Tentamen
EDAF85 – Realtidssystem

2025-08-28, 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. Förklara kortfattat, med hjälp av ett enkelt exempel, vad prioritetsinversion är och hur man kan göra för att undvika att det inträffar. *(2p)*

2. En javaklass som implementerar komplexa tal har två attribut och fyra metoder enligt följande:

```
class Complex {
   float re;
   float im;
   synchronized void setValue(float re, float im) {
       this.re = re; this.im = im;
   }
   float getRe() { return re; }
   float getIm() { return im; }
   float getAbs() { return Math.sqrt(re*re+im*im); }
}
```

Vad innebär det egentligen att nyckelordet `synchronized` används i deklarationen av metoden `setValue`? Anta att vi har två trådar T1 och T2 som båda har referenser till två objekt av klassen `Complex`, C1 och C2. Svara genom att för varje påstående nedan ange om det är sant eller falskt.

1. När T1 exekverar `C1.setValue(1.0,1.1)` så kommer ett anrop i T2 av `C2.setValue(2.0,2.2)` blockera tills T1 har avslutat anropet av `C1.setValue(1.0,1.1)`.
2. När T1 exekverar `C1.setValue(1.0,1.1)` så kommer ett anrop i T2 av `C1.setValue(2.0,2.2)` blockera tills T1 har avslutat anropet av `C1.setValue(1.0,1.1)`.
3. Eftersom `setValue` är `synchronized` så kan T2 inte exekvera metoden `getAbs` så länge T1 exekverar `setValue`.
4. Eftersom en variabel av typen `float` kan tilldelas/returneras atomärt (dvs utan risk för ett kontextbyte) så kan vi (som en sorts optimering, dock ej rekommenderat, och så som det är gjort i koden ovan) utelämna nyckelordet `synchronized` i deklarationen av `getAbs` utan att förändra realtidskorrektheten hos programmet (dvs att det inte finns någon risk för kapplöpning).

(2p)

3. 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();
```

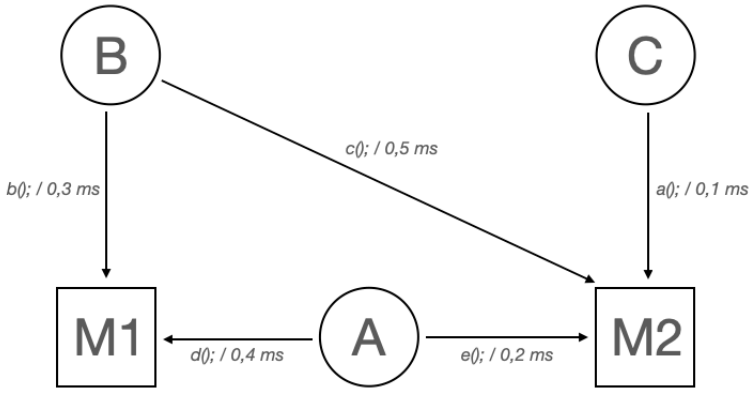
Observera att vi kan ha ett godtyckligt antal instanser av varje trådtyp (klass).

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? *(1p)*

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)*

4. Ett realtidssystem (med dynamisk prioritetsbaserad schemaläggning och prioritetsarv) innehåller tre periodiska 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. 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)*

c) Skulle vi kunna använda trådarnas CPU-utnyttjandegrad för att avgöra trådarna schemaläggningsbarhet? Motivera ditt svar – ett svar utan motivering ger ingen poäng. *(1p)*

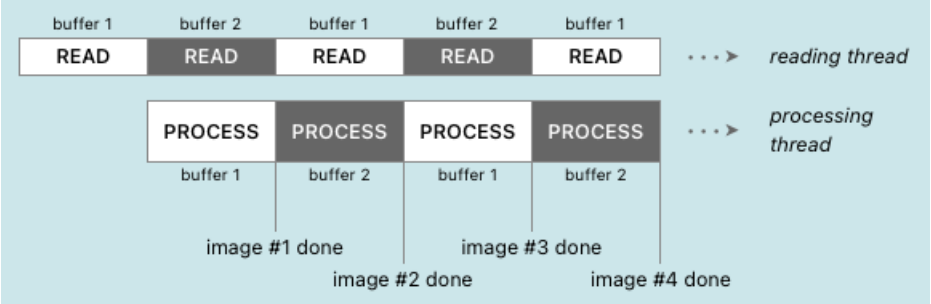
5. Dubbelbuffring med ActorThread

I programmet `VideoProcessing` nedan behandlas en videoström en bild i taget. Två operationer utförs för varje bild:

- först läses bilddata från en bildsensor till en databuffert (`read()`);
- sedan bearbetas bilden i bildbufferten och visas (`process()`).

Den givna koden kör de två operationerna sekventiellt för varje bild enligt följande:

Vi vill nu förbättra genomströmningen, dvs bildfrekvensen. De två operationerna måste fortfarande köras sekventiellt för varje bild men de kan köras parallellt för olika bilder. Vi behöver då två buffertar (vilket här illustreras med olika färger) för att utföra operationerna:



Den här tekniken kallas för dubbelbuffring. En tråd läser bilden till en bildbuffert och lämnar sedan över den till en annan tråd för behandling. En ny bild kan sedan läses in medan den andra tråden behandlar den föregående. Tiden det tar att behandla varje enskild bild är densamma som tidigare, men genomströmningshastigheten ökar.

Uppgift Förbättra genomströmningen genom att använda dubbelbuffring enligt beskrivningen ovan. Använd meddelandesändning (med hjälp av `ActorThread`) och två databuffertar.
Ytterligare vägledning:

- Inför `ActorThread`-trådar efter behov.
- Vi antar att `VideoProcessing` kör på en dator med minst två processorkärnor så att vi faktiskt kan utnyttja parallelliteten för att snabba upp programmet.
- Du får inte använda monitorer (`synchronized`), semaforer, lås eller några andra klasser från `java.util.concurrent`-paketet.
- Gör inga antaganden om hur lång tid `read()` eller `process()` tar att köra. Din lösning ska fungera även om exekveringstiden för de enskilda anropen varierar.

Du hittar specifikation för `ActorThread` och kod för `VideoProcessing` på nästa sida.

(8p)

ActorThread

