move` i sin tur använder metoderna `prepare` och `perform` för att utföra en rörelse. De senare metoderna har egenskaperna:

prepare: Beräknar rörelsens tidsförlopp utifrån dynamiska modeller etc. För denna uppgift behöver vi (endast) veta att i vissa fall (beroende på begärd rörelse och grad av optimering) innebär anrop av denna metod *mycket beräkningar men ingen blockering*.

perform: Använder det beräknade tidsförloppet genom att periodiskt (säg var 10 ms) evaluera önskad vinkel för varje robotled och skicka detta som referensvärde till reglersystemet. Denna metod medför *mycket lite CPU-tid men pågår under lång tid* (pga blockering varje period tills den mekaniska rörelsen utförts).

I ovan beskrivna implementering anropas således `prepare` och `perform` direkt av tolken, via metoden `move`. Fördelen med detta är att man i tolken hela tiden vet vilken rörelse som håller

på att utföras och när denna är klar. Därmed kan också `grasp` anropas direkt av tolken. Nackdelen är att då en rörelse utförts så tar `prepare`-anropet för nästa rörelse för lång tid, vilket medför att roboten stannar upp mellan varje programmerad punkt, vilket inte är acceptabelt.

Din uppgift är nu att förbättra styrsystemet enligt följande specifikationer:

- Tolken och dess anrop ändras ej; fortfarande anropas `move` och `grasp`. Istället skall implementeringen av klassen `RobotMotion` göras om.
- Tolkens **anrop av `move` och den då anropade `prepare` skall kunna utföras innan föregående rörelse (d.v.s. anropet av `perform`) är klar**. Detta innebär buffring av de genererade trajektorierna vilket då ger en jämnare CPU-belastning och roboten behöver normalt sett **inte stanna upp i väntan på `prepare`-anropet**.
- Upp till 16 rörelser skall kunna förberäknas och buffras.
- P.g.a. tidskrav och egenskaperna hos det underliggande operativsystemet så är det inte tillåtet att skapa en ny tråd för varje rörelse. Gör en normal design med fast antal trådar.
- Systemet skall förberedas för att exekvering av `prepare` skall kunna ske på en annan dator än den som exekverar `perform`. Detta innebär att buffring av beräknade trajektorier göres meddelandebaserat med klassen `RTEvent` som basklass.
- Observera att om instruktionerna till roboten t.ex. är sekvensen `move - grasp - move` så skall `grasp` utföras när det först `move` utförts (d.v.s. när `perform` för denna är klar) och inte när tolken beordrat rörelsen. Detta för att grip/släpp skall utföras på rätt ställe.

Ur lösningen skall det klart framgå:

1. Vilka klasser ingår och eventuella arv?
2. Vilka object/klasser utgör monitorer och vilka utgör aktiva objekt (trådar)?
3. Hur interagerar de olika trådarna?

Implementera den nya versionen av klassen `RobotMotion` samt eventuella ytterligare klasser. Klassen `RTEventQueue.java` bifogas. (10 p)

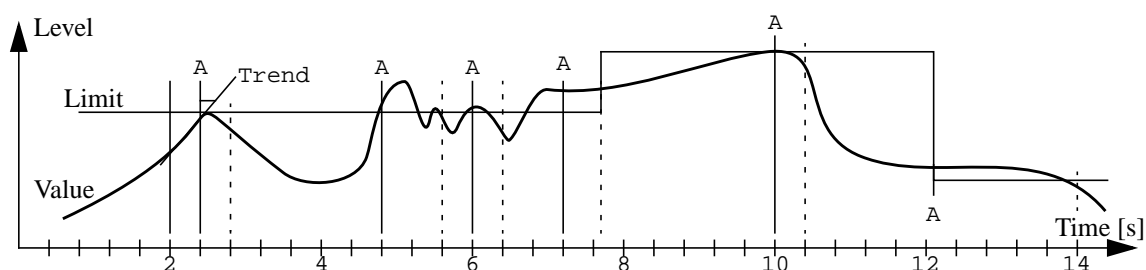
9. Alarmhantering

I ett så kallat SCADA-system (Supervisory Control And Data Acquisition) skall ingå en larm-detektor med inbyggd prediktering enligt följande specifikationer:

