LUNDS TEKNISKA HÖGSKOLA Institutionen för datavetenskap

Tentamen
EDAF85 – Realtidssystem

2023-08–24, 14.00–19.00

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 1-5 (15 poäng), och *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

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

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

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

1. Nedanstående loop är avsedd att skriva ut aktuell tid (i mikrosekunder) precis en gång per sekund:

```
while (true) {
    long t = System.currentTimeMillis();
    System.out.println(t);
    Thread.sleep(1000);
}
```

I praktiken så visar det sig dock att skillnaden mellan de utskrivna värdena är större än 1000 millisekunder (ibland betydligt större). Ange två olika skäl till denna avvikelse från det avsedda resultatet.

(2p)

2. Kommer du ihåg hissimuleringen från laboration 2? Vi hade ett antal passagerartrådar och en hisstråd som kommunicerade med varandra via en delad monitor. Den enda synkroniseringen mellan trådarna utgjordes av monitorn.

I denna variant av simuleringen används en enkel hissalgoritm (betydligt enklare än i din egen lösning av laborationen): Hissen rör sig från en våning till nästa, öppnar dörren, väntar en bestämd tid, stänger dörren och rör sig vidare till nästa våning. Med andra ord så öppnar hissen dörren på varje våning oavsett om någon passagerare vill gå på den eller inte. Passagerarna anländer, precis som i laborationens variant, till hissen vid slumpmässiga tidpunkter.

Antag vidare att hissmonitorn innehåller följande monitormetod:

```
/** Wait until the lift stands still, with open doors at floor f. */
public synchronized void awaitLiftStillAndOpenAtFloor(int f) { ... }
```

Alla passagerartrådar exekverar samma kod som bland annat innehåller följande rader:

```
Passenger passenger = view.createPassenger();
int f = passenger.getStartFloor();

// ...

passenger.begin();            // animates passenger walking in from left
monitor.awaitLiftStillAndOpenAtFloor(f);
passenger.enterLift();        // animates passenger entering lift

// ...
```

Ibland kommer passagerarna dock att gå in i en stängd dörr, eller en dörr som håller på att stängas.

a) Vad brukar man kalla ett fel av denna typ? *(1p)*

b) Förklara vad som händer när felet inträffar. *(1p)*

c) Skissa på (beskriv översiktligt) hur monitorn och/eller koden i passagerartråden skulle behöva förändras för att undvika att felet inträffar. Du kan svara i textform och/eller med pseudokod. Tänk på att man vill att flera passagerare ska kunna se ut att röra sig samtidigt på skärmen (dvs att flera anrop av `begin()`/`enterLift()` ska kunna exekvera samtidigt i olika passagerartrådar. *(2p)*

3. Ett realtidssystem består av tre periodiska trådar som exekverar oberoende av varandra (ingen inbördes blockering) och med maximal exekveringstid (C), periodtid (T) och CPU-utnyttjandegrad (C/T) enligt nedanstående tabell.

	C (ms)	T (ms)	C/T
T1	3	7	$\leq 0,429$
T2	2	26	$\leq 0,077$
T3	2	11	$\leq 0,182$

a) Ange prioritetsordningen (från högst till lägst) för trådarna om vi använder oss av rate monotonic scheduling (RMS). *(1p)*

b) Kommer systemet att klara alla sina deadlines? Vi antar RMS och att deadline för varje tråd är lika med dess periodtid. För att få poäng måste du motivera ditt svar. *(1p)*

4. Ett realtidssystem innehåller tre trådar, T1, T2 och T3. De exekverar bland annat nedanstående kodsekvenser i sina respektive `run()`-metoder. S1, S2, S3, S4 och S5 är alla mutexsemaforer i form av objekt av typen `ReentrantLock`. Alla semaforer är olåsta från början. Anropen av typen `useXYZ()` representerar kod för vilken semaforerna X, Y och Z måste vara tagna (låsta).

