Tentamen
EDAF85 – Realtidssystem

2022-10–29, 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

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

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

1. Nedanstående Javaprogram har en tendens att hamna i dödläge.

```
1  public static void main(String[] args){
2
3      Lock lockA = new ReentrantLock();
4      Lock lockB = new ReentrantLock();
5      Lock lockC = new ReentrantLock();
6      Lock lockD = new ReentrantLock();
7
8      for (int i=0;i<2;i++) {
9          new Thread(() -> {
10             while (true) {
11                 lockB.lock();
12                 lockD.lock();
13                 useBD();
14                 lockC.lock();
15                 useBCD();
16                 lockD.unlock();
17                 lockB.unlock();
18                 lockA.lock();
19                 useAC();
20                 lockA.unlock();
21                 lockC.unlock();
22             }
23         }).start();
24     }
25
26     while (true) {
27         lockC.lock();
28         useC();
29         lockC.unlock();
30         lockA.lock();
31         lockD.lock();
32         useAD();
33         lockD.unlock();
34         lockA.unlock();
35     }
36 }
```

Anmärkningar

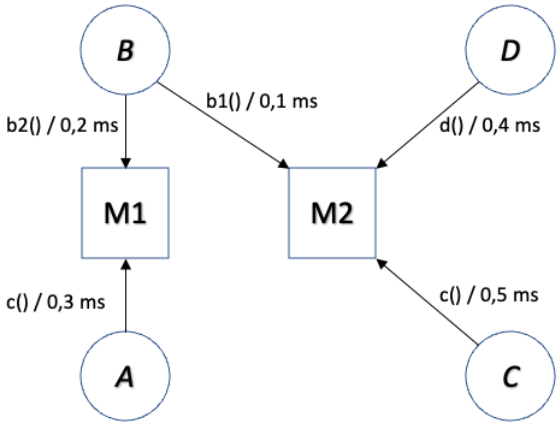
- Definitionerna av operationerna av typen `useXYZ()` har utelämnats, men är ej relevanta för uppgiften.
- `Runnable` är ett så kallat *funktionellt interface*. Det innebär att man som i exemplet ovan (och som vi sett i kursen) kan använda ett lambdauttryck som parameter till konstruktorn för klassen `Thread` för att deklarera och skapa en ny tråd.
- `Reentrantlock` representerar, som vi gått igenom i kursen, en mutexsemafor där operationen `lock()` låser resursen och `unlock()` frigör den.

a) Rita en resursallokeringsgraf för programmet ovan (ge lämpliga egna beteckningar för trådarna och resurserna). *(2p)*

b) Ange på vilka kodrader trådar måste befinna sig då dödläge inträffar. *(1,5p)*

c) Dödläge kan undvikas om vi byter plats på två kodrader i programmet. Vid varje anrop av operationer av typen `useXYZ()` ska samma lås vara låsta som tidigare. Vilka är dessa två rader? *(0,5p)*

2. Antag att vi har ett system som schemaläggs enligt Deadline Monotonic Scheduling, använder sig av dynamiskt prioritetsarv, och består av fyra trådar kallade A, B, C och D. Trådarna använder sig av två monitorer, kallade M1 och M2, för sin inbördes synkronisering enligt nedanstående figur där också respektive monitoroperations maximala exekveringstid är angiven.



Trådarnas värstafallsexekveringstider (C), periodtider (T) och deadlines (D) ges av följande tabell:

Tråd	*C (ms)*	*T (ms)*	*D (ms)*
A	3	30	10
B	1	5	2
C	4	20	20
D	2	10	5

Räkna ut värstafallsresponstiden *R* för respektive tråd. För full poäng måste du redovisa alla dina beräkningar.

(6p)

3. En tråd kan ur schemaläggningshänseende sägas vara i ett av tre tillstånd: *running* (körande), *körbar* (runnable) och *blockerad* (blocked). För vart och ett av dessa tre tillstånd:

1. Vad kännetecknar en tråd som befinner sig i detta tillstånd?
2. Ge för vart av de andra två tillstånden exempel på en händelse som kan få tråden att byta från detta tillståndet till det andra. Om övergång direkt till det andra tillståndet inte är möjligt så ange i stället det.

(2p)

4. I kursen har vi pratat om begreppen *prioritetsinversion* (priority inversion) och *prioritetsärvingsprotokoll*.

a) Förklara kortfattat – illustrera gärna med en figur – vad prioritetsinversion är. *(1,5p)*

b) Ett vanligt förekommande exempel på ett prioritetsärvningsprotokoll är *dynamiskt prioritetsarv* (basic inheritance protocol). Förklara kortfattat vad detta protokoll innebär och hur det löser problemet med prioritetsinversion. *(1,5p)*

5. **M**eddelandeserver

En meddelandetjänst (eller chattjänst) gör det möjligt för en användare att posta meddelanden och att följa (prenumerera på) meddelanden från en utvald mängd andra användare. Tjänsten utgörs av en klientapplikation som användaren kör för att skriva in sina egna meddelanden och läsa andras samt en central serverapplikation som lagrar och förmedlar nya meddelanden till de klienter som prenumererar på dem. Varje klient betjänar exakt en användare.

