Tentamen
EDAF85 – Realtidssystem

2023–10–28, 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

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

$$\lim_{n \to \infty} n(2^{1/n} - 1) \approx 0,693147$$

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

1. Ett enkelt realtidssystem består av två periodiska trådar med maximal exekveringstid (C) och periodtid (T) enligt nedanstående tabell. Vi antar att trådarnas deadline är lika med deras respektive periodtid ($D = T$). Trådarna schemaläggs enligt principen *Earliest Deadline First* (EDF).

	C (ms)	T (ms)
A	5	8
B	1	5

a) Förklara kortfattat (med en eller två meningar) hur schemaläggningen går till när Earliest Deadline First används. *(1p)*

b) Antag att båda trådarna skulle vilja börja köra vid tidpunkten 0 (noll). Visa, genom att rita upp ett *schemaläggningsdiagram* för tiden 0–30 ms, hur trådarna kommer att köra i detta fall (vi utgår från att vi kör på en enkelprocessormaskin med en enda CPU-kärna). För att underlätta uppritandet bifogas ett särskilt svarsblad sist i denna tentamen som du kan använda om du vill. Riv loss arket och lämna in det tillsammans med dina övriga svar.

(2p)

2. Betrakta nedanstående `main()`-metod. Metoderna `useXY()`, `useYZ()`, `useZX()`, `useXW()`, `useYW()` och `useWZ()` visas inte, men tar inga ytterligare lås. Däremot förutsätter de att låsen med samma namn som bokstäverna efter ”use” i namnet är låsta när de körs.

När programmet körs förväntas användaren mata in ett heltal n. Detta antal bestämmer hur många instanser av trådklassen T3 som skapas. Programmet hamnar tyvärr ibland i dödläge.

```
 1 public static void main(String[] args) {
 2
 3     Lock x = new ReentrantLock();
 4     Lock y = new ReentrantLock();
 5     Lock z = new ReentrantLock();
 6     Lock w = new ReentrantLock();
 7
 8     Scanner scan = new Scanner(System.in);
 9     System.out.print("Enter number of T3 thread instances: ");
10     int n = scan.nextInt();
11
12     new Thread(() -> {
13         while (true) {     // T1, one instance
14             x.lock();
15             y.lock();
16             useXY();
17             x.unlock();
18             z.lock();
19             useYZ();
20             z.unlock();
21             y.unlock();
22         }
23     }).start();
24
25     new Thread(() -> {
26         while (true) {     // T2, one instance
27             z.lock();
28             x.lock();
29             useZX();
30             z.unlock();
31             w.lock();
32             useXW();
33             w.unlock();
34             x.unlock();
35         }
36     }).start();
37
38     for (int i = 0; i < n; i++) {
39         new Thread(() -> {     // T3, n instances
40             while (true) {
41                 y.lock();
42                 w.lock();
43                 useYW();
44                 y.unlock();
45                 z.lock();
46                 useWZ();
47                 w.unlock();
48                 z.unlock();
49             }
50         }).start();
51     }
52 }
```

a) När programmet startas anger man värdet på `n` vilket styr antalet trådar av typen T3 som startas. Vad är det lägsta värdet på `n` som kan ge upphov till dödläge? *(1p)*

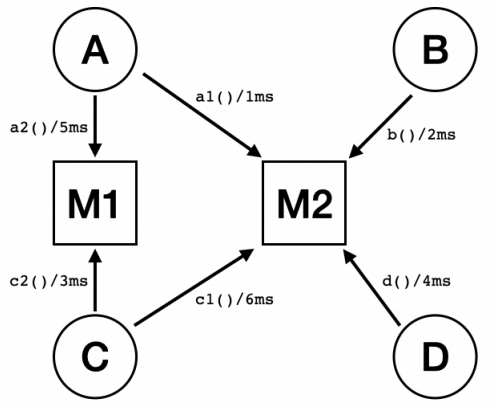
b) När ett program med minimalt antal T3-trådar som krävs för dödläge faktiskt hamnar i dödläge så kommer varje tråd som ingår i dödläget befinna sig på en bestämd rad. Ange för varje trådobjekt som är inblandat i dödläget vilket radnummer den befinner sig på. *(2p)*

3. Ett realtidssystem består av tre periodiska trådar med maximal exekveringstid (C), periodtid (T) och blockeringsfaktor (B) enligt nedanstående tabell. Vi antar att trådarnas deadline är lika med deras respektive periodtid ($D = T$) och att direkt prioritetsarv tillämpas. Trådarna schemaläggs enligt principen rate monotonic scheduling (RMS). Ange för varje tråd vad dess värstafallssvarstid (R) blir.