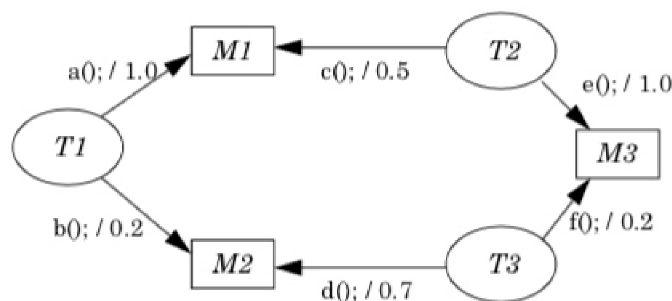
T1.run()	T2.run()	T3.run()
S5.lock();	S4.lock();	S1.lock();
useS5();	useS4();	useS1();
S5.unlock();	S5.lock();	S1.unlock();
S2.lock();	S1.lock();	S2.lock();
S5.lock();	useS1S4S5();	S3.lock();
useS2S5();	S1.unlock();	useS2S3();
S2.unlock();	S5.unlock();	S4.lock();
useS5();	S4.unlock();	useS2S3S4();
S5.unlock();	S2.lock();	S4.unlock();
S1.lock();	S3.lock();	S3.unlock();
S2.lock();	useS2S3();	S2.unlock();
useS1S2();	S3.unlock();	
S2.unlock();	S2.unlock();	
S1.unlock();		

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

b) Systemet kan hamna i dödläge. Motivera utifrån resursallokeringsgrafen varför det är så. *(1p)*

c) Föreslå en ändring av koden som gör att dödläge inte kan inträffa och som inte förändrar vilka semaforer som är låsta när respektive `useXYZ()`-metod anropas. *(1p)*

5. Ett realtidssystem med dynamisk prioritetsbaserad schemaläggning och prioritetsarv (basic inheritance protocol) består av tre periodiska trådar (T1, T2 och T3). De kommunicerar med varandra genom att anropa monitoroperationerna `a()`, `b()`, `c()`, `d()`, `e()` och `f()` i monitorerna M1, M2 och M3 och med maximala exekveringstider (millisekunder) enligt nedanstående figur. Monitormetoderna anropas en i taget (inga nästlade anrop). T1 har högst prioritet och T3 har lägst prioritet.



Ange för var och en av de tre trådarna (T1, T2 och T3) den maximala tid tråden kan bli blockerad av lägre prioriterade trådar under en och samma körning. *(3p)*

6. Klassen `DelayQueue` är en blockerande kö i vilken varje element är associerad med en fördröjning (*delay*): Om ett element läggs in i kön vid tiden t och ges fördröjningen d så definierar vi dess *släpptid* (*release time*) som $t + d$. Ett element kan inte tas ut ur kön förrän tidigast vid dess släpptid. Metoden `take()` blockerar tills kön innehåller åtminstone ett element vars släpptid har passerats, Alla tider mäts i millisekunder (som `System.currentTimeMillis()` returnerar dem).

Ett ofullständigt utkast till implementation av `DelayQueue` återfinns nedan. Den inre klassen kan användas för att lagra element tillsammans med dess släpptid. Attributet `q` är en `PriorityQueue` som används för att lagra `Nodes`-objekt i tidsordning.

```
public class DelayQueue<E> {

    private class Node implements Comparable<Node> {
        private final E element;
        private final long releaseTime;

        private Node(E element, long delay) {
            this.element = element;
            releaseTime = System.currentTimeMillis() + delay;
        }

        @Override
        public int compareTo(Node other) {
            return Long.compare(releaseTime, other.releaseTime);
        }
    }

    private final Queue<Node> q = new PriorityQueue<>();

    // you may add attributes if you want to (but you don't need to)

    /**
     * Adds an element with a delay to the queue. The element
     * will not be returned (in take() below) until 'delay'
     * milliseconds after it was added.
     */
    public void add(E e, long delay) {
        // TODO: implement this method
    }

    /**
     * Blocks until the queue contains at least one element whose
     * release time is met. If more than one such element is
     * available, the one with the earliest release time is returned.
     */
    public E take() throws InterruptedException {
        // TODO: implement this method
    }
}
```

Enkelt exempel på användning av klassen:

```
DelayQueue<String> dq = new DelayQueue<>();
dq.add("EDAF85",1000); // Element will be available after 1 second
try {
    System.out.println(dq.take()); // Prints "EDAF85" after 1 second
} catch(InterruptedException e) { System.err.println(e); System.exit(1); }
```

Din uppgift

Komplettera klassen `DelayQueue` med metoderna för att lägga till element (`void add(E e,long delay)`) respektive ta ut element (`E take()`).