1. Mätvärden större än en viss gräns (Limit i fig nedan) innebär att larm skall ges.
2. Alarmdetektorn skall implementeras i form av en monitor-klass `AlarmMonitor` enligt nedan.
3. Mätvärden (Value i fig nedan) erhålls genom att avläsa attributet `filteredInput` (av typen `float`) i ett objekt av typen `Sample` som tillhandahålls som argument till konstruktorn för klassen `AlarmMonitor`.
4. Mätvärdena avläses periodiskt, vilket hanteras utanför monitorn som ju inte är någon tråd. Antag samplingsperioden 0.4 sek.
5. Ett troligt nära förestående larm skall detekteras genom att en trend beräknas med enkel numerisk derivering vid varje sampel. Om aktuellt mätvärde plus trenden gånger en tidshorisont överstiger larmgränsen skall larm ges. Antag tidshorisonten 0.2 sek.
6. Aktuell larmgräns har initialvärdet 0.0 och sätts via metoden `setLimit`.
7. Användare av alarmdetektorn, dvs de trådar i systemet som vill vänta på att ett larm skall inträffa, anropar metoden `awaitAlarm`. Om du behöver dess argument `lastAlarm` och/eller dess returvärde så förklara hur.
8. Varje anropande tråd får larmas högst en gång per sekund. Observera att denna sekund gäller från det att just den tråden larmades förra gången.
9. Efter att larm detekterats så skall nya värden över larmgränsen inte ge upphov till nya larm om inte ett mätvärde under larmgränsen och med negativ trend först avlästs.
10. Flera trådar skall samtidigt kunna vänta på larm. Väntan innebär blockering.

11. Då gränsvärdet (Limit) ändras kan den nya gränsen ligga under aktuellt mätvärde. De trådar som inte larmats senaste sekunden skall då larmas. Observera att detta sker synkront med ändringen av gränsen (utan att vänta tills nästa sampel).
12. Larmdetekteringen, inklusive avläsningen av mätvärden och aktuell tid, kan fördröjas av mera högprioriterade aktiviteter. För att detta inte skall ge en felaktig trend skall avläsningen av mätvärde ske inom en 10 ms period och vid beräkning av trenden tas hänsyn till vid vilka tidpunkter de två ingående värdena avlästs. (Du skall inte sätta några prioriteter i denna uppgift, så ett mätvärde kan behöva läsas flera gånger innan det erhållits vid en välbestämd tidpunkt.)

Följande figur illustrerar önskad funktionalitet. Markeringarna på tidskalan markerar samplingen som sker med periodtiden 0.4 sek. Varje detekterat larm anges med ett A. I figuren har antagits att endast en tråd anropar och att denna tråd efter alarm åter anropar `awaitAlarm` inom en sekund. Det första larmet ges trots att gränsvärdet aldrig passerats. Vid tiden 10 tangeras gränsvärdet och den positiva trenden (ej utritad) ger då larm. Det sista larmet ges i samband med ändringen av gränsvärdet. Streckade linjer anger att larmtillstånd inte längre gäller vilket innebär att nya larm kan ges.



Eventuella trådar som behövs för att uppfylla specifikationerna skall implementeras, men gränssnittet till detektorn sett från användaren skall utgöras av följande klass:

```
public class AlarmMonitor {

    /**
     * Blocks caller until alarm occurs.
     */
    synchronized public long awaitAlarm(long lastAlarm)
        throws InterruptedException {
        // Write this code ...
    }

    /**
     * Change the alarm limit. Awaiting threads are notified if the
     * new limit implies an alarm condition.
     */
    synchronized public void setLimit(float newLimit) {
        // Implement it!!
    }

    AlarmMonitor(Sample sample) {
        // The constructor ....
    }

    synchronized public void start() {
        // Start alarm monitoring here.
    }

}
```

(10 p)