Tentamen
EDAF85 – Realtidssystem

2024-04-11, 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 1-4 (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 (5) på varje del för sig.

Formelsamling

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

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

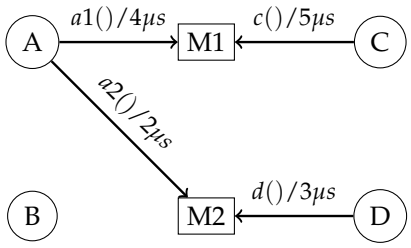
$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

1. Ange för vart och ett av nedanstående fyra påståenden om det är sant eller falskt.

1. Alla synkroniseringsproblem som du kan lösa med en semafor i Java kan du också lösa med en monitor.
2. Ett anrop av `notify()` eller `notifyAll()` i en synkroniserad monitormetod måste i Java ske *efter* att ett monitorattribut ändrats. Om det sker *före* ändringen så skulle en annan tråd, om vi har otur med schemaläggningen, kunna missa ändringen av attributet och läsa dess gamla värde.
3. Om en tråd anropar en monitormetod (metod deklarerad `synchronized`) i ett objekt betyder det att en annan tråd som anropar en *annan* monitormetod i *samma* objekt blir blockerad tills den första tråden exekverat klart sin monitormetod.
4. I en väl designad monitor i Java är alla attribut deklarerade `private` (eller möjligen `protected`) och alla publika metoder är deklarerade `synchronized`.

(2p)

2. Ett realtidssystem med dynamisk prioritetsbaserad schemaläggning och prioritetsarv (basic inheritance protocol) består av fyra periodiska trådar (A, B, C och D). Trådarna A, C och D kommunicerar med varandra genom att anropa monitoroperationerna `a1()`, `a2()`, `c()`, och `d()` i monitorerna M1 och M2 och med maximala exekveringstider (mikrosekunder) enligt nedanstående figur. Monitormetoderna anropas en i taget (inga nästlade anrop). B kommunicerar inte med de andra trådarna. A har högst prioritet, B näst högst, C näst lägst och D har lägst prioritet. 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 deras blockeringsfaktorer, B_i). *(3p)*

b) Antag att vi slog ihop de två monitorerna M1 och M2 till en enda, gemensam, monitor (med oförändrade monitoranrop). Skulle någon av trådarna i så fall få en *längre* blockeringstid i värsta fall? Ange i så fall vilken/vilka tråd/trådar det gäller. *(1p)*

c) Under samma förutsättning med en hopslagen monitor, skulle någon av trådarna i så fall få en *kortare* blockeringstid i värsta fall? Ange i så fall vilken/vilka tråd/trådar det gäller. *(1p)*

d) Hur skulle en hopslagning av de två monitorerna påverka systemets totala CPU-användning (*eng. CPU utilization*)? Skulle den bli högre, lägre eller oförändrad? *(1p)*

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) och körs på en dator med en CPU-kärna.

	C (ms)	T (ms)	B (ms)
A	1	8	0,2
B	3	10	0
C	2	4	0,5

a) Ange vad systemets CPU-utnyttjande (*eng. CPU utilization*) är. *(1p)*

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

c) Är systemet schemaläggningsbart. Du måste ange en korrekt motivering for att få poäng. *(1p)*