Denna uppgift behandlar serverapplikationen i vilken klassen `MessageServer` utgör den centrala delen. Main-programmet i serverapplikationen består av en loop som väntar på att klienter ska koppla upp sig via nätverket och identifiera sig med sitt namn och en lista över vilka andra användare de vill se meddelanden från. Själva nätverkskommunikationen till och från klienten hanteras av ett objekt av klassen `NetworkConnection` (se specifikation på följande sidor). Därefter anropas metoden `handleClient()` i `MessageServer`. Det vidare ansvaret för att sköta kommunikation med klienten ligger sedan hos `MessageServer`. Själva meddelandena lagras i en egen datastruktur som implementeras av klassen `MessageDispatcher` (specifikation enligt följande sidor).

När `handleClient` anropas händer två saker: först startas en tråd som ansvarar för att läsa in nya meddelanden från klienten och lagra dem i vår `MessageDispatcher` och sedan startas ytterligare en tråd vars ansvar är att upptäcka nya meddelanden vi prenumererar på i vår `MessageDispatcher` och skicka ut dem via nätverket till klienten. En första version av `MessageServer` återfinns nedan.

Den första versionen av `MessageServer` som du återfinner på efterföljande sidor har dock två problem:

- Klassen `MessageDispatcher` är inte trådsäker: om `addMessage()` och `checkForNewMessages()` anropas samtidigt kan den interna datastrukturen förstöras.
- En av trådarna använder sig av busy-wait vilket leder till att alla trådar i systemet inte får köra som de ska och applikationen konsumerar all tillgänglig CPU-tid.

Din uppgift

Uppdatera koden för klassen `MessageServer` så att den hanterar flera samtidiga klienter på ett korrekt sätt.

Instruktioner

- Du får införa nya klasser, metoder och attribut efter behov.
- Ett objekt av klassen `NetworkConnection` kan utan problem användas av två trådar samtidigt så länge som den ena bara anropar `receiveMessage()` och den andra bara anropar `sendMessage()`.
- `Message` är en enkel klass som bara består av två strängar (användarnamn och meddelande). Du ska aldrig behöva anropa några metoder eller använda några attribut i denna klass i din kod.
- Använd en *monitor* för all synkronisering och signalering som behövs.
- Använd *inte* busy-wait!
- En hel del av koden kommer säkert att vara oförändrad. För att minska mängden kod du måste skriva kan du därför ersätta en sekvens med oförändrade rader med en rektangel innehållande de radnummer du vill lämna oförändrade. Om du t.ex. bara skulle vilja ändra namnet på meddelandevariabeln i den första tråden från `m` till `mess` men lämna allt annat i lösningen oförändrat kan du svara:

  1-18

  ```
  Message mess = conn.receiveMessage();
  dispatcher.addMessage(mess);
  ```

  21-37

(8p)

NetworkConnection

```
/** A connection from a client application to this server. */
public class NetworkConnection {

    /**
     * Returns a message from the client. Each call to this
     * method will return another, new message. If no message
     * is available, this method blocks until one is.
     */
    public Message receiveMessage() throws InterruptedException { ... }

    /** Sends a message to the client. */
    public void sendMessage(Message m) { ... }
}
```

MessageDispatcher

```
/**
 * A data structure for keeping all messages from all clients.
 * NOTE: this data structure is not thread-safe.
 */
public class MessageDispatcher {

    /** Stores a message. */
    public void addMessage(Message m) { ... }

    /**
     * Checks whether a new message is available, according to these criteria:
     *  - the message was sent by any of the users in the set 'senders', and
     *  - the message is more recent than 'previous'.
     *
     * If several messages match these criteria, the oldest one is returned.
     * If no message meets these criteria, null is returned.
     *
     * If 'previous' is null, the oldest message matching 'senders'
     * is returned (or null, if none is available).
     *
     * If this method returns a (non-null) Message reference, that reference
     * should be passed as 'previous' in the next call.
     */
    public Message checkForNewMessage(Set<String> senders, Message previous)
        { ... }
}
```

