Tentamen
EDAF85 – Realtidssystem

2026-04-09, 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 1-4 (15 poäng), och *programmeringsuppgifter*, fråga 5 och 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

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

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

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

1. En programmerare har skrivit följande rader kod i syfte att åstadkomma ömsesidig uteslutning över anropet av `doSomething()`:

```
while (mutex==1) { }    // Wait for mutex lock to become free
mutex = 1;              // Acquire mutex lock
doSomething();          // Perform task requiring mutual exclusion
mutex = 0;              // Release lock
```

Variabeln `mutex` är deklarerad så här: "`volatile int mutex = 0;`". Nyckelordet `volatile` innebär att alla trådar som använder variabeln ser samma värde och att ingen tråd behåller ett eget lokalt värde på variabeln som andra trådar inte ser (inga effekter av caching eller kodoptimering).

a) Vad brukar vi vanligtvis kalla detta sätt att vänta på att ett villkor blir uppfyllt (”`while (mutex==1) { }`”)? Vilken nackdel har denna metod? *(1p)*

b) Garanterar koden ovan ömsesidig uteslutning för anropen av `doSomething()`? Förklara varför det är så! *(1p)*

2. I ett Javaprogram hittar vi två typer (klasser) av trådar, T1 och T2, som i sina respektive run()-metoder exekverar följande linjära sekvenser av semaforoperationer på de fyra mutexsemaforerna A, B, C och D (mellanliggande kod som är beroende av semaforerna representeras av funktionsanropen på formen ”useXY();”, där ”XY” avser att semaforerna X och Y måste vara tagna när koden utförs):

```
    T1              T2
    ==              ==
 1. B.acquire();    B.acquire();
 2. C.acquire();    A.acquire();
 3. useBC();        useAB();
 4. D.acquire();    A.release();
 5. useBCD();       D.acquire();
 6. D.release();    useBD();
 7. C.release();    B.release();
 8. B.release();    A.acquire();
 9. A.acquire();    useAD();
10. C.acquire();    A.release();
11. useAC();        D.release();
12. C.release();
13. A.release();
```

Observera att vi kan ha ett godtyckligt antal instanser av varje trådtyp (klass).

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

b) Hur många instanser (trådobjekt) måste det minst finnas av vardera T1 och T2 och på vilka radnummer i koden ovan måste respektive trådinstans befinna sig på för att det ska finnas risk för dödläge i systemet? *(2p)*

c) Föreslå en enkel ändring av koden ovan som gör att dödläge garanterat inte kan uppstå i systemet, men som fortfarande garanterar att relevanta semaforer (och inga andra) är låsta då de olika ”useXY();”-funktionerna anropas. *(1p)*

3. Beskriv med en eller två meningar vardera innebörden av följande schemaläggningsbegrepp:

a) Strikt prioritetsbaserad, dynamisk, schemaläggning

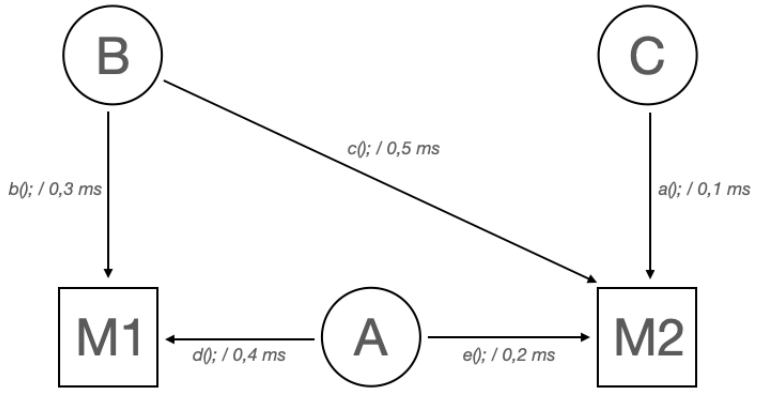
b) Earliest deadline first scheduling (EDF)

c) Rate monotonic scheduling (RMS)

d) Statisk schemaläggning

(2p)

4. Ett realtidssystem (med dynamisk prioritetsbaserad schemaläggning och prioritetsarv) innehåller tre periodiskt exekverande trådar (A, B och C) som kommunicerar med varandra genom att anropa monitoroperationerna a, b, c, d och e i monitorerna M1 och M2 och med maximala exekveringstider för operationerna (i millisekunder) enligt figur. Exempelvis visar figuren att tråd B anropar b() i monitorobjektet M1 en gång och metoden c() i monitorobjektet M2 en gång varje period (dock inte nödvändigt i den ordningen). Trådarna anropar en monitoroperation i taget, dvs att anropen inte är nästlade.



Vidare har trådarna värstafallsexekveringstider (C), periodtider (T) och deadlines (D) enligt följande tabell.

