Tentamen
EDAF85 – Realtidssystem

2024-10-25, 14.00–19.00

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 två *programmeringsuppgifter*, fråga 6–7 (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

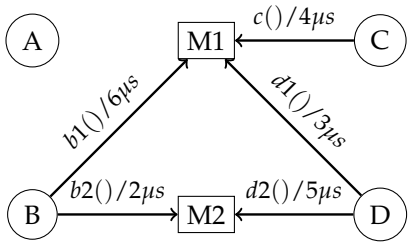
$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

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

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

1. Förklara kortfattat skillnaden mellan schemaläggning enligt principen Deadline Monotonic Scheduling (DMS) och Earliest Deadline First (EDF). *(1p)*

2. Ett realtidssystem med dynamisk prioritetsbaserad schemaläggning och prioritetsarv (basic inheritance protocol) består av fyra periodiska trådar (A, B, C och D). Trådarna A, B, C och D kommunicerar med varandra genom att anropa monitoroperationerna `b1()`, `b2()`, `c()`, `d1()` och `d2()` i monitorerna M1 och M2 och med maximala exekveringstider (mikrosekunder) 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. Notera att tråd A inte gör några monitoranrop och låser alltså aldrig någon monitor. Vi förutsätter att trådarna exekverar på en dator med en CPU-kärna.



a) 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 trådarnas blockeringsfaktorer, B_i). *(3p)*

b) Antag att vi slår ihop monitorerna M1 och M2 till en enda monitor, kallad M, men behåller alla monitoranrop som de är. Dvs att tråd B kommer att anropa först `b1()` och därefter `b2()` i den nya monitorn M. På samma sätt anropar tråden D först `d1()` och sedan `d2()` i monitorn M. Vad blir trådarnas blockeringsfaktorer, B_i, nu? *(2p)*

3. Ett realtidssystem består av fyra periodiska trådar med maximal exekveringstid (C), periodtid (T), blockeringsfaktor (B) och deadline (D) enligt nedanstående tabell. Vi antar att direkt prioritetsarv tillämpas. Trådarna schemaläggs enligt principen deadline monotonic scheduling (DMS) och körs på en dator med en CPU-kärna.

	C (ms)	T (ms)	B (ms)	D (ms)
A	1,5	5	0,6	3
B	4	12	0,3	10
C	2	24	0	24
D	2	10	0,4	4

Ange för varje tråd vad dess värstafallssvarstid (R) blir. *(3,5p)*

4. Betrakta följande tre javaklasser:

```
public class A {
    private Semaphore mutex = new Semaphore(0);

    public void print1() {
        mutex.acquire();
        System.out.println("1");
        print2();
        mutex.acquire();
    }

    public void print2() {
        mutex.acquire();
        System.out.println("2");
        mutex.release();
    }
}

public class B {
    private Semaphore mutex = new Semaphore(1);

    public void print1() {
        mutex.acquire();
        System.out.println("1");
        print2();
        mutex.acquire();
    }

    public void print2() {
        mutex.acquire();
        System.out.println("2");
        mutex.release();
    }
}

public class C {
    private Lock mutex = new ReentrantLock();

    public void print1() {
        mutex.lock();
        System.out.println("1");
        print2();
        mutex.unlock();
    }

    public void print2() {
        mutex.lock();
        System.out.println("2");
        mutex.unlock();
    }
}
```