Instruktioner och tips

- Använd Javas inbyggda monitorbegrepp för all synkronisering.
- Använd *inte* något från paketet `java.util.concurrent` (vilket inkluderar semaforer/lås och `PriorityBlockingQueue`).
- Du behöver bara svara med metoderna `add()` och `take()`.
- Vi förväntar oss inte att det ska behövas fler attribut eller privata hjälpmetoder, men det går bra att lägga till sådana om så önskas. Ta i så fall med även den/dem i ditt svar.
- Nedan finns en specifikation för `Queue` som visar metoder i klassen `PriorityQueue` som kan vara användbara. Observera att din klass `DelayQueue` *inte* ska implementera `Queue`!

```
/**
 * A collection designed for holding elements prior to processing.
 * Besides basic Collection operations, queues provide additional
 * insertion, extraction, and inspection operations.
 */
public interface Queue<E> extends Collection<E>, Iterable<E> {

  /**
   * Inserts the specified element into this queue.
   * Always returns true (as specified by Collection.add(E)).
   */
  boolean add(E e);

  /**
   * Retrieves, but does not remove, the head of this queue.
   * This method differs from peek() only in that it throws an
   * exception if this queue is empty.
   */
  E element();

  /**
   * Retrieves, but does not remove, the head of this queue,
   * or returns null if this queue is empty.
   */
  E peek();

  /**
   * Retrieves and removes the head of this queue,
   * or returns null if this queue is empty.
   */
  E poll();

  /**
   * Retrieves and removes the head of this queue.
   * This method differs from poll() only in that it throws an
   * exception if this queue is empty.
   */
  E remove();
}
```

(8p)

7. I denna uppgift, som kan ses som en fortsättning på den föregående uppgiften, kan den klass, `DelayQueue`, som du där utvecklade komma till användning. Du behöver inte ha löst den föregående uppgiften för att kunna lösa denna uppgift, men du bör ha läst igenom den tidigare uppgiften så att du vet hur klassen `DelayQueue` fungerar.

Din uppgift är att skriva en klass `Dispatcher` vars uppgift är att se till att arbetsuppgifter utförs efter givna fördröjningar. Varje arbetsuppgift representeras av ett objekt som implementerar interfacet `Runnable` (man utför alltså arbetsuppgiften genom att anropa `run()` på objektet):

```
interface Runnable {
    public void run();
}
```

En ofullständig implementation av klassen `Dispatcher`, som du ska skriva, ser ut så här:

```
class Dispatcher {
    // TODO: put your own private declarations here

    /**
     * Constructor
     */
    public Dispatcher() {
        // TODO: implement this method
    }

    /**
     * Adds the task r to the dispatcher. The dispatcher will
     * automatically call the tasks run() method as soon as
     * possible after the delay (in milliseconds) given by delay.
     */
    public addTask(Runnable r,long delay) {
        // TODO: implement this method
    }
}
```

Följande program illustrerar hur klassen kan användas:

```
Dispatcher d = new Dispatcher();
d.addTask(()->{System.out.println("1");},3000);
d.addTask(()->{System.out.println("2");},1000);
d.addTask(()->{System.out.println("3");},4000);
d.addTask(()->{System.out.println("4");},2000);
...
```

Resultatet blir att det kommer att skrivas ut en rad i sekunden (ungefär) och i följande ordning:

```
2
4
1
3
```

Instruktioner

- Lägg till egna privata attribut, metoder och/eller trådar efter behov.
- Tänk på att en arbetsuppgift kan ta godtyckligt lång tid att utföra och därför måste utföras parallellt.
- Det skulle kunna hända att väldigt många arbetsuppgifter vill utföras precis samtidigt. För att inte överbelasta systemet bestämmer vi därför att högst fem (5) arbetsuppgifter får exekveras samtidigt. Övriga arbetsuppgifter vars fördröjning har gått ut får vänta tills någon av de redan startade arbetsuppgifterna kört färdigt (dess `run()`-metod har avslutats).
- Arbetsuppgifter utförs i den ordning deras fördröjningar tar slut. Om flera arbetsuppgifter blir körklara exakt samtidigt så har det ingen betydelse vilken arbetsuppgift som utförs först.

(7p)