	C (ms)	*T (ms)*	*D (ms)*
A	3	15	15
B	2	5	5
C	4	20	10

För att tilldela trådarna prioriteter används principen DMS - Deadline Monotonic Scheduling.

a) Ange för varje tråd (A, B och C) hur lång tid tråden i värsta fall kan bli blockerad av lägre prioriterade trådar under en och samma körning. *(3p)*

b) Ange för varje tråd (A, B och C) vad deras värstafallsresponstid (R) blir. *(3p)*

5. AudioController

I laboration 3 skrev du programvaran som styrde en tvättmaskin. Vi ska nu lägga till ytterligare en kontrolltråd till tvättmaskinen: `AudioController`. Denna kontrolltråd är ansvarig för att generera ljud, till exempel när tvättprogrammet har avslutats.

Du kommer att använda klassen `Buzzer` för att styra högtalaren:

```
public class Buzzer {
    /** Place the loudspeaker membrane in the ON position. */
    public void on()              { ... }

    /** Place the loudspeaker membrane in the OFF position. */
    public void off()             { ... }
}
```

Hur man skapar ljud med hjälp av `Buzzer`

Man skapar ljud genom att snabbt växla högtalarens membran mellan på- och avlägena (ON respektive OFF). Ljudets frekvens beror på hur snabbt man växlar mellan lägena. För att skapa ett ljud med frekvensen 5 kHz måste högtalaren kopplas på och av 5000 gånger/sekund.

Följande kod skapar ett ljud vars frekvens är *nästan* 5 kHz. Frekvensen 5 kHz innebär en periodtid på 1/5000 = 0,0002 sekunder, eller 200000 nanosekunder. I detta exempel kommer ljudet att fortsätta låta tills programmet stoppas (`while (true) ...`).

```
Buzzer b = new Buzzer();
while (true) {
  b.on();
  TimeUnit.NANOSECONDS.sleep(100000);
  b.off();
  TimeUnit.NANOSECONDS.sleep(100000);
}
```

Notera två saker angående anropet av `sleep()` ovan:

Den speciella varianten `TimeUnit.NANOSECONDS.sleep(...)` innebär att fördröjningen mäts i nanosekunder (10^{-9} sekunder) istället för i millisekunder. Varje anrop av `sleep()` ger en fördröjning på en halv period om 100000 nanosekunder. Det går två halvperioder på en hel cykel: en med membranet i läge ON och en med membranet i läge OFF.

Varför *nästan* 5 kHz?

I exemplet ovan kommer varje anrop av `sleep()` ge en fördröjning som i praktiken är något mer än 100000 nanosekunder. Det finns flera orsaker till detta, som du kanske kommer ihåg från laboration 1 (alarmklockan). Effekten blir att ljudfrekvensen kommer att variera något och bli något lägre än 5 kHz i exemplet ovan.

AudioRequest-meddelanden

Du ska skriva en trådklass `AudioController` som förväntas kunna ta emot meddelanden av typen `AudioRequest`:

```
public class AudioRequest {
  /**
   * Returns the half-period (in nanoseconds) for the tone to play,
   * or zero (0) if no tone is to be played (silence).
   */
  public int getHalfPeriod() { ... }

  // ...
}
```

Ett meddelande av typen `AudioRequest` ska tolkas enligt följande:

- Ett `AudioRequest` med en halvperiod > 0 innebär att motsvarande ton ska spelas tills ett nytt meddelande anländer.
- Ett `AudioRequest` med en halvperiod = 0 innebär att högtalaren ska vara tyst tills nästa meddelande anländer.
- Du kan ignorera `AudioRequest` med en halvperiod < 0

Att mäta tid i nanosekunder

Du är säkert van vid att mäta tid i millisekunder i Java. I denna uppgift kommer vi att arbeta med nanosekunder.

Använd `System.nanoTime()` för att mäta tid.

`System.nanoTime()` fungerar precis som `System.currentTimeMillis()` fast tiden mäts i nanosekunder.

```
long t0 = System.nanoTime();
// ...
long t1 = System.nanoTime();
long dt = t1 - t0;     // time in nanoseconds
```

Du kan använda `TimeUnit.NANOSECONDS.sleep()` för att fördröja exekveringen ett antal nanosekunder. `TimeUnit.NANOSECONDS.sleep()` fungerar precis som `Thread.sleep()` förutom att tiden mäts i nanosekunder (se Buzzer-exemplet ovan).

Klassen `ActorThread`[1] har modifierats i denna uppgift så att metoden `receiveWithTimeout()` mäter tiden i nanosekunder:

```
public class ActorThread<M> extends Thread {
  /**
   * 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 'timeoutNano' nanoseconds if none available.
   * Returns null if no message is received
   * within 'timeoutNano' nanoseconds.
   */
  protected M receiveWithTimeout(long timeoutNano)
    throws InterruptedException      { ... }
}
```

Din uppgift

Du ska skriva en klass `AudioController` som är en subklass av `ActorThread`, och som ska kunna användas på samma sätt som de andra kontrolltrådarna i tvättmaskinslaborationen. `ActorThread` har, som beskrevs ovan, modifierats så att den arbetar med nanosekunder i stället för millisekunder.