a) Ange vad som skrivs ut om vi utför satsen

`new A().print1();`

Motivera ditt svar. *(0,5p)*

b) Ange vad som skrivs ut om vi utför satsen

`new B().print1();`

Motivera ditt svar. *(0,5p)*

c) Ange vad som skrivs ut om vi utför satsen

`new C().print1();`

Motivera ditt svar. *(0,5p)*

5. Betrakta följande `main()`-metod. Metoderna useRT(), useSTX(), useTR(), useW(), useWT(), useX() och useXW() visas inte, men utför inga operationer på några lås. Däremot förutsätter de att de i namnen indikerade låsen är tagna när de anropas. Om man kör programmet märker man att det då och då råkar ut för dödläge (deadlock).

```
1 public static void main(String[] args) {
2
3     Semaphore x = new Semaphore(1);
4     Semaphore w = new Semaphore(1);
5     Semaphore s = new Semaphore(1);
6     Semaphore t = new Semaphore(1);
7     Semaphore r = new Semaphore(1);
8
9     new Thread(() -> {  // Thread A
10        try {
11            while (true) {
12                w.acquire();
13                useW();
14                t.acquire();
15                useWT();
16                w.release();
17                r.acquire();
18                useTR();
19                r.release();
20                t.release();
21            }
22        } catch (InterruptedException e) {
23            throw new Error();
24        }
25    }).start();
26
27    new Thread(() -> {  // Thread B
28        try {
29            while (true) {
30                t.acquire();
31                r.acquire();
32                useRT();
33                r.release();
34                t.release();
35                x.acquire();
36                useX();
37                w.acquire();
38                useXW();
39                w.release();
40                x.release();
41            }
42        } catch (InterruptedException e) {
43            throw new Error();
44        }
45    }).start();
46
47    new Thread(() -> {  // Thread C
48        try {
49            while (true) {
50                s.acquire();
51                t.acquire();
52                x.acquire();
53                useSTX();
54                x.release();
55                t.release();
56                s.release();
57            }
58        } catch (InterruptedException e) {
59            throw new Error();
60        }
61    }).start();
62 }
```

a) Rita en resursallokeringsgraf för programmet. *(1,5p)*

b) Ange för varje tråd radnumret för den operation tråden försöker exekvera när den hamnar i dödläge. *(1,5p)*

c) Föreslå en ändring av programmet som gör att det inte längre riskerar att hamna i dödläge. Programmet ska även efter ändringen anropa funktionerna av typen `useXY()` i samma ordning som tidigare och med samma semaforer låsta som tidigare vid anropstillfällena. *(1p)*

6. Trådbarriär med monitor

En *barriär* används för att synkronisera n trådar på så sätt att alla trådarna har kommit fram till barriären innan de tillåts fortsätta. En tråd som anropar `await()` i barriären blockerar och fortsätter inte exekvera förrän alla n trådarna anropat `await()`. Vi skulle kunna beskriva det som ”vi väntar tills hela gruppen anlänt och sedan fortsätter vi tillsammans”.

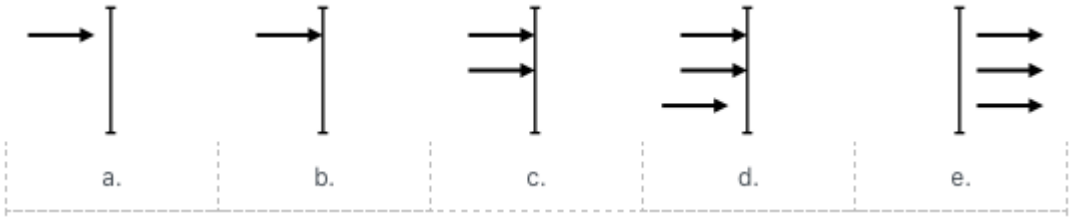
En mer formell beskrivning av barriären ser ut så här:

- En `Barrier` skapas med n som parameter till konstruktorn. Vi antar att $n \geq 2$.
- De $n - 1$ första trådarna som anropar `await()` blockerar.
- När den n:te tråden anropar `await()` händer följande:
  1. Alla n trådar lämnar barriären. Dvs att de kör vidare efter anropet av `await()`.
  2. Barriären återställs och är redo för en ny cykel med n trådar.
- Värdet som returneras av `await()` är ett heltal k, $0 \leq k < n$, som anger ordningen i vilken trådarna anlände till barriären. Den första tråden som anropade `await()` kommer att få $k = 0$ och den sista $k = n - 1$.

