Tentamen
EDAF85 – Realtidssystem

2023-04–20, 08.00–13.00

Det är tillåtet att använda Java snabbreferens och miniräknare. Det går bra att använda både engelska och svenska uttryck för svaren.

Den här tentamen består av två delar: *teori*, fråga 1-6 (15 poäng), och *programmeringsuppgifter*, fråga 7-8 (15 poäng). För godkänd (betyg 3) krävs preliminärt sammanlagt hälften av alla 30 möjliga poäng samt en tredjedel av poängen (5) på varje del för sig.

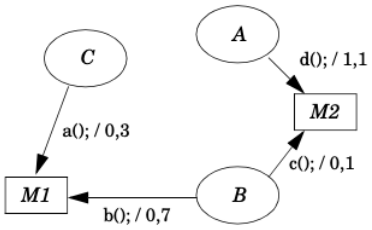
Formelsamling

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

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

1. Vad är prioritetsinversion? Illustrera med ett litet exempel. Beskriv kortfattat en mekanism som kan användas för att undvika att prioritetsinversion kan uppkomma. *(2p)*

2. Ett realtidssystem (med dynamisk prioritetsbaserad schemaläggning och prioritetsarv) innehåller tre trådar (A, B och C) som kommunicerar med varandra genom att anropa monitoroperationerna a, b, c, och d i monitorerna M1 och M2 och med maximala exekveringstider (i millisekunder) enligt figur. Trådarna anropar en monitoroperation i taget.



Vidare har trådarna värstafallsexekveringstider (C) och periodtider (T) enligt följande tabell. Deadline är lika med periodtiden för varje tråd.

Tråd	*C (ms)*	*T (ms)*
A	4	25
B	8	40
C	3	10

För att tilldela trådarna prioriteter används principen RMS - Rate monotonic Scheduling.

a) Ange för varje tråd (A, B och C) hur lång tid tråden i värsta fall kan bli blockerad av lägre prioriterade trådar under en och samma körning. *(2p)*

b) Ange för varje tråd (A, B och C) vad deras respektive värstafallsresponstid blir. *(2p)*

3. I ett Javaprogram hittar vi två typer (klasser) av trådar, T1 och T2, som i sina respektive run()-metoder exekverar följande linjära sekvenser av semaforoperationer på de fem mutexsemaforerna A, B, C, D och E (mellanliggande kod som är beroende av semaforerna representeras av funktionsanropen på formen ”useXY();”, där ”XY” avser att semaforerna X och Y måste vara tagna när koden utförs):

```
T1                  T2
A.take();           C.take();
B.take();           B.take();
D.take()            useBC();
useABD();           B.give();
A.give();           A.take();
E.take();           useAC();
useBDE();           A.give();
E.give();           C.give();
D.give();           ...
B.give();           E.take();
                    C.take();
                    useCE();
                    C.give();
                    E.give();
```

Observera att vi kan ha ett godtyckligt antal instanser av varje trådtyp (klass).

a) Rita en resursallokeringsgraf för systemet. *(2p)*

b) Hur många instanser (trådobjekt) måste det minst finnas av vardera T1 och T2 för att det ska finnas risk för dödläge i systemet? *(1p)*

4. Ordet preemption förekommer i två olika sammanhang i kursen. Dels som beteckning för påtvingad tidsdelning, och del i samband med villkoren för dödläge. Förklara de två olika betydelserna och motivera varför man vill ha eller inte ha respektive egenskap hos sin programexekvering i ett realtidssystem. *(2p)*

5. En javaklass som implementerar komplexa tal har två attribut och fyra metoder enligt följande:

```
class Complex {
   float re;
   float im;
   synchronized void setValue(float re, float im) {
       this.re = re; this.im = im;
   }
   float getRe() { return re; }
   float getIm() { return im; }
   float getAbs() { return Math.sqrt(re*re+im*im); }
}
```

Vad innebär det egentligen att nyckelordet `synchronized` används i deklarationen av metoden `setValue`? Anta att vi har två trådar T1 och T2 som båda har referenser till två objekt av klassen `Complex`, C1 och C2. Svara genom att för varje påstående nedan ange om det är sant eller falskt.

