Tentamen
EDAF85 – Realtidssystem

2025-04-24, 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 1-3 (15 poäng), och en *programmeringsuppgift*, fråga 4 (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. Under kursen har vi använt semaforer till två olika saker, nämligen för att åstadkomma ömsesidig uteslutning och för signalering. Redogör kortfattat 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. *(2p)*

2. Ett realtidssystem innehåller tre trådinstanser, T1, T2, and T3. Deras `run()`-metoder innehåller nedanstående kodsekvenser. S1, S2, S3, S4 och S5 är alla räknande semaforer med startvärde 1. Anrop av typen `useXYZ()` representerar ett kodavsnitt för vilket semaforerna X, Y och Z måste vara tagna (*acquired*).

radnr	**T1.run()**	**T2.run()**	**T3.run()**
1	S5.acquire();	S4.acquire();	S1.acquire();
2	useS5();	useS4();	useS1();
3	S5.release();	S5.acquire();	S1.release();
4	S2.acquire();	S1.acquire();	S2.acquire();
5	S5.acquire();	useS1S4S5();	S3.acquire();
6	useS2S5();	S1.release();	useS2S3();
7	S2.release();	S5.release();	S4.acquire();
8	useS5();	S4.release();	useS2S3S4();
9	S5.release();	S2.acquire();	S4.release();
10	S1.acquire();	S3.acquire();	S3.release();
11	S2.acquire();	useS2S3();	S2.release();
12	useS1S2();	S3.release();	
13	S2.release();	S2.release();	
14	S1.release();		

I ett sådant system skulle det kunna finnas risk för *dödläge*.

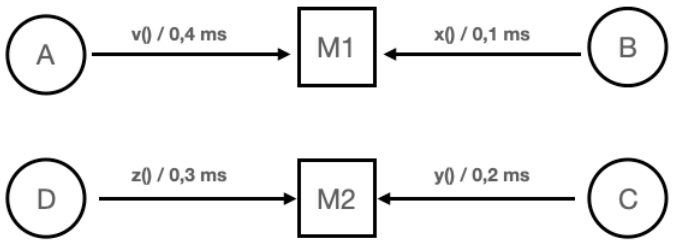
a) Rita en resursallokeringsgraf för systemet. *(2p)*

b) Hur ser man från resursallokeringsgrafen om det finns risk för dödläge i systemet? *(1p)*

c) Hur många olika fall (scenarier) finns det i vilka systemet ovan kan hamna i dödläge? *(1p)*

d) Ange för vart och ett av fallen i förra deluppgiften vilka rader (radnummer) som de olika trådarna som är inblandade i dödläget försöker exekvera när de blockerar. *(2p)*

3. Antag att vi har ett system som schemaläggs enligt Deadline 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.



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

Tråd	*C (ms)*	*T (ms)*	*D (ms)*
A	4	20	20
B	1	4	2
C	3	12	6
D	2	24	12

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 termen B_i står för och vad som påverkar dess värde. *(1p)*

b) Vi skulle kunna räkna ut det totala CPU-utnyttjandet för systemet, dvs $\sum_{i=1}^{n} \frac{C_i}{T_i}$. Vilken, om någon, slutsats kan vi dra om det aktuella systemets schemaläggningsbarhet utifrån värdet vi då får fram? Motivera ditt svar. *(1p)*

c) Räkna ut B_i för respektive tråd. *(2p)*

d) Räkna nu ut R_i för respektive tråd. *(2p)*

e) Är systemet schemaläggningsbart? För att få poäng krävs en korrekt motivering till svaret. *(1p)*

4. Dörröppnare

Denna uppgift handlar om att skriva styrprogrammet till en dörröppnare för en skjutdörr som aktiveras genom att man trycker på en knapp (eller av någon typ av närvarosensor) när man vill öppna dörren. Dörren öppnas då helt och förblir öppen en viss tid (i vårt fall 5 sekunder). Därefter stängs åter dörren. Om dörren håller på att stängas när ytterligare en person vill passera kan denne trycka på knappen igen och stängning avbryts och dörren öppnas igen. När den varit helt öppen i ytterligare 5 sekunder gör den ett nytt försök att stänga sig. Du är säkert väl bekant med funktionen från verkligheten.

Antag att följande klass finns tillgänglig för att styra (slå på/av) dörrmotorerna samt för att känna av om någon trycker på knappen eller om dörren når (eller står i) något av sina ändlägen – fullt öppen eller helt stängd:

```
class DoorIO {
    /**
        Blocks the calling thread until the button is pressed.
        The method is thread safe, i.e., it and other threads in the class can
        be called simultaneously by different threads without conflict.
     */
    public void awaitButton();

    /**
        Blocks the calling thread until the door reaches its fully open position.
        The method is thread safe, i.e., it and other threads in the class can
        be called simultaneously by different threads without conflict.
     */
    public void awaitOpen();

    /**
        Blocks the calling thread until the door reaches its fully closed position.
        The method is thread safe, i.e., it and other threads in the class can
        be called simultaneously by different threads without conflict.
     */
    public void awaitClosed();

    /** Activates the door motor in order to close the door. */
    public void startOpening();

    /** Activates the door motor in order to close the door. */
    public void startClosing();

    /** Stops the door motor. */
    public void stop();
}
```

Ett enkelt styrprogram för dörren skulle då kunna se ut så här:

```
class DoorOpenerDemo {
    public static void main(String [] args) throws InterruptedException {
        DoorIO io = new DoorIO();

        while (true) {
            io.awaitButton();
            io.startOpening();
            io.awaitOpen();
            io.stop();
            Thread.sleep(5000);
            io.startClosing();
            io.awaitClosed();
            io.stop();
        }
    }
}
```

Först väntar vi på att knappen trycks in varefter vi aktiverar motorn som öppnar dörren. När dörren nått sitt fullt öppna läge stänger vi av motorn och väntar i 5 sekunder. Därefter aktiverar vi motorn som stänger dörren och när den är helt stängd stoppar vi motorn och börjar om från början.