Nedanstående figur illustrerar detta för en barriär med $n = 3$. Den vertikala linjen representerar barriären, pilarna representerar trådarna och en pil som slutar precis på barriären representerar en tråd som blockerat i anropet av `await()`.

När den första tråden anländer (a) blockerar den (b) liksom nästa tråd (c). När den tredje tråden anländer (d) så blocker den *inte* (eftersom $n = 3$). I stället kommer alla tre trådarna släppas loss och är fria att passera barriären (e). Med andra ord: alla tre trådarna returnerar från anropet av `await()`.

Konceptuellt så returnerar trådarna från sina anrop av `await()` i princip samtidigt så som (e) antyder. I praktiken beror det exakta beteendet naturligtvis på hur operativsystemet väljer att schemalägga trådarna. Det innebär att vi måste ta hänsyn till vissa specialfall, vilket vi beskriver i nästa stycke.



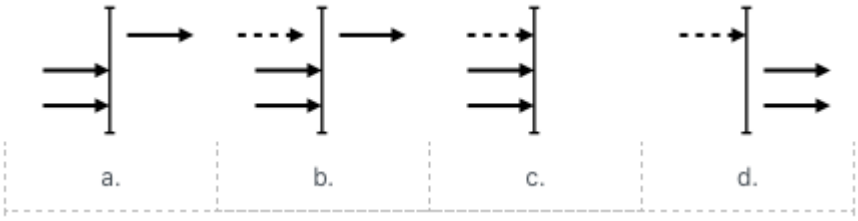
Barriärcykler

Texten ovan beskriver en cykel för en `Barrier`. När en cykel tar slut lämnar de n trådarna barriären och den görs redo för nästa cykel om n anländande trådar. Men, som vi berättade tidigare, en tråd kan ju bli fördröjd när den lämnar barriären beroende på operativsystemets schemaläggningsval. Detta innebär att trådar som tillhör nästa (nyss startade) cykel kan anropa `await()` innan trådarna som tillhör den förra (nyss avslutade) cykeln har hunnit lämna anropet av `await()`.

Betrakta följande scenario som illustreras av figuren på nästa sida: Antag att en cykel tar slut och en tråd har hunnit lämna anropet av `await()` medan två andra trådar inte ännu hunnit göra det (a). Antag vidare att en ny tråd (tillhörande nästa barriärcykel) anländer till barriären (anropar `await()`). Tråden är streckad i figuren nedan. Denna nya tråd ska blockera (c), men de övriga två trådarna ska tillåtas lämna barriären så fort som möjligt (d).

Detta innebär att det inte räcker med att hålla reda på antalet trådar i vår `Barrier`. Vi skulle i så fall felaktigt kunna dra slutsatsen att alla tre trådarna i steg (c) skulle släppas loss eftersom $n = 3$. För att hantera denna situation korrekt på enklaste sätt kommer du också att behöva en räknare (ett heltal) som håller reda på vilken cykel i ordningen som är den aktuella och se till att varje enskild

tråd bara väntar i barriären så länge inte räknarens värde har förändrats sedan tråden anropade `await()`.



Uppgiften
Implementera klassen `Barrier` med hjälp av Javas inbyggda stöd för monitorer (`synchronized`). En påbörjad lösning återfinns nedan. Komplettera/ändra den påbörjade lösningen så att den fungerar enligt beskrivningen ovan.

```
public class Barrier {

    // ...

    public Barrier(int n) {
        if (n < 2) {
            // a barrier for fewer than
            // 2 threads makes no sense
            throw new IllegalArgumentException();
        }

        // ...
    }

    public int await() throws InterruptedException {
        // ...
        return -7;  // you will need to change this
    }
}
```

En tråd kan stoppas genom ett anrop av `interrupt()` men vi ställer inga krav angående vilket tillstånd vår barriär ska hamna i efter att en tråd avbrutits på detta sätt. Med andra ord behöver du inte hantera detta fall på annat sätt än genom att se till att du *inte* fångar eventuella `InterruptedException` utan låter de kastas vidare från anropet av `await()` (metoden är ju deklarerad `throws InterruptedException`).