1. När T1 exekverar `C1.setValue(1.0,1.1)` so kommer ett anrop i T2 av `C2.setValue(2.0,2.2)` blockera tills T1 har avslutat anropet av `C1.setValue(1.0,1.1)`.
2. När T1 exekverar `C1.setValue(1.0,1.1)` so kommer ett anrop i T2 av `C1.setValue(2.0,2.2)` blockera tills T1 har avslutat anropet av `C1.setValue(1.0,1.1)`.
3. Eftersom `setValue` är `synchronized` så kan T2 inte exekvera metoden `getAbs` i samma objekt så länge T1 exekverar `setValue`.
4. Eftersom en variabel av typen `float` kan tilldelas/returneras atomärt (dvs utan risk för ett kontextbyte) så kan vi (som en sorts optimering, dock ej rekommenderat, och så som det är gjort i koden ovan) utelämna nyckelordet `synchronized` i deklarationen av `getAbs` utan att förändra realtidskorrektheten hos programmet.

(2p)

6. Förklara kortfattat (med en eller några få meningar) följande begrepp inom schemaläggning för realtidssystem:

a) Kontextbyte

b) Statisk schemaläggning

c) Schemaläggning enligt principen Rate Monotonic Scheduleing

d) Schemaläggning enligt principen Earliest Deadline First

(2p)

7. Betrakta ett affärssystem i vilket kunder kan samla bonuspoäng (dessa kan t.ex. erhållas för tidigare köp och användas för att få rabatt på kommande köp). Bonuspoängen för en kund hanteras av ett objekt av typen `CustomerBonus`. Affärssystemet är multitrådat och ett `CustomerBonus`-objekt kan delas mellan flera trådar. `CustomerBonus` implementeras med hjälp av en monitor enligt nedanstående kod. Ibland händer det dock att systemet låser sig genom att två trådar som använder sig av samma `CustomerBonus`-objekt stannar upp och aldrig kommer vidare. Systemet är sedan låst tills det startas om.

Ändra koden för `CustomerBonus` så att ovanstående problem inte kan inträffa!

- Du behöver bara redovisa de delar av klassen som du ändrar i. Metoder, attribut och andra deklarationer som du inte redovisar kommer antas vara oförändrade vid rättningen av uppgiften.
- Om `addBonus()` och `compareTo()` anropas samtidigt på samma objekt ska resultatet bero på *antingen* det gamla eller det nya värdet för bonusen, men det har ingen betydelse vilket.

```
public class CustomerBonus implements Comparable<CustomerBonus> {
    private final String name;
    private double bonus = 0;

    public CustomerBonus(String name) {
        this.name = name;
    }

    public synchronized void addBonus(double b) {
        bonus += b;
    }

    public synchronized double getBonus() {
        return bonus;
    }

    @Override
    public synchronized int compareTo(CustomerBonus o) {
        int r = Double.compare(bonus, o.getBonus());
        if (r == 0) {
            // customers with same bonus
            // are ordered by name
            return name.compareTo(o.name);
        } else {
            return r;
        }
    }
}
```

(3p)

8. I många sammanhang kan det vara användbart att införa någon typ av övervakning av att trådarna i ett realtidssystem faktiskt utför sina uppgifter i tid. Det kan till exempel användas i debugsyfte för att upptäcka att ett dödläge uppstått, att systemet är överlastat eller att någon tråd kraschat av någon anledning och behöver startas om. En teknik som kan användas och som lämpar sig särskilt bra för periodiska trådar är att införa en så kallad *watchdog*. En sådan kan implementeras i mjukvara genom att låta varje tråd som ska övervakas regelbundet anropa en bestämd metod som ett sätt att signalera att tråden fortfarande lever och fungerar som den ska. Om metoden inte åter anropas inom en given tidsperiod (som kan variera från tråd till tråd) inträffar en timeout och lämplig åtgärd kan vidtas[1].

Nedan följer en ofullständig implementation av en klass `LivenessTracker` som implementerar en sådan watchdog. Trådar som ska övervakas anropar regelbundet metoden `reportActivity()` och anger varje gång en maxtid inom vilken den kommer att anropa metoden på nytt. Om den inte gör det inom den specificerade tiden ska metoden `timeoutAlert()` i automatiskt anropas, vilken i sin tur vidtar lämplig åtgärd (beroende på vad vi använder vår watchdog till).

```
public abstract class LivenessTracker {
    public LivenessTracker() {
        ..
    }

    /** Called by threads to request a timeout after timeout milliseconds unless
        this method is called again. If timeout=0 no timeout will be generated.
    */
    public void reportActivity(long timeout) {
        ...
    }

    /** Automatically called when a timeout occurs for thread t.
        Implement the method in a subclass in order to describe
        what should happen at a timeout event.
    */
    protected abstract void timeoutAlert(Thread t);
}
```