4. Programmet nedan råkar då och då ut för dödläge.

a) Rita en resursallokeringsgraf som beskriver programmet. *(1p)*

b) Ange för var och en av de två trådarna, `t1` och `t2`, den sista textraden som skrivs ut när programmet råkar ut för dödläge. *(1p)*

```
public class Deadlock {
    private final X x = new X();
    private final Y y = new Y();
    private final Z z = new Z();

    class X {
        public synchronized void m1() {
            System.out.println("A");
            y.m3();
            System.out.println("B");
        }
    }

    class Y {
        public synchronized void m2() {
            System.out.println("C");
        }

        public synchronized void m3() {
            System.out.println("D");
            z.m5();
            System.out.println("E");
        }
    }

    class Z {
        public synchronized void m4() {
            System.out.println("F");
            y.m2();
            System.out.println("G");
        }

        public synchronized void m5() {
            System.out.println("H");
        }
    }

    private void startThreads() {
        Thread t1 = new Thread(() -> z.m4());
        Thread t2 = new Thread(() -> x.m1());
        t1.start();
        t2.start();
    }

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

5. SimpleReentrantLock

I den här uppgiften ska du implementera en förenklad variant av låsklassen `ReentrantLock` som finns i Javas standardbibliotek. Din klass ska ha följande egenskaper:

Ömsesidig uteslutning (mutual exclusion) Endast en tråd i taget kan ta låset. Om låset har tagits av lås A och en annan tråd B anropar `lock()` så kommer B att få vänta tills A låser upp låset (genom anrop av `unlock()`).

Ägandeskap (ownership) *Ägaren* till ett lås är den tråd som för tillfället har tagit låset. Om låset är ledigt saknas ägare.

När en tråd tar låset blir tråden den aktuella ägaren till låset. När låset så småningom låses upp och i sin tur tas av en annan tråd övergår ägandet till den nya tråden.

Ett lås kan endast låsas upp av den tråd som för närvarande äger låset. Om `unlock()` anropas av en tråd som *inte* är den nuvarande ägaren till låset kastas `IllegalMonitorStateException`.

Rekursiva anrop (reentrancy) Tråden som är ägare till ett lås kan anropa `lock()` och därigenom ta låset igen upprepade gånger utan att ha anropat `unlock()` däremellan. Tråden förblir ägaren av låset (och låset förblir taget) tills ägaren anropat `unlock()` lika många gånger som låset har tagits. Se följande exempel:

```
SimpleReentrantLock l = new SimpleReentrantLock();
l.lock();
l.lock();    // The owner thread takes the lock again: locking count is 2
l.lock();    // The owner thread takes the lock again: locking count is 3
// ... critical region ...
l.unlock();  // The lock remains locked: locking count is 2
l.unlock();  // The lock remains locked: locking count is 1
l.unlock();  // The lock is free: another thread can now lock it
```

Timeout Klassen ska, förutom metoderna `lock()` och `unlock()` även ha en metod `tryLock()` som fungerar som `lock()` fast med den skillnaden att den genererar en timeout om låset inte kunde tas inom ett givet antal millisekunder (anges som parameter). Returvärdet är antingen `true` (om låset är taget utan timeout) eller `false` (om timeout skedde och låset inte kunde tas). Metoden `tryLock()` är tänkt att användas enligt följande mönster:

```
SimpleReentrantLock l = new SimpleReentrantLock();
// ...
if (l.tryLock(50)) {
    // lock taken: run critical region
    l.unlock();
} else {
    // lock unavailable, and did not become
    // available within 50 ms: do something else
}
```

Uppgiften

Implementera `SimpleReentrantLock` med hjälp av Javas monitorbegrepp (`synchronized` etc.) enligt följande anvisningar:

- `SimpleReentrantLock` ska implementera följande interface (`IllegalMonitorStateException` är ett s.k. *unchecked error* och metoder som kan kasta ett sådant, som `unlock()`, behöver inte deklarera det):

```
interface SimpleLock {
    void lock() throws InterruptedException;
    void unlock();
    boolean tryLock(long timeout) throws InterruptedException;
}
```

- När en tråd väntar i `lock()/tryLock()` ska det gå att avbryta det genom att anropa `interrupt()` på tråden. Anropet av `lock()` eller `tryLock()` ska då kasta ett `InterruptedException` och låset ska lämnas i ett konsistent tillstånd. Det betyder att låset ska ha samma tillstånd det skulle ha om anropet av `lock()/tryLock()` aldrig hade skett.
- Anropa `Thread.currentThread()` för att ta reda på vilken tråd som är den just nu körande.
- För att kasta ett `IllegalMonitorStateException` kan du skriva:

```
throw new IllegalMonitorStateException();
```

- Din lösning får *inte* använda standardklassen `ReentrantLock` eller några andra klasser från `java.util.concurrent`.

(10p)

6. 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`.

Uppgifterna för `Coordinator` blir alltså:

- Ta emot information om lediga kassor från `ClerkHandler` och spara denna.
- Ta emot beställning på ett nytt könummer från `CustomerHandler` och svara med ett meddelande som talar om vilket könummer som ska skrivas ut.
- Om något av ovanstående leder till att det både finns en ledig kassa och en kund som inte blivit betjänad redan så ska dessa paras ihop genom att ett `DispData`-objekt fylls i och skickas till `DisplayHandler`.

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 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;
}
```