Hemtentamen, EDAF85 Realtidssystem

2020–10–30, 08.00–13.00

Tillåtna hjälpmedel för hemtentamen

- dator, med Eclipse och Java-kompilator (även annan utvecklingsmiljö är tillåten)
- kurslitteraturen
- Internet, för att slå upp detaljer vid behov
  (som dokumentation för Javas standardklasser, **inte** för kommunikation med utomstående!)

Du ska lösa hemtentan själv. **Du får inte kommunicera med någon annan än kursens lärare medan du löser hemtentan.** De lösningar du lämnar in ska vara dina egna och de texter du lämnar in ska vara skrivna av dig själv och inte direkt hämtade från någon annanstans. Brott mot detta kommer att betraktas som fusk och anmälas till universitetets disciplinnämnd. Inlämnade lösningar kommer att jämföras med varandra samt matchas mot texter på Internet med hjälp av automatiska verktyg.

Instruktioner

Denna hemtentamen består dels av ett antal teorifrågor (fråga 1–3) där du i kort essäform ska redogöra för utvalda teoretiska frågeställningar samt redogöra för hur man löser – och lösa – något räkneproblem och dels av en programmeringsuppgift (uppgift 4). Poängbedömningen för essäfrågorna kommer att grunda sig på hur pass väl och fullständigt du redogör för den aktuella frågeställningen. Som stöd kommer du att få ett antal nyckelord/frågeställningar som bör ingå/besvaras. Programmeringsuppgiften kommer att bedömas på liknande sätt som vid vanliga skriftliga tentor, dvs vi kommer att lägga mer vikt vid att bedöma huruvida din lösning uppvisar en lämplig tråddesign, synkroniserar trådarna korrekt och uppfyller tidskraven än att koden är helt korrekt och går att kompilera och köra. Det är alltså fullt tänkbart att en lösning kan få full poäng fast den inte går att kompilera och köra likväl som att den kan bli underkänd fast den till synes verkar exekvera felfritt. Preliminär gräns för godkänt betyg är 15 poäng (av totalt 30) med åtminstone 7 poäng på teoridelen (av 15) och 7 poäng på programmeringsdelen (av 15). Svar kan ges på svenska eller engelska.

Lämna in lösningar även om du inte kan lösa uppgifterna helt. Poäng utdelas även för delvis lösta uppgifter. Fastna inte för länge på problem med att få programmet att kompilera och köra – se ovan.

Textkommentarer kan skrivas i Java-koden på vanligt sätt om du vill förtydliga vad du gör.

Inlämning via mail – senast kl 13.00

Du lämnar in din lösning genom att svara på det mail du fick och som innehöll denna hemtentamen. Bifoga dels dina lösningar på teoridelen i form av en vanlig textfil, PDF-fil eller ett Word-dokument och dels javafilen med dina tillägg. Svaret ska vara tidsstämplat senast 13.00.

VIKTIGT: Skriv ditt namn i varje fil du lämnar in! I javafilen finns en ruta längst upp för detta.

Frågor under tentans gång

Frågor om tentamen besvaras under tentans gång av kursansvarig på Zoom. Adressen till zoommötet är: https://lu-se.zoom.us/j/68943732085. Länk till mötet hittar du även på kurshemsidan. Skulle Zoom av någon anledning inte fungera så går det bra att ringa 046-2229635.

1. Dödläge

Redogör för hur dödläge kan uppstå i ett flertrådat program där trådarna använder semaforer för att åstadkomma ömsesidig uteslutning sinsemellan samt hur detta kan undvikas.

Din beskrivning ska innehålla svar på följande frågeställningar:

- Vad innebär dödläge?
- Vilka villkor måste generellt vara uppfyllda för att dödläge ska kunna uppstå?
- Vilka av dessa villkor kan vi påverka när vi skriver våra program? Motivera – varför är det något vi kan påverka och varför kan/vill vi inte påverka de andra?