	C (ms)	T (ms)	B (ms)
A	3	10	0,3
B	4	20	0
C	2	6	0,2

(3p)

4. Ett realtidssystem med dynamisk prioritetsbaserad schemaläggning och prioritetsarv (basic inheritance protocol) består av fyra periodiska trådar (A, B, C och D). De kommunicerar med varandra genom att anropa monitoroperationerna `a1()`, `a2()`, `b()`, `c1()`, `c2()` och `d()` i monitorerna M1 och M2 och med maximala exekveringstider (millisekunder) enligt nedanstående figur. Monitormetoderna anropas en i taget (inga nästlade anrop). A har högst prioritet, B näst högst, C näst lägst och D har lägst prioritet.



Ange för var och en av de fyra trådarna (A, B, C och D) den maximala tid tråden kan bli blockerad av lägre prioriterade trådar under en och samma körning (dvs deras blockeringsfaktorer – B). *(3p)*

5. Förklara, med hjälp av ett enkelt exempel, vad *prioritetsinversion* (eller prioritetsinvertering, eng. priority inversion) är för något. Förklaringen ska också, för full poäng, beskriva varför prioritetsinversion är dåligt i system med strikta realtidskrav. *(2p)*

6. Förklara kortfattat vad begreppet *busy-wait* innebär och varför det är en dålig teknik i ett flertrådat program. Vilken/vilka effekter kan användandet av busy-wait få? *(1p)*

7. Readers/Writers

Från kursen bör du vara bekant med lås (eng. lock) som är ett slags mutexsemafor. Ett lås tillhandahåller ömsesidig uteslutning: det kan bara vara taget av en tråd åt gången. Andra trådar kan inte ta ett upptaget lås förrän den nuvarande ägaren har släppt låset.

Ibland kan detta vara otillräckligt i den meningen att det ger upphov till onödigt mycket blockeringar mellan trådarna. Tänk dig en datastruktur som utan problem kan *läsas* av många trådar samtidigt, men för vilken en tråd som *skriver* till datastrukturen behöver ha exklusiv access. Om ett vanligt lås tas av en läsande tråd så blockeras andra trådar från samtidig läsning. Ett vanligt lås kan alltså i ett sådant läge begränsa parallelliteten – särskilt om läsning är mycket vanligare än skrivning.

För att hantera detta kan vi istället använda oss av ett så kallat *readers-writers-lås*. Ett sådant lås har två olika operationer för låsning, `lockR()` och `lockW()`, vilka fungerar enligt följande:

- Att ta låset för läsning (`lockR()`) innebär att vänta på alla eventuella skrivande trådar som har tagit eller vill ta låset. Andra läsande trådar stoppar däremot oss inte från att ta låset.
- Att ta låset för skrivning (`lockW()`) innebär att vi måste vänta på att alla andra trådar som håller låset (läsare och/eller skrivare) har släppt det innan vi kan ta det själva.
- Om både läsare och en skrivare väntar på att ta låset ska skrivare prioriteras och ges förtur.

När en läsning eller skrivning är klar så anropar tråden som håller låset operationen `unlockR()` eller `unlockW()` beroende på om tråden var en läsare eller en skrivare för att släppa låset:

```
lock.lockR();
// perform read access
lock.unlockR();
```

```
lock.lockW();
// perform write access
lock.unlockW();
```

Uppgift

Skriv en klass `RWLock` som implementerar ett readers-writers-lås som fungerar enligt ovanstående beskrivning. Använd dig av Javas inbyggda monitorbegrepp (`synchronized` etc.). Utgå från följande klasskelett:

```
public class RWLock {
    // Put your private data and your constructor (if you need one) here
    public synchronized void lockR() throws InterruptedException { ... }
    public synchronized void unlockR() { ... }
    public synchronized void lockW() throws InterruptedException { ... }
    public synchronized void unlockW() { ... }
}
```

Kommentarer

- Du kan *inte* göra några antagande om vilka prioriter de olika trådarna har. Din lösning får alltså inte bygga på trådprioriteter.
- **Tips:** Förutom att hålla reda på om låset är taget för skrivning eller inte vill du troligen även hålla reda på antalet trådar som just nu håller på att läsa och antalet trådar som vill börja skriva.
- Det ska gå att avbryta en tråd som blockerat i ett anrop av `lockR()` eller `lockW()` genom att anropa `interrupt()` på densamma. Anropet av `lockR()` eller `lockW()` ska då kasta ett `InterruptedException`.
- För full poäng (10 poäng) får låset efter att ha avbrutits av ett anrop av `interrupt()` inte hamna i ett inkonsistent läge som gör att efterföljande anrop inte fungerar korrekt. Låset ska i det läget bete sig som om låsanropet aldrig skett. En lösning som ignorerar detta fall ger högst 8 poäng.