Tips

- Använd ett attribut av typen `int` för att hålla reda på vilken som är den aktuella cykeln.
- Kom ihåg att en lokal variabel i en metod också är lokal (unik) för den enskilda trådinstansen.
- De två kodexemplen på nästa sida illustrerar hur en `Barrier` skulle kunna användas i praktiken. De är medtagna för att hjälpa dig förstå hur en `Barrier` förväntas fungera, men spelar annars ingen roll i uppgiften.

(10p)

Trådbarriär, kodexempel 1

Bildbehandling på en 8-kärnig dator. 8 trådar arbetar parallellt med en bild. Varje tråd behandlar 1/8 av bilden. Vi använder returvärdet från `await()` för att se till att endast en tråd anropar `displayImage()`.

```
private final int[][] v = getImage();

// ...

Barrier b = new Barrier(8);
for (int i = 0; i < 8; i++) {                   // 8 threads
    int chunkSize = v.length / 8;
    int start = chunkSize * i;
    new Thread(() -> {
        try {
            processImage(v, start, chunkSize); // each thread processes 1/8 of the image
            int k = b.await();                  // all threads wait until all threads done
            if (k == 0) {                       // one thread displays results
                displayImage(v);
            }
        } catch (InterruptedException e) {
            // ...
        }
    }).start();
}
```

Trådbarriär, kodexempel 2

Detta exempel påminner om kodexempel 1, men här upprepas bildbehandlingen i en loop.

```
private volatile int[][] v = getFirstImage();

// ...

Barrier b = new Barrier(8);
for (int i = 0; i < 8; i++) {
    int chunkSize = v.length / 8;
    int start = chunkSize * i;
    new Thread(() -> {
        try {
            while (true) {
                processImage(v, start, chunkSize);
                int k = b.await();              // all threads wait until image completed
                if (k == 0) {
                    displayImage(v);
                    v = getNextImage();         // fetch data for next image
                }
                b.await();                      // all threads wait for next image
            }
        } catch (InterruptedException e) {
            // ...
        }
    }).start();
}
```

7. Trådbarriär med semaforer

Denna uppgift är en lite svårare uppgift som i första hand är till för dig som vill ha överbetyg (4/5) på kursen. Du bör först lösa den föregående uppgiften innan du börjar på denna uppgift.

I den förra uppgiften var problemet att implementera klassen `Barrier` med hjälp av Javas inbyggda stöd för monitorer, dvs med hjälp av `synchronized` och därtill hörande operationer. Den här gången ska du implementera samma klass `Barrier`, men denna gång enbart med hjälp av semaforer (`Semaphore` och/eller `Lock/ReentrantLock`).

Vi utgår återigen från samma påbörjade implementation:

```
public class Barrier {

    // ...

    public Barrier(int n) {
        if (n < 2) {
            // a barrier for fewer than
            // 2 threads makes no sense
            throw new IllegalArgumentException();
        }

        // ...
    }

    public int await() throws InterruptedException {
        // ...
        return -7;  // you will need to change this
    }
}
```

Komplettera med attribut i klassen och fyll i kod för konstruktorn och metoden `await()`.

Tips

- Använd *enbart* semaforer/lås för att synkronisera trådarna.
- Det går bra att lägga till egna privata metoder om du skulle vilja.
- Använd lämpligen en för trådarna gemensam semafor för att vänta på att den sista tråden ska anlända.
- Låt i så fall också varje generation av trådar (varje barriärcykel) ha sin egen unika semafor och skapa en ny när nästa cykel börjar. På så sätt undviker du att nyanlända trådar blandas ihop med trådarna från den förra cykeln.
- Kom ihåg att lika många trådar kan passera en semafors `acquire()` som antalet körtillstånd man tillfört semaforen (jämför bufferten från övning 1).

(5p)