Enda nackdelen med denna lösning är att vi inte kan avbryta stängningen medan den pågår utan måste vänta tills dörren stängts helt varvid de som ska passera dörren kan klämmas eller åtminstone bli rejält irriterade. Problemet är att vi inte kan vänta på både en knapptryckning och att dörren ska nå sitt fullt stängda läge samtidigt från en och samma tråd, se `awaitClosed()` respektive `awaitButton()`.

Detta kan vi lösa genom att använda trådar och vi kommer att presentera två olika lösningar i deluppgifterna nedan. De är dock inkompletta kodmässigt och det är därför din uppgift att komplettera med den kod som saknas.

Deluppgift 1 – monitorbaserad lösning

För att lösa problemet med att vi behöver bevaka flera möjliga insignaler samtidigt kan vi använda oss av separata trådar som bevakar insignalerna – en för varje möjlig insignal. Dessutom behöver vi en extra tråd som baserat på de insignaler som kommer styr dörrens motor. För att samordna alla trådarna använder vi en monitor med följande utseende:

```
class DoorMonitor {
        /* Tells the monitor that the door is now fully open.  */
        public synchronized void reportOpen() ;

        /* Tells the monitor that the door is now fully closed. */
        public synchronized void reportClosed();

        /*  Tells the monitor that the button has been pressed. */
        public synchronized void reportButton();

        /* Waits until the button has been pressed and starts opening the door.
            When the door is fully open the door motor is stopped.  */
        public synchronized void handleOpening(DoorIO io);

        /* Starts closing the door and waits until the door is either fully closed
            or the button is pressed. Then it stops the door motor. */
        public synchronized void handleClosing(DoorIO io);
}
```

Själva styrprogrammet kan nu skrivas så här:

```
public static void main(String [] args) throws InterruptedException {
        DoorIO io = new DoorIO();
        DoorMonitor mon = new DoorMonitor();

        new Thread(() -> {
                while (true) {
                        io.awaitButton();
                        mon.reportButton();
                }
        }).start();

        new Thread(() -> {
                while (true) {
                        io.awaitOpen();
                        mon.reportOpen();
                }
        }).start();
```

```
        new Thread(() -> {
                while (true) {
                        io.awaitClosed();
                        mon.reportClosed();
                }
        }).start();

        while (true) {
                mon.handleOpening(io);
                Thread.sleep(5000);
                mon.handleClosing(io);
        }
}
```

Lösningen är komplett bortsett från att monitorn `DoorMonitor` inte är färdigimplementerad. Det är din uppgift! Du får lägga till egna privata attribut och metoder i klassen och du får även vid behov komplettera med egna helt nya hjälpklasser om så önskas. *(8p)*

Deluppgift 2 – mailboxbaserad lösning

I denna lösningsvariant använder vi meddelandesändning (mailboxes) för synkronisering/kommunikation mellan trådarna i stället för en monitor. Återigen finns det en tråd för varje möjlig insignal, se trådklasserna `ButtonThread`, `OpenThread` samt `CloseThread` nedan. Skillnaden är att i stället för att trådarna anropar en metod i en monitor ska de skicka meddelanden till en central tråd som sköter motorstyrningen, se trådklassen `DoorHandlerThread`.

Din uppgift är att bestämma hur meddelandena som skickas mellan trådarna ska se ut samt fylla i koden för `run()`-metoderna i de fyra klasserna nämnda ovan. Du behöver också ange meddelandetypen för meddelandena.

- Klassen `ActorThread` som vi använde i tvättmaskinslaborationen antas vara tillgänglig för att underlätta meddelandesändning mellan trådarna. Du hittar dess specifikation sist i koden nedan.
- Typargumentet till klassen `ActorThread` i deklarationen av `DoorHandlerThread` avgör hur dina meddelanden ser ut. Ange på något sätt vad typen ska vara.
- Du får lägga till egna privata attribut och metoder i klasserna som du ska komplettera och du får även vid behov lägga till egna helt nya hjälpklasser om så önskas.

(7p)

Kod till deluppgift 2

```
class DoorHandlerThread extends ActorThread< ... > {
    // Ange typparametern för ActorThread ovan
    private DoorIO io;

    public DoorHandlerThread(DoorIO io) {
        super(10);
        this.io = io;
    }

    public void run() {
        // Komplettera denna metod!
    }
}
```

```
class ButtonThread extends Thread {
    private DoorIO io;
    private DoorHandlerThread dht;

    public ButtonThread(DoorIO io,DoorHandlerThread dht) {
        this.io = io;
        this.dht = dht;
    }

    public void run() {
        // Komplettera denna metod!
    }

}

class OpenThread extends Thread {
    private DoorIO io;
    private DoorHandlerThread dht;

    public OpenThread(DoorIO io,DoorHandlerThread dht) {
        this.io = io;
        this.dht = dht;
    }

    public void run() {
        // Komplettera denna metod!
    }

}

class CloseThread extends Thread {
    private DoorIO io;
    private DoorHandlerThread dht;

    public CloseThread(DoorIO io,DoorHandlerThread dht) {
        this.io = io;
        this.dht = dht;
    }

    public void run() {
        // Komplettera denna metod!
    }

}


// Denna klass ska du INTE implementera!
public abstract 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;
}
```