(10p/8p)

8. Kölappssystem

I övning 2 i kursen gick vi igenom hur man kunde implementera ett enkelt kölappssystem som använde sig av en monitor för att synkronisera trådarna och skydda delade data. I korthet bestod systemet av tre olika trådklasser som alla arbetade mot en gemensam monitor:

`CustomerHandler` Blockerar tills en anländande kund trycker på en knapp på kölappsautomaten. Därefter anropas monitorn som ger tillbaka nästa lediga könummer. Könumret skrivs ut på en papperslapp.

`ClerkHandler` Det finns ett trådobjekt av denna typ för varje kassa. När personen i kassan är ledig att betjäna en ny kund trycker hen på en knapp. Monitorn anropas då för att tala om för den att kassan med angivet nummer nu är ledig.

`DisplayHandler` Denna tråd ansvarar för att visa på en display vilken kund som ska gå till vilken kassa. Tråden anropar monitorn och när det är möjligt att kombinera ihop en kund med en ledig kassa returneras ett objekt från monitorn som talar om vilken kund som ska gå till vilken kassa. Information visas minst 10 sekunder innan nästa kund och kassa visas.

I denna uppgift vill vi lösa samma uppgift, men nu genom att använda oss av meddelandesändning med hjälp av `ActorThread` som du använde i laboration 3, tvättmaskinen. Vi byter då ut den gemensamma monitorn mot en tråd, `Coordinator`, som ansvarar för att para ihop kunder med lediga kassor. Trådklasserna `CustomerHandler`, `ClerkHandler` och `DisplayHandler` måste då ändras så att istället för att göra ett monitoranrop så skickar de ett meddelande till `Coordinator` och eventuella returvärden kommer nu i form av ett svarsmeddelande från `Coordinator`.

En stor del av koden för denna variant av systemet är redan skriven, se nedan, men det återstår för dig att skriva koden för run()-metoden i klassen `Coordinator`. Svara med den kod som ska stoppas in på platsen märkt ”`/* INSERT YOUR CODE HERE */`”.

Kommentarer

- Du ska inte behöva ändra något i koden förutom i `run()`-metoden i `Coordinator`.
- Klassen `Coordinator` tar emot meddelanden i form av heltal. Värdet 0 (noll) betyder att meddelandet kommer från `CustomerHandler`. Värden större än 0 kommer från `ClerkHandler`-trådarna och består av den aktuella kassans nummer.
- I `run()`-metoden i `Coordinator` finns det redan några föreslagna lokala variabler för att hålla reda alla nödvändiga data. Använd dem om du vill och behöver (eller låt bli).
- Hur du väljer vilken kassa som ska betjäna en kund påverkar ej poängbedömningen (så länge kassan är ledig).
- Klassen `Hardware` har utelämnats, men borde vara självförklarande.
- Sist i uppgiften återfinns en specifikation för klassen `ActorThread`.

(5p)

