Tentamen
EDAF85 – Realtidssystem

2021-10–30, 08.00–14.00

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*, frågor 1-4 (15 poäng), och *programmeringsuppgifter*, fråga 5-6 (15 poäng). För godkänd (betyg 3) krävs preliminärt sammanlagt hälften av alla 30 möjliga poäng samt en tredjedel av poängen 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. En programmerare har skrivit följande rader kod i syfte att åstadkomma ömsesidig uteslutning över anropet av `doSomething()`:

```
while (mutex==1) { }    // Wait for mutex lock to become free
mutex = 1;              // Acquire mutex lock
doSomething();          // Perform task requiring mutual exclusion
mutex = 0;              // Release lock
```

Variabeln `mutex` är deklarerad så här: "`volatile int mutex = 0;`". Nyckelordet `volatile` innebär att alla trådar som använder variabeln ser samma värde och att ingen tråd behåller ett eget lokalt värde på variabeln som andra trådar inte ser.

a) Vad brukar vi vanligtvis kalla detta sätt att vänta på att ett villkor blir uppfyllt (”`while (mutex==1) { }`”)? Vilken nackdel har denna metod? *(1p)*

b) Garanterar koden ovan ömsesidig uteslutning för anropen av `doSomething()`? Förklara varför det är så! *(1p)*

2. 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[1] på de fyra mutexsemaforerna A, B, C och D (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
    ==              ==
 1. B.acquire();    B.acquire();
 2. C.acquire();    A.acquire();
 3. useBC();        useAB();
 4. D.acquire();    A.release();
 5. useBCD();       D.acquire();
 6. D.release();    useBD();
 7. C.release();    B.release();
 8. B.release();    A.acquire();
 9. A.acquire();    useAD();
10. C.acquire();    A.release();
11. useAC();        D.release();
12. C.release();
13. A.release();
```

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 och på vilka radnummer i koden ovan måste respektive trådinstans befinna sig på för att det ska finnas risk för dödläge i systemet? *(2p)*

c) Föreslå en enkel ändring av koden ovan som gör att dödläge garanterat inte kan uppstå i systemet, men som fortfarande garanterar att relevanta semaforer (och inga andra) är låsta då de olika ”useXY();”-funktionerna anropas. *(1p)*

[1] För dig som läst kursen före 2020: Semaforoperationerna `take()` och `give()` som användes då du läste kursen ersattes 2020 med operationerna `acquire()` och `release()` i samband med att vi övergick till att använda Javas standardbibliotek för semaforer. Betydelsen är dock densamma – det är bara namnen som skiljer.

3. Beskriv med en eller två meningar vardera innebörden av följande schemaläggningsbegrepp:

a) Strikt prioritetsbaserad, dynamisk, schemaläggning

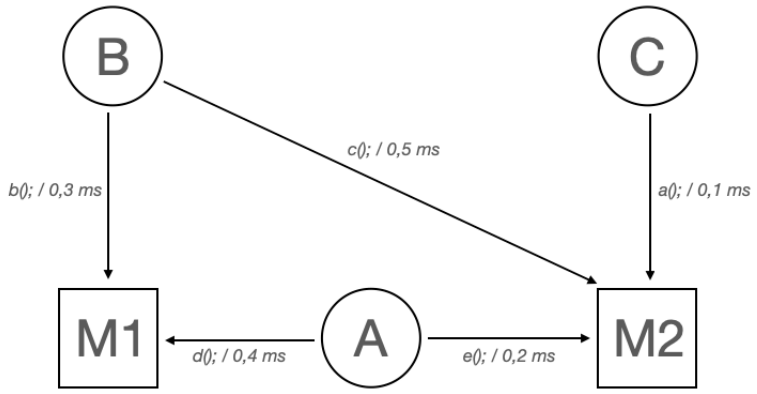
b) Earliest deadline first scheduling (EDF)

c) Rate monotonic scheduling (RMS)

d) Statisk schemaläggning

(2p)

4. 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, d och e i monitorerna M1 och M2 och med maximala exekveringstider för operationerna (i millisekunder) enligt figur. Trådarna anropar en monitoroperation i taget, dvs att anropen inte är nästlade.



Vidare har trådarna värstafallsexekveringstider (C), periodtider (T) och deadlines (D) enligt följande tabell.

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

För att tilldela trådarna prioriteter används principen DMS - Deadline 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. *(3p)*

b) Ange för varje tråd (A, B och C) vad deras värstafallsresponstid (R) blir. *(3p)*

5. AudioController

I laboration 3 skrev du programvaran som styrde en tvättmaskin. Vi ska nu lägga till ytterligare en kontrolltråd till tvättmaskinen: `AudioController`. Denna kontrolltråd är ansvarig för att generera ljud, till exempel när tvättprogrammet har avslutats.

Du kommer att använda klassen `Buzzer` för att styra högtalaren:

```
public class Buzzer {
    /** Place the loudspeaker membrane in the ON position. */
    public void on()              { ... }

    /** Place the loudspeaker membrane in the OFF position. */
    public void off()             { ... }
}
```

Hur man skapar ljud med hjälp av `Buzzer`

Man skapar ljud genom att snabbt växla högtalarens membran mellan på- och avlägena (ON respektive OFF). Ljudets frekvens beror på hur snabbt man växlar mellan lägena. För att skapa ett ljud med frekvensen 5 kHz måste högtalaren kopplas på och av 5000 gånger/sekund.

Följande kod skapar ett ljud vars frekvens är *nästan* 5 kHz. Frekvensen 5 kHz innebär en periodtid på 1/5000 = 0,0002 sekunder, eller 200000 nanosekunder. I detta exempel kommer ljudet att fortsätta låta tills programmet stoppas (`while (true) ...`).

```
Buzzer b = new Buzzer();
while (true) {
  b.on();
  TimeUnit.NANOSECONDS.sleep(100000);
  b.off();
  TimeUnit.NANOSECONDS.sleep(100000);
}
```

Notera två saker angående anropet av `sleep()` ovan:

Den speciella varianten `TimeUnit.NANOSECONDS.sleep(...)` innebär att fördröjningen mäts i nanosekunder (10^{-9} sekunder) istället för i millisekunder. Varje anrop av `sleep()` ger en fördröjning på en halv period om 100000 nanosekunder. Det går två halvperioder på en hel cykel: en med membranet i läge ON och en med membranet i läge OFF.

Varför *nästan* 5 kHz?

I exemplet ovan kommer varje anrop av `sleep()` ge en fördröjning som i praktiken är något mer än 100000 nanosekunder. Det finns flera orsaker till detta, som du kanske kommer ihåg från laboration 1 (alarmklockan). Effekten blir att ljudfrekvensen kommer att variera något och bli något lägre än 5 kHz i exemplet ovan.

AudioRequest-meddelanden

Du ska skriva en trådklass `AudioController` som förväntas kunna ta emot meddelanden av typen `AudioRequest`:

```
public class AudioRequest {
  /**
   * Returns the half-period (in nanoseconds) for the tone to play,
   * or zero (0) if no tone is to be played (silence).
   */
  public int getHalfPeriod() { ... }

  // ...
}
```

Ett meddelande av typen `AudioRequest` ska tolkas enligt följande:

- Ett `AudioRequest` med en halvperiod > 0 innebär att motsvarande ton ska spelas tills ett nytt meddelande anländer.
- Ett `AudioRequest` med en halvperiod = 0 innebär att högtalaren ska vara tyst tills nästa meddelande anländer.
- Du kan ignorera `AudioRequest` med en halvperiod < 0

Att mäta tid i nanosekunder

Du är säkert van vid att mäta tid i millisekunder i Java. I denna uppgift kommer vi att arbeta med nanosekunder.

Använd `System.nanoTime()` för att mäta tid. `System.nanoTime()` fungerar precis som `System.currentTimeMill` fast tiden mäts i nanosekunder.