Varje gång en timeout inträffar ska metoden `timeoutAlert()` anropas. Denna är deklarerad `abstract` vilket betyder att vi måste implementera metoden i en subklass för att beskriva vad som ska hända när timeout sker – jämför med designmönstret Template Method från kursen i objektorienterad modellering och design. Man skulle till exempel kunna tänka sig att logga någon sorts varning eller försöka starta om tråden som inte svarat i tid. För att skapa en LivenessTracker som bara genererar en varningsutskrift på standard output varje gång en timeout inträffar skulle vi som exempel kunna skriva (detta är bara ett exempel – att implementera en sådan subklass ingår *inte* i uppgiften):

```
public class MessageLoggingTracker extends LivenessTracker {
    protected void timeoutAlert(Thread t) {
        System.out.println("Warning! Thread timed out: "+t.toString());
    }
}
```

[1] En watchdog kan även implementeras i hårdvara genom att man har en digital räknare som räknas ned varje gång en klockpuls kommer och om den kommer till noll innan räknarens värde återställs av det körande programmet så genereras en resetsignal. På så sätt kan en låst dator automatiskt återstartas om den låst sig.

Implementera klassen `LivenessTracker` ovan enligt följande anvisningar!

1. Använd Javas inbyggda monitorbegrepp för all synkronisering/signalering som behövs mellan aktuella trådar.
2. Du får lägga till privata attribut/klasser samt privata metoder efter behov.
3. Du får ändra på deklarationen av metoden reportActivity, men den ska fortfarande anropas enligt specifikationen ovan, dvs den ska bara ta en timeouttid som parameter och den ska inte returnera något värde.
4. Behöver du någon extra hjälptråd kan du lämpligen starta denna i klassens konstruktor.
5. Av effektivitetsskäl får du inte starta en ny tråd varje gång en timeout begärts.
6. Innehåller din lösning en tom konstruktor kan du utelämna den.
7. Metoden `timeoutAlert` ska anropas utan onödig fördröjning när en timeout inträffar. Lösningen ska därför inte bygga på att med jämna intervall kontrollera om någon timeout inträffar.
8. Avsikten är att det ska räcka med ett enda objekt av klassen `LivenessTracker` oavsett hur många trådar det är som ska bevakas (för att minimera antalet extra trådar). Det innebär att objektet behöver hålla reda på separata timeouttider för de olika trådarna. Om du vill förenkla uppgiften – mot poängavdrag, se nedan – kan du nöja dig med att anta att varje tråd som ska bevakas använder sig av sitt eget `LivenessTracker`-objekt. Då behöver ett `LivenessTracker`-objekt bara hålla reda på en tråd och tillhörande timeouttid.

Tips (som kan vara relevanta eller inte beroende på vilken problemvariant du väljer):

- I `reportActivity` behöver du säkert veta vilken tråd som anropat metoden. Det kan du få veta genom att anropa funktionen `Thread.currentThread()`.
- I Java får man inte deklarera abstrakta metoder `synchronized` (men den implementerande metoden i subklassen kan vara `synchronized`).
- Ett sätt att hålla reda kommande timeouter är att använda en prioritetskö. Dokumentation för klassen `PriorityQueue` samt andra klasser/interface som möjligen kan vara användbara bifogas.
- Om du vill kan du anta att följande klass som representerar en tråd och dess timeouttid finns tillgänglig (utan att du behöver skriva/deklarera/importera den):

```
public class Timeout implements Comparable<Timeout> {
    public final Thread thread;
    public final long timeout;
    public Timeout(Thread thread, long timeout) {
        this.thread = thread;
        this.timeout = timeout;
    }
    public int compareTo(Timeout other) {
        return Long.compare(this.timeout, other.timeout);
    }
}
```

Poängbedömning

En lösning där ett och samma `LivenessTracker`-objekt kan hålla reda på flera trådar och dess timeouttider ger maximalt 12 poäng. En lösning på det förenklade problemet där varje `LivenessTracker`-objekt bara kan hålla reda på en tråds timeout ger maximalt 8 poäng.

(8p/12p)