```
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 { ... }
}
```

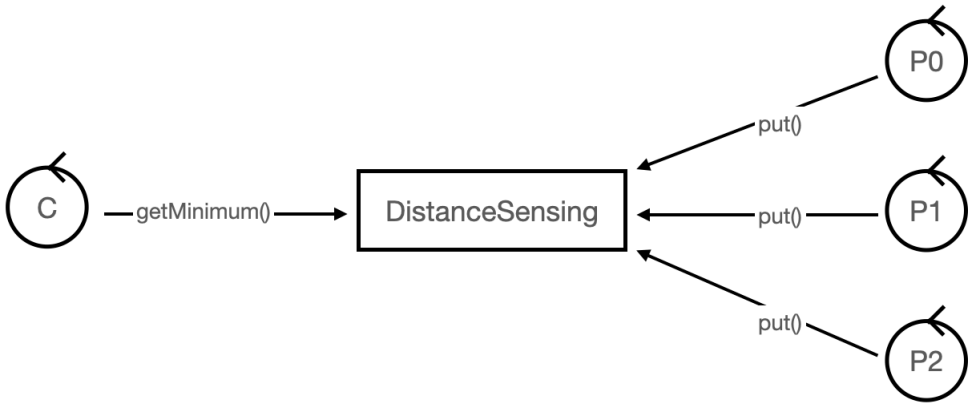
VideoProcessing

```
class VideoProcessing {

    ActorThread<DataBuffer> t1 = new ActorThread<>() {
        @Override
        public void run() {
            try {
                DataBuffer b = receive();
                while (true) {
                    b.read();       // read sensor data into buffer
                    b.process();    // process data in buffer and show on display
                }
            } catch (InterruptedException unexpected) {
                throw new Error(unexpected);
            }
        }
    };

    void startThreads() {
        t1.start();