(3p)

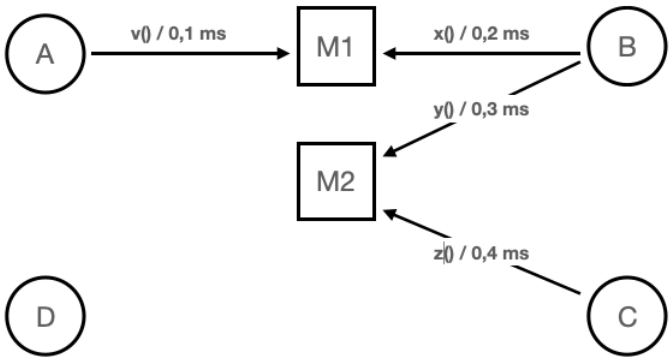
2. Semaforer

Under kursen har vi använt semaforer till två olika saker, nämligen för att åstadkomma ömsesidig uteslutning och för signalering. Redogör för vad som skiljer sig åt vid användningen av en semafor i dessa två fall samt illustrera det med ett enkelt exempel i (pseudo-)kod.

(3p)

3. Generaliserad schemaläggningsanalys

Antag att vi har ett system som schemaläggs enligt Rate Monotonic Scheduling, använder sig av dynamiskt prioritetsarv, och består av fyra trådar kallade A, B, C och D. Trådarna använder sig av två monitorer, kallade M1 och M2, för sin inbördes synkronisering enligt nedanstående figur där också respektive monitoroperations maximala exekveringstid är angiven. Som framgår av figuren är en av trådarna helt fristående från de andra och använder sig inte av någon av monitorerna och exekverar alltså helt oberoende av de andra trådarna. Trådarna anropar bara en av monitoroperationerna i taget (dvs att monitoranropen ej är nästlade).



Trådarnas värstafallsexekveringstider (C), periodtider (T) och deadlines (D) ges av följande tabell:

Tråd	*C (ms)*	*T (ms)*	*D (ms)*
A	3	15	10
B	1	5	2
C	4	20	15
D	2	8	4

För att beräkna värstafallssvarstiderna (responstiderna) i ett sådant system har vi använt oss av följande formel:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

a) Redogör, med egna ord, kortfattat för vad formeln betyder. Din beskrivning ska innehålla svar på följande frågeställningar:

- Formelns högerled består av tre termer. Vad representerar de tre olika termerna?
- Hur är det möjligt att beräkna formelns värde trots att R_i förekommer i både vänster- och högerledet?

(2p)

b) Räkna ut B_i för respektive tråd. För full poäng måste du kort motivera varför respektive tråd får det angivna värdet på B_i. *(3p)*

c) Räkna nu ut R_i för respektive tråd. För full poäng måste du redovisa alla dina beräkningar.

Takfunktionen (avrundning uppåt till närmsta större heltal) är ju naturligtvis besvärlig att skriva på tangentbordet. Använd i så fall följande notation[1]: $ceil(x)$ för att ange att du vill avrunda x uppåt. I stället för att t.ex. skriva $\lceil \frac{3}{7} \rceil$ kan du skriva $ceil(3/7)$. *(4p)*

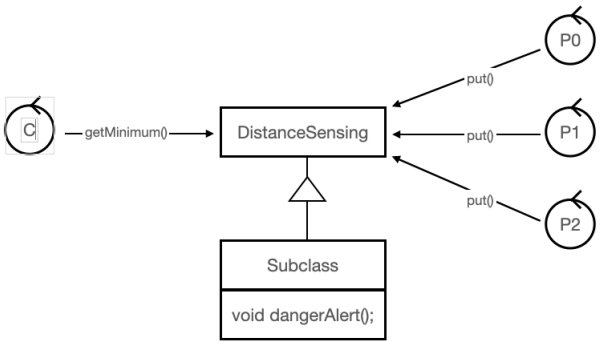
[1] Förkortning av ceiling – tak.