MessageServer

```
1  import java.util.Set;
2
3  public class MessageServer {
4
5      private final MessageDispatcher dispatcher = new MessageDispatcher();
6
7      /**
8       * Called automatically whenever a client connects.
9       *
10      * @param conn            The connection to the client
11      * @param user            The user's username
12      * @param followedUsers   The users followed by this user
13      */
14     public void handleClient(NetworkConnection conn, String user,
15                              Set<String> followedUsers) {
16         new Thread(() -> {
17             try {
18                 while (true) {
19                     Message m = conn.receiveMessage();
20                     dispatcher.addMessage(m);
21                 }
22             } catch (InterruptedException e) {
23                 throw new Error(e);
24             }
25         }).start();
26         new Thread(() -> {
27             Message previous = null;
28             while (true) {
29                 Message m = dispatcher.checkForNewMessage(followedUsers, previous);
30                 if (m != null) {
31                     conn.sendMessage(m);
32                     previous = m;
33                 }
34             }
35         }).start();
36     }
37 }
```

6. Trådpool med semaforer

Observera: Den här uppgiften ska du lösa med semaforer och/eller lås.

Ibland vill man begränsa antalet trådar som är aktiva samtidigt trots tt det fins många fler uppgifter som skulle kunna utföras parallellt. Detta är till exempel användbart om man vill göra någon stor beräkning som passar för parallellisering, men man har en begränsad mängd CPU-kärnor till sitt förfogande och att då skapa fler trådar än CPU-kärnor skulle inte snabba upp beräkningen – snarare tvärtom. Då kan man använda sig av en så kallad *trådpool*. Idén är att man lägger in alla uppdrag, eller *jobb*, som ska utföras i en kö och sedan har man en liten mängd trådar som var och en tar ut nästa jobb ur kön och utför det. När en tråd har klarat av ett jobb hämtar det nästa jobb ur kön, osv. Om inget jobb väntar på att utföras så inväntar trådarna att det läggs in nya jobb i kön.

Ett enkelt sätt att beskriva ett enskilt jobb i Java är att använda sig av objekt som implementerar interfacet `Runnable`. Jobben läggs in i en kö, t.ex. en `ArrayList<Runnable>` och tas sedan ut av trådarna som sedan anropar metoden `run()` på objekten.

Ett första utkast till en sådan här trådpool (eng. *thread pool*), inklusive en main-metod som bara är till för att illustrera hur trådpoolen är tänkt att användas återfinns på nästa sida.

Tyvärr lider lösningen av samma typ av problem som meddelandetjänsten i föregående uppgift, nämligen:

- `ArrayList` är inte trådsäker.
- Trådarna i poolen använder sig av busy-wait.

Din uppgift
Uppdatera koden för klassen `ThreadPool` så att den blir jämlöpandekorrekt.

Instruktioner

- Du får införa nya klasser och metoder efter behov.
- Använd *semaforer* och/eller *lås* för all synkronisering och signalering som behövs. En lösning baserad på monitorer eller meddelandesändning (mailboxar) godkänns ej.
- Använd *inte* busy-wait!
- Ändra inte på main-metoden.
- En hel del av koden kommer säkert att vara oförändrad. För att minska mängden kod du måste skriva kan du även i denna uppgift därför ersätta en sekvens med oförändrade rader med en rektangel innehållande de radnummer du vill lämna oförändrade. Om du t.ex. bara skulle vilja ändra namnet på parametern till konstruktorn från `n` till `workers` men lämna allt annat i lösningen oförändrat kan du svara:

  1-7
  ```
  public ThreadPool(int workers) {
     waitingJobs = new ArrayList<>();
     noOfWorkers = workers;
  ```
  11-46

(7p)

ThreadPool.java

```
1  import java.util.ArrayList;
2
3  public class ThreadPool {
4
5      private ArrayList<Runnable> waitingJobs;
6      private int noOfWorkers;
7
8      public ThreadPool(int n) {
9          waitingJobs = new ArrayList<>();
10         noOfWorkers = n;
11     }
12
13     public void init() {
14         for(int i=0;i<noOfWorkers;i++) {
15             new Thread(() -> {
16                 while (true) {
17                     if (waitingJobs.size()>0) {
18                         Runnable job = waitingJobs.remove(0);
19                         job.run();
20                     }
21                 }
22             }).start();
23         }
24     }
25
26     public void submitJob(Runnable job) {
27         waitingJobs.add(job);
28     }
29
30     // For testing purpose only
31     public static void main(String [] args) {
32         ThreadPool pool = new ThreadPool(4);
33         pool.init();
34         for(int i=0;i<100;i++) {
35             pool.submitJob(() -> {
36                 System.out.println("Starting...");
37                 try {
38                     Thread.sleep(2000);
39                 } catch(InterruptedException e) {
40                     throw new Error(e);
41                 }
42                 System.out.println("Finished!");
43             });
44         }
45     }
46 }
```