QueueSystem

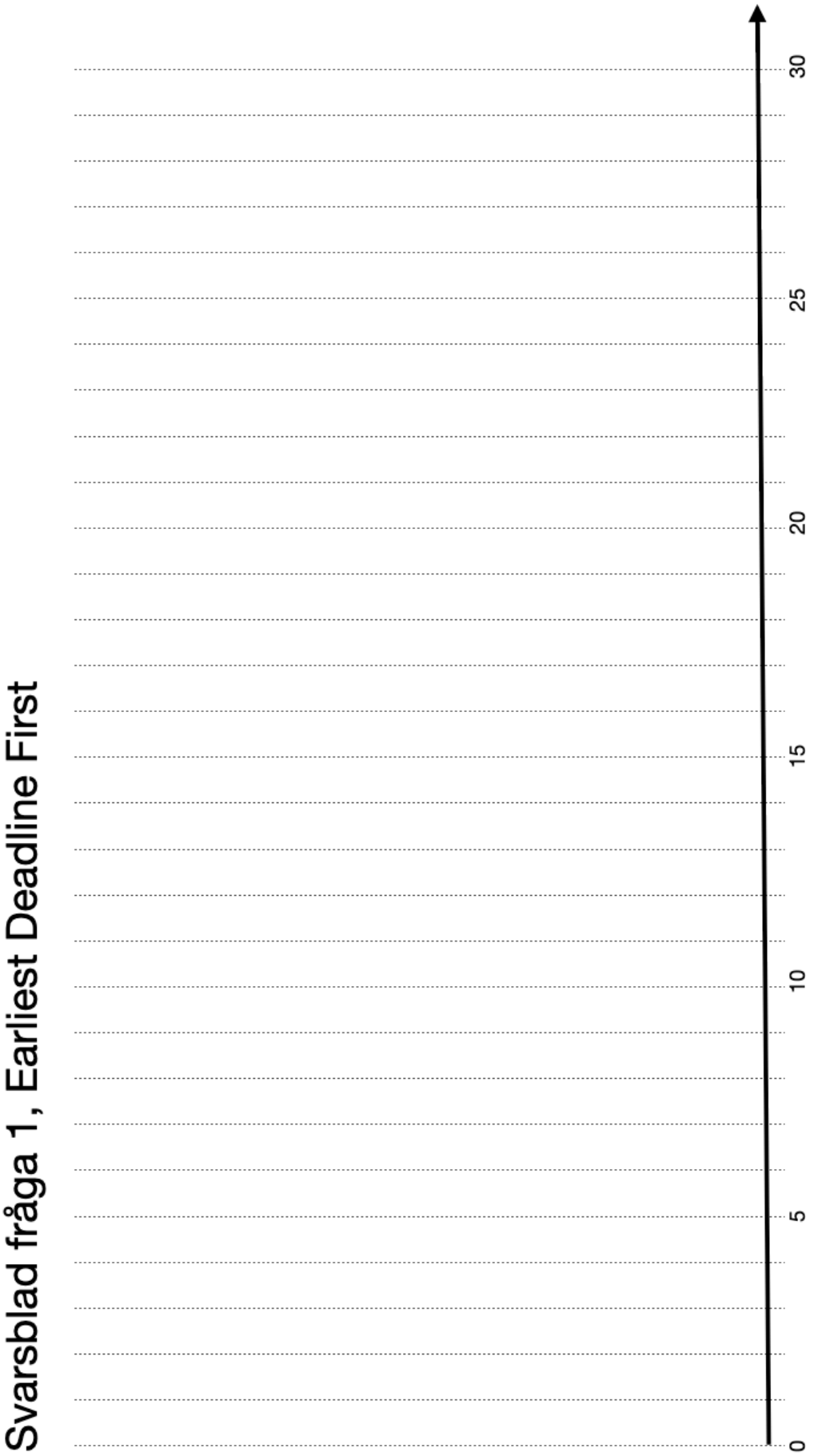
```
public class QueueSystem {

    public final int NO_OF_CLERKS = 5;
    public final int CUSTOMER_ID = 0;

    public CustomerHandler customerHandler;
    public ActorThread<Integer> coordinator;
    public ActorThread<DispData> displayHandler;
    public Hardware hw;

    public static void main(String [] args) {
        new QueueSystem().run();
    }
```

```
    public void run() {
        hw = new Hardware();

        customerHandler = new CustomerHandler(this);
        displayHandler = new DisplayHandler(this);
        coordinator = new Coordinator(this);

        customerHandler.start();
        displayHandler.start();
        coordinator.start();
        for (int i=1;i<=NO_OF_CLERKS;i++) {
            new ClerkHandler(this,i).start();
        }

    }
}

class DispData {
        public int ticket;
        public int counter;
}

// Thread handling printing queue slips when customer arrives
class CustomerHandler extends ActorThread<Integer> {
    private QueueSystem queue;

    public CustomerHandler(QueueSystem queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            while (true) {
                queue.hw.waitCustomerArrived();
                queue.coordinator.send(queue.CUSTOMER_ID);
                queue.hw.printTicket(receive());
            }
        } catch(InterruptedException e) {
            throw new Error("Unexpected interrupt");
        }
    }
}

// Threads marking clerks free
class ClerkHandler extends Thread {
    private QueueSystem queue;
    private int id;

    public ClerkHandler(QueueSystem queue,int id) {
        this.queue = queue;
        this.id = id;
    }

    public void run() {
        while (true) {
            queue.hw.waitClerkButton(id);
            queue.coordinator.send(id);
        }
    }
}

// Thread handling the display
class DisplayHandler extends ActorThread<DispData> {
    private QueueSystem queue;

    public DisplayHandler(QueueSystem queue) {
        this.queue = queue;
    }
```

```
    public void run() {
        try {
            while (true) {
                DispData d = receive();
                queue.hw.display(d.ticket%100,d.counter);
                sleep(10000);
            }
        } catch(InterruptedException e) {
            throw new Error("Unexpected interrupt");
        }
    }
}

// Thread pairing customers with free clerks
class Coordinator extends ActorThread<Integer> {
    private QueueSystem queue;

    public Coordinator(QueueSystem queue) {
        this.queue = queue;
    }

    public void run() {
        // Suggested data - use them if you like
        int lastCustomerArrived = 0;  // Last number printed on slip
        int lastCustomerServed = 0;   // Last customer assigned to a clerk
        int lastClerkAssigned = 0;    // Number of clerk last assigned a customer
        int nrOfFreeClerks = 0;       // Number of currently free clerks
        boolean[] clerkFree = new boolean[queue.NO_OF_CLERKS]; // Clerk status

        /* INSERT YOUR CODE HERE */
     }
}
```

ActorThread – specifikation

```
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;
}
```


Svarsblad fråga 1, Earliest Deadline First
0
5
10
15
20
25
30