```
long t0 = System.nanoTime();
// ...
long t1 = System.nanoTime();
long dt = t1 - t0;     // time in nanoseconds
```

Du kan använda `TimeUnit.NANOSECONDS.sleep()` för att fördröja exekveringen ett antal nanosekunder. `TimeUnit.NANOSECONDS.sleep()` fungerar precis som `Thread.sleep()` förutom att tiden mäts i nanosekunder (se Buzzer-exemplet ovan).

Klassen `ActorThread`[2] har modifierats i denna uppgift så att metoden `receiveWithTimeout()` mäter tiden i nanosekunder:

```
public class ActorThread<M> extends Thread {
  /**
   * 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 'timeoutNano' nanoseconds if none available.
   * Returns null if no message is received
   * within 'timeoutNano' nanoseconds.
   */
  protected M receiveWithTimeout(long timeoutNano)
    throws InterruptedException      { ... }
}
```

Din uppgift

Du ska skriva en klass `AudioController` som är en subklass av `ActorThread`, och som ska kunna användas på samma sätt som de andra kontrolltrådarna i tvättmaskinslaborationen. `ActorThread` har, som beskrevs ovan, modifierats så att den arbetar med nanosekunder i stället för millisekunder.

[2] Till dig som läste kursen före 2020: När du gjorde laborationen ärvde kontrolltrådarna från klassen `RTThread` som tog emot meddelanden av typen `RTEvent`. Denna klass har i laborationen numera ersatts av `ActorThread` som tar emot meddelanden av typen given av `M`.

Följande första försök till lösning är given:

```
class AudioController extends ActorThread<AudioRequest> {
    public void run() {
        try {
            Buzzer buzzer = new Buzzer();
            while (true) {
                AudioRequest message = receive();

                int halfPeriod = message.getHalfPeriod();

                while (true) {
                    buzzer.on();
                    TimeUnit.NANOSECONDS.sleep(halfPeriod);
                    buzzer.off();
                    TimeUnit.NANOSECONDS.sleep(halfPeriod);
                }
            }
        } catch (InterruptedException unexpected) {
            throw new Error(unexpected);
        }
    }
}
```

Den givna klassen `AudioController` har två problem:

1. När den väl börjat spela en ton så reagerar den inte längre på meddelanden. Den fortsätter bara spela samma ton för evigt.
2. Tonen håller inte riktigt rätt frekvens utan är något för låg (se avsnittet ”Varför *nästan* 5 kHz?” ovan).

Modifiera klassen `AudioController` ovan så att...

1. .. den reagerar på nya meddelanden. När ett nytt meddelande anländer ska tråden börja spela tonen med den nya frekvens eller, om halvperioden är 0, upphöra med att spela tonen.
2. ...spelar en korrekt ton genom att korrigera för klockdriften.
3. ...inte konsumerar onödig CPU-kraft när inget ljud ska spelas.

Observera att till skillnad från de andra kontrolltrådarna i tvättmaskinslaborationen förväntas denna tråd *inte* skicka några bekräftelsemeddelanden på att den tagit emot ett nytt kommando (ACK). *(6p)*

6. Kaos på mäklarfirman

Hjälp!!! Ett gäng glada ekonomer har nyss lärt sig programmera och gett sig på att skriva ett programsystem i Java som ska hjälpa dem matcha köpare med säljare på en råvarubörs. För att förenkla för sig så nöjer de sig till en början med att hantera ett enda råvaruslag, men det hjälpte inte. De är säkra på att logiken i hur själva matchningen mellan köpare och säljare går till har blivit rätt, men eftersom systemet ska kunna hantera många köpare och säljare som samtidigt lägger köp- respektive säljorder så behöver de göra systemet flertrådat. Där har dock något gått allvarligt snett. Det blir därför din uppgift att rädda dagen och fixa synkroniseringen i deras program.

Hur matchas köpare och säljare?
Köpare som vill köpa en mängd av råvaran lägger en köporder som består av uppgifter om hur står mängd (mäts i *enheter*) som köpet gäller och vad det maximala priset får vara. För att veta vem som är köparen har även varje köpare ett unikt ID-nummer. En köporder har också en giltighetstid och om den överskrids stryks ordern helt och genomförs inte. På liknande sätt kan säljare lägga in en säljorder som talar om hur stor mängd av varan som är till försäljning och vilket pris som begärs för varan (pris per enhet). Även säljorder är tidsbegränsade, men det behöver inte finnas köpare till hela den tillgängliga mängden för att ett köp ska kunna genomföras. Om giltighetstiden gått ut utan att hela mängden blivit såld stryks den resterande delen av ordern.

När en köporder kommer in till systemet läggs den i en lista sorterad i den ordning de kom in. Inkommande säljorder läggs också in i en lista, men denna är sorterad med avseende på pris i första hand och, om flera order har samma pris per enhet, i andra hand i den ordning de kom in till systemet. För att matcha köpare med säljare tittar man hela tiden på dessa två listor och försöker matcha en köpare med en, eller om nödvändigt (vid stora köporder) flera, säljare.

För att hitta en matchning så går man igenom listan av inkomna köporder med början på den äldsta och söker framåt i listan tills man hittar en order som går att uppfylla givet de inkomna säljorderna. För att avgöra om en köporder går att uppfylla, till så bra pris som möjligt, letar man igenom listan med säljorder med början på den äldsta av de billigaste säljorderna. Om man kan hitta tillräckligt många säljorder med ett tillräckligt lågt pris för att förse köparen med tillräckligt många enheter matchar man köpordern med de funna säljorderna, genomför köpet genom att räkna ner antalet tillgängliga enheter i de aktuella säljorderna och tar bort dem om antalet kvarvarande enheter för ordern blir noll. Köpordern tas också bort när den är genomförd. Sedan börjar man om från början igen och försöker hitta en ny order att matcha. Går det inte att hitta en möjlig matchning väntar man tills nya order kommer in och gör en matchning möjlig.

Programdesign
I våra ekonomvänners program finns det två slags trådar: *klienttrådar* som hanterar kommunikationen med kunderna (köpare och säljare) samt en central *mäklartråd* som utför matchningarna och genomför affärerna. Alla dessa trådar kommunicerar med varandra genom ett centralt objekt av typen `BrokerCentral`. När en klienttråd vill köpa eller sälja ett parti av råvaran anropar den en av metoderna `buy()` eller `sell()` i `BrokerCentral`. Dessa metoder blockerar tills ordern antingen är helt genomförd eller dras tillbaka eftersom dess giltighetstid gått ut. `BrokerCentral` innehåller dessutom en metod *matchBuyersAndSellers()* som mäklartråden anropar. Det är denna metod som kontinuerligt genomför matchningarna mellan köpare och säljare och genomför affärerna.

Datastruktrurer

För att representera orderna och lagra dessa finns följande klasser:

```
// Abstract class representing an order
class Order {
    public int units;           // Number of units available or requested
    public double pricePerUnit; // Price per unit. Selling price or maximum buying price
    public int clientId;        // Identifies the buyer/seller
}

// Represents an order to buy units
class BuyOrder extends Order {
    public double pricePaid;  // Total price actually paid or 0.0 if order was cancelled
}

// Represents an order to sell units
class SellOrder extends Order {
    public int unitsSold; // The number of units actually sold
}

// Represents a match between a buy order and the sell orders that best satisfies the buy
// A list of sell orders is needed because one sell order may not be enough to satisfy
// the need of the buyer.
class Match {
    public BuyOrder buyer;
    public SellOrderList sellers;
}


// Stores orders sorted by price first and arrival time second
class SellOrderList extends LinkedList<SellOrder> {
    // Inserts a new order in the correct place in the list
    public void insert(SellOrder o) {
        ListIterator<SellOrder> iter = listIterator(0);
        while(iter.hasNext()) {
            SellOrder s = iter.next();
            if (o.pricePerUnit<s.pricePerUnit) {
                iter.previous();
                iter.add(o);
                return;
            }
        }
        addLast(o);
    }
}

// Stores orders in the order they arrived
class BuyOrderList extends LinkedList<BuyOrder> {
    // Inserts a new order in the correct place in the list
    public void insert(BuyOrder o) {
        addLast(o);
    }
}
```

Klassen `BrokerCentral`

Hjärtat i ekonomernas program är ett objekt av typen `BrokerCentral`. Här finns de tre metoder som anropas av de olika trådarna i programmet:

double buy(BuyOrder o,long validUntil) En klienttråd som vill genomföra ett köp fyller i ett `BuyOrder`-objekt som skickas till denna metod tillsammans med en giltighetstid för ordern. Anropet blockerar tills antingen köpet kunde genomföras eller tiden löpt ut. Om ordern kunde genomföras returneras det totala priset som köparen fick betala (kan variera beroende på vilka priser de inblandade säljarna begärde för sina partier). Annars returneras 0.

int sell(SellOrder o,long validUntil) Anropas när en klienttråd har ett parti till salu. Ett motsvarande `SellOrder`-objekt skapas och skickas till denna metod tillsammans med en giltighetstid för ordern. Anropet blockerar tills antingen hela partiet blivit sålt eller tiden löpt ut. Därefter returneras hur många enheter som faktiskt blev sålda.

void matchBuyersAndSellers() Anropas av den ensamma mäklartråden och är den som ansvarar för matchningen av köpare med säljare. Returnerar ej.

Koden som ekonomerna har skrivit ser ut så här:

```
class BrokerCentral {
    private SellOrderList sellOrders = new SellOrderList();
    private BuyOrderList buyOrders = new BuyOrderList();

    // Attempts to buy the requested units in o for the given maximum price or less
    // before System.currentTimeMillis() reach validUntil.
    // If the total order cannot be satisfied before then the order is cancelled.
    // Returns the total amount paid or 0.0 if the order was cancelled.
    public double buy(BuyOrder o,long validUntil) {
        o.pricePaid = 0.0;
        buyOrders.insert(o);
        while (o.pricePaid==0.0 && System.currentTimeMillis()<validUntil) { }
        if (o.pricePaid==0.0) {
            buyOrders.remove(o);
            System.out.println("Buy order cancelled for client "+o.clientId+".");
        }
        return o.pricePaid;
    }

    // Attempts to sell the amount of units in o for a price no less than the given
    // price before System.currentTimeMillis() reach validUntil.
    // Return the total number of units actually sold.
    public int sell(SellOrder o,long validUntil) {
        o.unitsSold = 0;
        sellOrders.insert(o);
        while (o.units>0 && System.currentTimeMillis()<validUntil) { }
        if (o.units>0) {
            sellOrders.remove(o);
            System.out.println("Sell order cancelled for client "+o.clientId+". "
                               +o.units+" remaining unsold.");
        }
        return o.unitsSold;
    }
```

```
    // Continuously matches buyers and sellers
    public void matchBuyersAndSellers() {
        while (true) {
            Match m = nextBuyOrder();
            if (m!=null) {
                for(SellOrder s: m.sellers) {
                    int n = m.buyer.units;
                    if (n>s.units) {
                        n = s.units;
                    }
                    m.buyer.pricePaid += s.pricePerUnit*n;
                    s.units -= n;
                    s.unitsSold += n;
                    if (s.units==0) {
                        sellOrders.remove(s);
                    }
                    System.out.println("Client "+m.buyer.clientId+" buys "+n
                                       +" units from client "+s.clientId
                                       +" at price "+s.pricePerUnit+".");
                }
                buyOrders.remove(m.buyer);
            }
        }
    }

    // Returns a Match object describing a buy order that can be fully satisfied
    // together with a list of the sell orders used to satisfy the order.
    // If no match is found, null is returned.
    private Match nextBuyOrder() {
        Match m = new Match();
        for(BuyOrder b: buyOrders) {
            m.buyer = b;
            m.sellers = new SellOrderList();
            int unitsWanted = b.units;
            for(SellOrder s: sellOrders) {
                if (b.pricePerUnit>=s.pricePerUnit) {
                    unitsWanted -= unitsWanted<s.units ? unitsWanted : s.units;
                    m.sellers.add(s);
                    if (unitsWanted==0) {
                        return m;
                    }
                }
            }
        }
        return null;
    }
}
```

Din uppgift

En IT-konsult ekonomerna anlitat har kommit fram till att problemen med ekonomernas program beror på att klassen `BrokerCentral` inte är skriven på ett sätt som hanterar synkroniseringen mellan trådarna på ett korrekt sätt, men att alla problem kan lösas om man bara ändrar i koden för denna klass. Tyvärr var konsulten för dyr för att ekonomerna skulle anlita denne för att göra ändringarna (eller var ekonomerna kanske för snåla?).

I stället blir det din uppgift att göra de ändringar i `BrokerCentral` som är nödvändiga. Du behöver bara redovisa de delar av klassen i vilka du gjort ändringar. Om du inte ändrar på attributen eller lämnar någon av metoderna oförändrad behöver du inte ta med dessa i ditt svar, men om du ändrar i en metod så ta med hela metoden. Du kan dock utelämna utskriftsraderna (de som börjar med ”`System.out...`”). Du ska inte ändra på någon annan del av programmet än `BrokerCentral`.

(9p)