[1] Till dig som läste kursen före 2020: När du gjorde laborationen ärvde kontrolltrådarna från klassen `RTThread` som tog emot meddelanden av typen `RTEvent`. Denna klass har i laborationen numera ersatts av `ActorThread` som tar emot meddelanden av typen given av `M`.

Följande första försök till lösning är given:

```
class AudioController extends ActorThread<AudioRequest> {
    public void run() {
        try {
            Buzzer buzzer = new Buzzer();
            while (true) {
                AudioRequest message = receive();

                int halfPeriod = message.getHalfPeriod();

                while (true) {
                    buzzer.on();
                    TimeUnit.NANOSECONDS.sleep(halfPeriod);
                    buzzer.off();
                    TimeUnit.NANOSECONDS.sleep(halfPeriod);
                }
            }
        } catch (InterruptedException unexpected) {
            throw new Error(unexpected);
        }
    }
}
```

Den givna klassen `AudioController` har två problem:

1. När den väl börjat spela en ton så reagerar den inte längre på meddelanden. Den fortsätter bara spela samma ton för evigt.
2. Tonen håller inte riktigt rätt frekvens utan är något för låg (se avsnittet ”Varför *nästan* 5 kHz?” ovan).

Modifiera klassen `AudioController` ovan så att...

1. .. den reagerar på nya meddelanden. När ett nytt meddelande anländer ska tråden börja spela tonen med den nya frekvens eller, om halvperioden är 0, upphöra med att spela tonen.
2. ...spelar en korrekt ton genom att korrigera för klockdriften.
3. ...inte konsumerar onödig CPU-kraft när inget ljud ska spelas.

Observera att till skillnad från de andra kontrolltrådarna i tvättmaskinslaborationen förväntas denna tråd *inte* skicka några bekräftelsemeddelanden på att den tagit emot ett nytt kommando (ACK). *(7p)*

6. CountDownLatch

En `CountDownLatch` kan vara användbar för signalering mellan trådar – särskilt när en eller flera trådar väntar på att något ska ha hänt ett visst antal gånger. En `CountDownLatch` är inte lika generell som till exempel en monitor eller en semafor, men i vissa applikationer kan den med fördel användas i stället för dessa.

Standardpaketet `java.util.concurrent` i Java innehåller en klass `CountDownLatch`. Dokumentationen för den klassen säger att en `CountDownLatch` är:

```
"A synchronization aid that allows one or more threads to wait until a set of
operations being performed in other threads completes. A CountDownLatch
is initialized with a given count. The await methods block until the current
count reaches zero due to invocations of the countDown() method, after
which all waiting threads are released and any subsequent invocations of
await return immediately. This is a one-shot phenomenon - the count cannot
be reset.

A CountDownLatch is a versatile synchronization tool and can be used for a
number of purposes. A CountDownLatch initialized with a count of one serves
as a simple on/off latch, or gate: all threads invoking await wait at the gate
until it is opened by a thread invoking countDown(). A CountDownLatch initialized
to N can be used to make one thread wait until N threads have completed some
action, or some action has been completed N times.

A useful property of a CountDownLatch is that it doesn't require that threads
calling countDown() wait for the count to reach zero before proceeding, it
simply prevents any thread from proceeding past an await until all threads
could pass."
```

Uppgift

Implementera din egen `CountDownLatch` enligt specifikationen nedan. Du får naturligtvis inte använda Javas standardklass `java.util.concurrent.CountDownLatch` i din lösning. Använd Javas inbyggda monitorbegrepp (`synchronized`/`wait()`/`notify()`) för dina synkroniseringsbehov.

Specifikationen nedan avviker något från Javas standardklass `java.util.concurrent.CountDownLatch`, men skillnaderna är oväsentliga i vårt fall.

```
1  public class CountDownLatch {
2
3      /**
4       * Constructs a CountDownLatch initialized with the given count.
5       *
6       * @param count  the number of times countDown() must be invoked
7       *               before threads can pass through await()
8       */
9      public CountDownLatch(int count) { ... }
10
11     /**
12      * Causes the current thread to wait until the latch has counted down to
13      * zero, unless the thread is interrupted.
14      *
15      * If the current count is zero then this method returns immediately.
16      */
17     public void await() throws InterruptedException { ... }
```

```
18      /**
19       * Causes the current thread to wait until the latch has
20       * counted down to zero, unless the thread is interrupted,
21       * or the specified waiting time elapses.
22       *
23       * If the current count is zero then this method returns immediately.
24       *
25       * @param timeout  maximal waiting time in milliseconds
26       *
27       * @return true if the count reached zero,
28       *         and false if the waiting time elapsed
29       *         before the count reached zero
30       */
31      public boolean await(long timeout) throws InterruptedException { ... }
32
33      /**
34       * Decrements the count of the latch, releasing all waiting
35       * threads if the count reaches zero.
36       */
37      public void countDown() { ... }
38  }
```

(8p)