Tentamen
EDAF85 – Realtidssystem

2025-10-24, 08.00–13.00

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

Den här tentamen består av två delar: *teori*, fråga 1-5 (15 poäng), och *programmeringsuppgifter*, fråga 6 och 7 (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

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

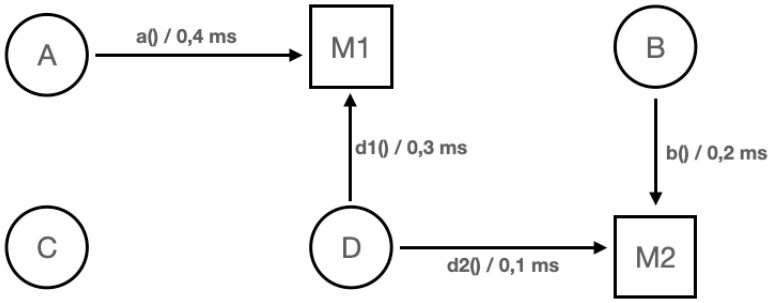
$$\begin{gathered} 2 \cdot (2^{1/2} - 1) \approx 0,828427 \\ 3 \cdot (2^{1/3} - 1) \approx 0,779763 \\ 4 \cdot (2^{1/4} - 1) \approx 0,756828 \\ 5 \cdot (2^{1/5} - 1) \approx 0,743492 \\ \ldots \\ \lim_{n \to \infty} n(2^{1/n} - 1) \approx 0,693147 \end{gathered}$$

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

1. Beskriv kortfattat skillnaden mellan schemaläggning enligt principen Earliest Deadline First och schemaläggning enligt principine Deadline Monotonic Scheduling.

(1p)

2. Ett realtidssystem (med dynamisk prioritetsbaserad schemaläggning, prioritetsarv och en enda CPU-kärna) består av fyra periodiska trådar (A, B, C och D) som kommunicerar med varandra genom att anropa monitoroperationerna `a()`, `b()`, `d1()`, och `d2()` i monitorerna M1 och M2 och med maximala exekveringstider för operationerna (i millisekunder) enligt figur. Tråd C anropar inga monitoroperationer. Trådarna anropar en monitoroperation i taget, dvs att anropen inte är nästlade.



Prioritetsordningen för trådarna är (från högst till lägst) A, B, C, D.

Ange för var och en av trådarna hur länge de maximalt kan bli fördröjda av *lägre prioriterade* trådar, dvs deras blockeringsfaktorer.

(3p)

3. Ett realtidssystem (med dynamisk prioritetsbaserad schemaläggning, prioritetsarv och en enda CPU-kärna) består av fyra periodiska trådar – A, B, C och D. Deras respektive värstafallsexekveringstid (C), periodtid (T) och blockeringsfaktorer (B – den längsta tid de kan bli blockerade av lägre prioriterade trådar) ges av nedanstående tabell. Trådarnas deadlines förutsätts vara lika med deras periodtider och Rate Monotonic Scheduling används för att schemalägga trådarna.

	C (ms)	*T (ms)*	*B (ms)*
A	4	15	0,3
B	5	30	0
C	2	10	0,5
D	2	8	0,2

Ange för var och en av de fyra trådarna vad deras värstafallssvarstid (värstafallsresponstid) blir.

(4p)

4. Roger har skrivit nedanstående program som använder sig av fyra `ReentrantLock()` för att synkronisera de tre trådarna T1, T2 och T3. Tyvärr hamnar programmet i dödläge.

```
1  import java.util.concurrent.locks.*;
2
3  class Locks {
4    public static void main(String [] args) {
5      Lock a = new ReentrantLock();
6      Lock b = new ReentrantLock();
7      Lock c = new ReentrantLock();
8      Lock d = new ReentrantLock();
9
10     // Thread T1
11     new Thread(() -> {
12       while (true) {
13         // Do some stuff
14         b.lock();
15         // Use b
16         a.lock();
17         // Use a and b
18         d.lock();
19         // Use a, b and d
20         d.unlock();
21         a.unlock();
22         b.unlock();
23         // Do more stuff
24       }
25     }).start();
26
27     // Thread T2
28     new Thread(() -> {
29       while (true) {
30         // Do some stuff
31         b.lock();
32         // Use b
33         c.lock();
34         // Use b and c
35         b.unlock();
36         d.lock();
37         // Use c and d
38         d.unlock();
39         c.unlock();
40         // Do more  stuff
41       }
42     }).start();
43
44    // Thread T3
45    new Thread(() -> {
46       while (true) {
47         // Do some stuff
48         b.lock();
49         // Use b
50         d.lock();
51         // Use b and d
52         b.unlock();
53         a.lock();
54         // Use a and d
55         a.unlock();
56         d.unlock();
57         // Do more stuff
58       }
59     }).start();
60   }
61 }
```

uppgiften fortsätter på nästa sida

a) Rita en resursallokeringsgraf för systemet.

(2p)

b) Vilka *två* trådar ingår i den cykel som bildar dödläget och vilka kodrader (ange radnummer) är det de försöker exekvera när dödläget uppstår?