4. Filtrering av sensorvärden

I övervakningssystemet till en mobil robot ingår tre olika sensorer som mäter avståndet till omgivande föremål i olika riktningar. Sensorerna läses av av tre trådar kallade P0, P1 respektive P2. Beteckningarna matchar sensorernas numrering (0, 1 och 2). När sensortrådarna har läst av ett nytt värde (avstånd) lagras detta värde i en monitor, `DistanceSensing`, vilken det är din uppgift att implementera. Trådarna anropar metoden `put()` för att lagra ett nytt värde i monitorn. Utöver de tre sensortrådarna finns det en kontrolltråd, C, som regelbundet hämtar det minsta avståndet till omgivande föremål genom att anropa metoden `getMinimum()`. Att skriva de fyra trådarna ingår inte i uppgiften – bara att skriva monitorn.

Javaklassen som representerar monitorn, `DistanceSensing`, är en abstrakt klass med den abstrakta metoden `dangerAlert()`. Denna metod ska anropas automatiskt när något av mätvärdena från sensorerna kommer under en viss gräns (`DANGER_DIST`) och är tänkt att kunna användas till exempel för att varna om roboten håller på att kollidera med något föremål. Rent praktiskt får man alltså då skapa en subklass till `DistanceSensing` som implementerar `dangerAlert()` och i vilken vi kan ta hand om den farliga situationen. Att skriva subklassen ingår dock inte i uppgiften. Det gör däremot att se till metoden `dangerAlert()` faktiskt anropas när den ska.

Nedanstående figur sammanfattar relationen mellan trådarna, monitorn och subklassen som implementerar `dangerAlert()`.



Varje enskilt mätvärde, eller sampel, representeras av ett objekt av klassen `Sample`. Sådana objekt är tänkta att vara oföränderliga (immutable), så det går bra att dela sådana objekt mellan olika trådar utan att göra kopior på vägen. Objekten får automatiskt en tidsstämpel som talar om när de skapades.

```
class Sample {
    private int id;
    private double value;
    private long timestamp;

    /** Constructor */
    public Sample(int id, double value) {
        this.id = id;
        this.value = value;
        timestamp = System.currentTimeMillis();
    }

    /** Returns the id (0,1,2) of the sensor */
    public int getId() { return id; }

    /** Returns the sampled (distance) value. */
    public double getDistance() { return value; }

    /** Returns the sample timestamp. */
    public long getTimestamp() { return timestamp; }
}
```

Din uppgift är att implementera själva monitorklassen `DistanceSensing`:

```
abstract class DistanceSensing {

    private static final double DANGER_DIST = 0.8;

    /** Constructor */
    public DistanceSensing() {
        /* to be implemented */
    }

    /** Stores a new sensor value in the monitor.
        This method never blocks. */
    public  void put(Sample s) {
        /* to be implemented */
    }

    /** Blocks until sensor values at least as new as time
        are available from all sensors anf then returns
        the sample with smallest value.
    */
    public  Sample getMinimum(long time) {
        /* to be implemented */
        return null;
    }

    /** Automatically called whenever it is detected that
        any of the current sensor values are lower than
        DANGER_DIST. Only one invokation of this method can
        be active at any time, i.e., if the method is called
        it can not be called again until the first call has
        finished. A new call also requires that all sensor
        values returns to a safe state (> DANGER_DIST)
        before it can be called again. A call to the method
        is allowed to take a long time, e.g., waiting for an
        operator to acknowledge an alarm.
    */
    protected abstract void dangerAlert();

}
```

Beskrivning av monitoroperationerna

`public void put(Sample s)` – Varje sensortråd producerar objekt av klassen `Sample` och anropar `put()` med dessa sampel. Monitorn ska bara lagra det allra senaste sampelobjektet för varje sensor. Du behöver alltså hålla ordning på tre olika mätvärden i monitorn. Om `put()` skulle anropas flera gånger i rad för en och samma sensor utan att det gamla värdet använts skrivs föregående värde helt enkelt över. Denna metod får aldrig blockera utan ska returnera utan onödigt dröjsmål.