        t1.send(new DataBuffer());
    }

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

6. Avståndsövervakning

I övervakningssystemet till en mobil robot ingår tre olika sensorer som mäter avståndet till omgivande föremål i olika riktningar. Sensorerna läses av av tre trådar kallade P0, P1 respektive P2. Beteckningarna matchar sensorernas numrering (0, 1 och 2). När sensortrådarna har läst av ett nytt värde (avstånd) lagras detta värde i en monitor, `DistanceSensing`, vilken det är din uppgift att implementera. Trådarna anropar metoden `put()` för att lagra ett nytt värde i monitorn. Utöver de tre sensortrådarna finns det en kontrolltråd, C, som regelbundet hämtar det minsta avståndet till omgivande föremål genom att anropa metoden `getMinimum()`. Att skriva de fyra trådarna ingår inte i uppgiften – bara att skriva monitorn.

Nedanstående figur sammanfattar relationen mellan trådarna, monitorn och subklassen som implementerar `dangerAlert()`.



Varje enskilt mätvärde, eller sampel, representeras av ett objekt av klassen `Sample`. Sådana objekt är tänkta att vara oföränderliga (immutable), så det går bra att dela sådana objekt mellan olika trådar utan att göra kopior på vägen. Objekten får automatiskt en tidsstämpel som talar om när de skapades.

```
class Sample {
    private int id;
    private double value;
    private long timestamp;

    /** Constructor */
    public Sample(int id, double value) {
        this.id = id;
        this.value = value;
        timestamp = System.currentTimeMillis();
    }

    /** Returns the id (0,1,2) of the sensor */
    public int getId() { return id; }

    /** Returns the sampled (distance) value. */
    public double getDistance() { return value; }

    /** Returns the sample timestamp. */
    public long getTimestamp() { return timestamp; }
}
```

Din uppgift är att implementera själva monitorklassen `DistanceSensing`:

```
public class DistanceSensing {

    /** Constructor */
    public DistanceSensing() {
        /* to be implemented */
        ...
    }

    /** Stores a new sensor value in the monitor.
        Old sensor value from the same sensor is
        overwritten. This method never blocks. */
    public  void put(Sample s) {
        /* to be implemented */
        ...
    }

    /** Blocks until sensor values arrived later than time
        (as measured by System.currentTimeMillis())
        are available from all three sensors and then returns
        the sample with smallest value. If any sensor does
        not have, or receives, a sufficiently recent sensor
        value (having arrived later than time) within
        timeout milliseconds after getMinimum() being called,
        null is returned.
    */
    public  Sample getMinimum(long time, long timeout) {
        /* to be implemented */
        ...
    }
}
```

Beskrivning av monitoroperationerna

`public void put(Sample s)` – Varje sensortråd producerar objekt av klassen `Sample` och anropar `put()` med dessa sampel. Monitorn ska bara lagra det allra senaste sampelobjektet för varje sensor. Du behöver alltså hålla ordning på tre olika mätvärden i monitorn. Om `put()` skulle anropas flera gånger i rad för en och samma sensor utan att det gamla värdet använts skrivs föregående värde helt enkelt över. Denna metod får aldrig blockera utan ska returnera utan onödigt dröjsmål.

`public Sample getMinimum(long time, long timeout)` – Denna metod blockerar tills det finns mätvärden tillgängliga från alla tre sensorerna sådana att deras tidstämplar är nyare än tiden angiven i parametern `time`. Tiden mäts med hjälp av `System.currentTimeMillis()`. När tre sådana mätvärden finns tillgängliga returneras det sampelobjekt som innehåller det *minsta* värdet på det uppmätta avståndet. Metoden blockerar dock som längst `timeout` millisekunder. Om denna tid överskrids ska värdet `null` returneras. Detta kan indikera att någon av sensorerna inte fungerar som de ska.

Du får ändra på de givna deklarationerna av metoderna i monitorn `DistanceSensing`, men inte så att anropssignaturen ändras. Dvs att metoderna måste fortfarande heta samma saker, acceptera samma argument samt returnera samma värden som tidigare. Du får dessutom lägga till egna privata attribut och metoder samt inre klasser efter behov. Det går också bra att lägga egna hjälpklasser utanför klassen `DistanceSensing` om så önskas.

Använd Javas inbyggda monitorbegrepp för all synkronisering.

(7p)