(1p)

c) Den tredje tråden – den som inte ingår i cykeln för dödläget – kommer också att fastna. Varför och på vilken eller vilka kodrader (ange radnummer) kan den göra det?

(1p)

d) Föreslå en ändring av programmet som gör att dödläge inte kan uppstå. De lås som nämns i kommentarerna ska fortfarande vara låsta vid den aktuella punkten i programmet.

(1p)

5. Nedanstående väldigt nerskalade program illustrerar hur dödläge kan uppstå om man har monitorer med nästlade anrop (anrop av en `synchronized`-metod inuti en `synchronized`-metod i en annan monitor). Föreslå en ändring av koden så att dödläge inte längre kan uppstå!

Den första tråden ska fortfarande anropa metoden `m1()` i monitorn `a1` och den andra tråden ska fortfarande anropa metoden `m1()` i monitorn `a2`. Vidare får du inte ändra på metoderna `m1()` eller `m2()`. Det går bra att ändra i klassen `Deadlock` inklusive de två trådarna och/eller att lägga till saker i `A`.

(2p)

```
1  class Deadlock {
2    public static void main(String [] args) {
3      A a1 = new A();
4      A a2 = new A();
5
6      new Thread(() -> {
7        while (true) {
8          a1.m1(a2);
9        }
10     }).start();
11
12     new Thread(() -> {
13       while (true) {
14         a2.m1(a1);
15       }
16     }).start();
17   }
18 }
19
20 class A {
21   synchronized void m1(A a) {
22     System.out.println("m1-1");
23     a.m2();
24     System.out.println("m1-2");
25   }
26
27   synchronized void m2() {
28     System.out.println("m2");
29   }
30 }
```

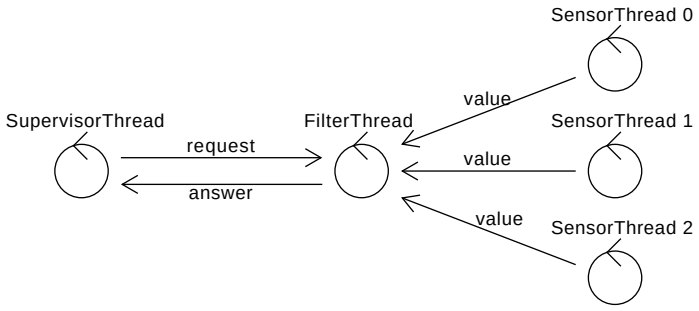
6. Filtrering av sensorvärden

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 av klassen `SensorThread`. Varje instans av `SensorThread` identifieras av ett heltal (0, 1 eller 2) som anger vilken av sensorerna den bevakar. När sensortrådarna har läst av ett nytt värde, vilket tar en viss (kort) tid, skickas värdet som ett meddelande innehållande det avlästa värdet och sensorns nummer (0, 1 eller 2) till en tråd av klassen `FilterThread` som lagrar uppgifterna.

Det finns dessutom en tråd av klassen `SupervisorThread` som regelbundet skriver ut det minsta av de tre uppmätta avstånden eller ett varningsmeddelande om att roboten kommit för nära ett omgivande föremål. Det gör den genom att skicka ett meddelande till `FilterThread`-tråden innehållande en önskad tidigaste samplingstidpunkt (normalt i framtiden). `FilterThread`-tråden skickar tillbaka ett svar enligt följande:

- När det finns tre aktuella mätvärden från de tre sensorna *som alla anlände vid den önskade samplingstidpunkten eller senare* returneras ett meddelande innehållande *det minsta av de tre sparade sensorvärdena*. Vi vill alltså inte använda gamla mätvärden utan alla ska vara färska. Om det *inte* finns tillräckligt nya mätvärden från alla tre sensorerna fördröjs svaret tills så är fallet.
- Om någon av sensorvärdena är mindre än `DANGER_DIST`, eller ett sådant värde anländer under tiden vi väntar på tre aktuella mätvärden, skickas det genast ett svarsmeddelande som säger att fara råder och att minimumavståndet är underskridet.

Se nedanstående figur för en översikt av systemet och hur meddelanden skickas mellan trådarna.



Uppgift

Mycket av koden för övervakningssystemet är färdigskrivet men det behöver kompletteras på ett antal ställen som har med sändningen av meddelanden och logiken för `FilterThread` och `SupervisorThread` att göra. Du finner den färdigskrivna koden nedan tillsammans med specifikationen för `ActorThread` (som du inte behöver skriva). Din uppgift är att komplettera koden så den blir komplett. Du behöver bara svara med den kod du kompletterar med, men det måste framgå tydligt var den ska stoppas in i den befintliga koden.

Det som behöver (eller kan behöva) kompletteras omfattar:

- Eventuella klasser för att representera meddelanden.
- Meddelandetyp för `ActorThread`-klasserna `SupervisorThread` och `FilterThread`.
- Hur meddelanden skickas i `SensorThread` (se dess `run()`-metod).
- Logiken för `SupervisorThread` (se dess `run()`-metod).
- Logiken för `FilterThread` (se dess `run()`-metod).