`public Sample getMinimum(long time)` – Denna metod blockerar tills det finns mätvärden tillgängliga från alla tre sensorerna sådana att deras tidstämplar är lika med, eller senare, än tiden angiven i parametern `time`. När tre sådan mätvärden finns tillgängliga returneras det sampelobjekt som innehåller det *minsta* värdet på det uppmätta avståndet (anropa `getDistance()`) utan vidare fördröjning.

`protected abstract void dangerAlert()` – Denna metod ska du *inte* skriva. Meningen är ju att den som använder din monitorklass ska subklassa den och i subklassen implementera metoden. Däremot ska du se till att den anropas vid rätt tillfälle. Denna metod ska automatiskt anropas efter att ett mätvärde som är mindre än `DANGER_DIST` lagrats i monitorn genom ett anrop av `put()`. Tänk dock på att ett anrop av `dangerAlert()` mycket väl skulle kunna ta lång tid (i testprogrammet

som bifogas uppgiften kommer den t.ex. att vänta på att en operatör bekräftar varningen) vilket står i motsatsförhållande till tidskraven för metoderna `put()` och `getMinimum()` ovan. I övrigt ska metoden fungera så att bara ett anrop av metoden ska kunna vara aktivt åt gången. Snabbt inkommande små mätvärden ska alltså inte kunna göra att flera anrop av metoden kör parallellt. När metoden har kört färdigt ska dessutom inget nytt anrop göras förrän mätvärdena från alla tre sensorer åter varit uppe på säkra nivåer däremellan. Ett anrop av denna metod får inte heller hindra någon av de fyra trådarna P0, P1, P2 och C att under tiden fortsätta stoppa in respektive hämta ut nya mätvärden.

Du får ändra på de givna deklarationerna av metoderna i monitorn `DistanceSensing`, men inte så att anropssignaturen ändras. Dvs att metoderna måste fortfarande heta samma saker, acceptera samma argument samt returnera samma värden som tidigare. Du får dessutom lägga till egna privata attribut och metoder samt inre klasser efter behov. Det går också bra att lägga egna hjälpklasser utanför klassen `DistanceSensing` om så önskas.

Praktisk implementation

Tillsammans med denna tenta ska du ha fått en fil `SensorFilter.java`. Skriv in din implementation av `DistanceSensing` tillsammans med eventuella hjälpklasser du behöver för din lösning i filen på angiven plats. Har du gjort rätt ska du sedan kunna kompilera och köra din kod med ditt favoritverktyg för javautveckling. Längst ner i filen finns ett mycket enkelt testprogram bestående av ett huvudprogram och lite hjälptrådar som gör att du kan provköra din kod – se kodkommentarerna för förklaring. Tänk på att testprogrammet inte testar alla aspekter av ditt program. Det är till exempel mycket svårt att från körningen avgöra om det finns risk för kapplöpningar någonstans i koden.

När du kör programmet kommer tre simulerade sensortrådar med lite slumpmässiga intervall skicka in simulerade mätvärden till din monitor. En simulerad kontrolltråd kommer att att hämta ut och skriva ut aktuella minimum-värden efter hand som de blir tillgängliga. Om mätvärden mindre än `DANGER_DIST` (0,8) genereras är det meningen att du ska se till att `dangerAlert()` anropas. Testprogrammet innehåller en implementation av denna som skriver ut en varning samt väntar på att du ska bekräfta varningen genom att trycka på Enter på tangentbordet. Först därefter ska nya varningar kunna genereras (men nya mätvärden ska fortsätta att strömma in och skrivas ut efter hand som de blir tillgängliga). Studera gärna testprogrammet för att få en bättre uppfattning om hur det är meningen att din monitor ska anropas, men *ändra inte i den givna koden för simulatorn*!

Bifoga den ifyllda filen `SensorFilter.java` när du skickar in ditt svar på tentan. Glöm inte att skriva ditt namn på avsedd plats överst i den inlämnade filen.

(15p)