Du får naturligtvis lägga till fler klasser och privata attribut och metoder i klasserna om du vill.

(10p)

Färdigskriven kod

```
1  class SensorThread extends Thread {
2    private int id;
3    private ActorThread filter;
4
5    public SensorThread(int id, ActorThread filter) {
6      this.id = id;
7      this.filter = filter;
8    }
9
10   public void run() {
11     while(true) {
12       double sensorValue = SensorIO.getValue(id);
13
14       // To Do: Send the sensor value to the filter thread
15
16     }
17   }
18 }
19
20
21 class SupervisorThread extends ActorThread< /* To Do: Add message type */  >  {
22   private ActorThread filter;
23
24   public SupervisorThread(ActorThread filter) {
25     this.filter = filter;
26   }
27
28   public void run() {
29     long nextTime = System.currentTimeMillis();
30     while(true) {
31       // To Do: 1. Send message requesting next minimum value
32       //           newer than nextTime to the filter thread.
33       //        2. Wait for the answer to arrive.
34       //        3. Depending on the contents of the
35       //           received message:
36       //           3a. Print the minimum distance.
37       //           3b. Print "Danger!"
38       nextTime = System.currentTimeMillis()+1000;
39     }
40   }
41 }
42
43 class FilterThread extends ActorThread< /* To Do: Add message type */  > {
44   private static final double DANGER_DIST = 0.8;
45
46   public void run() {
47     // To Do: React to request messages from other threads
48   }
49 }
50
51 public class FilterTask {
52     public static void main() {
53         FilterThread filter = new FilterThread();
54         filter.start();
55         new SupervisorThread(filter).start();
56         for(int i=0;i<3;i++) {
57           new SensorThread(i,filter).start();
58         }
59     }
60 }
```

ActorThread

```
public class ActorThread<M> extends Thread {

    /** Called by another thread,
        to send a message to this thread. */
    public void send(M message)     { ... }

    /** Returns the first message in the queue,
        or blocks if none available. */
    protected M receive()
        throws InterruptedException { ... }

    /** Returns the first message in the queue, or blocks
        up to 'timeout' milliseconds if none available.
        Returns null if no message is obtained within
        'timeout' milliseconds. */
    protected M receiveWithTimeout(long timeout)
        throws InterruptedException { ... }
}
```

7. SupervisedBuffer

Buffertar är väldigt vanliga konstruktioner i flertrådade program för att asynkront överföra information från en tråd till en annan och samtidigt göra det möjligt för trådarna att arbeta i varierande takt (meddelanden kan köas upp i bufferten tills mottagaren är beredd att ta hand om dem). Ett vanligt exempel på en sådan situation är det så kallade Producer–Consumer-mönstret där någon typ av data genereras av en tråd (producenten) varefter det sen skickas över till en annan tråd (konsumenten) för att behandlas. I Java kan en sådan buffert med fördel implementeras med hjälp av en monitor med en operation för att lägga in ett dataobjekt i bufferten (anropas av producenten) och en annan operation för att ta ut ett dataobjekt ur bufferten (anropas av konsumenten).

Uppgiften går ut på att implementera en sådan buffert som kan överföra dataobjekt av godtycklig typ (för enkelhetens skull av typen `Object` – det är ju trots allt bara en tentauppgift), men med lite extra funktionalitet som gör det möjligt för en tredje tråd att övervaka kommunikationen.

Uppgift

Komplettera följande implementation av en buffertklass med den kod som saknas:

```
public class SupervisedBuffer {
  private LinkedList<Object> queue = new LinkedList<>(); // Stores buffered messages

  /** Adds message to the buffer. Never blocks */
  public synchronized void add(Object message) {
    // To Do: Implement this method
  }

  /** Returns the next available message. The method blocks if the
      queue is empty until a new message is added. */
  public synchronized Object take() throws InterruptedException {
    // To Do: Implement this method
  }

  /** This method blocks until timeout milliseconds have passed since
      a message was last added to or returned from the buffer. Can be
      used by a supervisor thread to detect communication stall. */
  public synchronized void awaitInactivity(long timeout) throws InterruptedException {
    // To Do: Implement this method
  }

}
```

Anvisningar

- Metoden `awaitInactivity()` anropas av en övervakningstråd och blockerar tills dess ingenting har hänt i bufferten på `timeout` millisekunder, det vill säga till dess vare sig `add()` anropats eller `take()` har returnerat ett meddelande. Metoden kan användas av en övervakningstråd för att detektera att kommunikationen mellan producenten och konsumenten har upphört av någon anledning.
- Använd Javas inbyggda monitorbegrepp (`synchronized` och så vidare).
- Meddelanden ska lagras i den givna länkade listan `queue`. För uppgiftens skull får du inte använda några andra datastrukturer ur Javas standardbibliotek för detta.
- Det räcker med att du svarar med koden för de tre monitormetoderna ovan.
- Du får införa nya privata klasser, metoder och attribut, men då måste du (naturligtvis) redovisa dem också.
- Det ska vara möjligt att avbryta alla trådar med `interrupt()` utan att bufferten hamanr i ett inkonsistent tillstånd.

(5p)