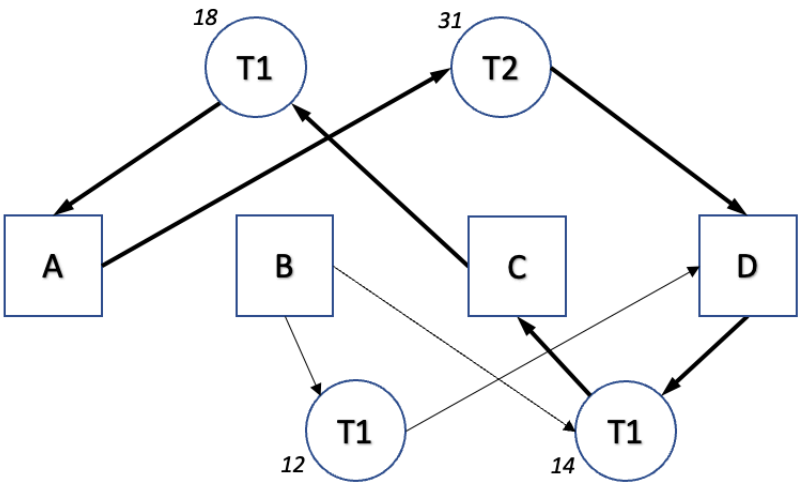
Lösningsförslag till Tentamen, EDAF85 Realtidssystem

2022–10–29, 08.00–13.00

1. a) Kalla resurserna A, B, C och D motsvarande lockA, lockB, lockC och lockD). Dessutom har vi tre trådar – eftersom for-loopen startar två separata, fast identiska, trådar och main-programmet i sig utgör en tråd. Det finns dock bara två *slags* trådar. Vi kallar trådtypen som motsvaras av lambdauttrycket för T1 och trådtypen som motsvaras av main-programmet för T2. Vi får då följande resursallokeringsgraf:



b) Vi ser i resursallokeringsgrafen att det finns ett cirkulärt beroende om vi har två stycken trådar av typ T1 och en tråd av typ T2. For-satsen runt lambdauttrycket gör ju att vi har detta så villkoret för dödläge är uppfyllt. Vi ser också i grafen att i så fall ska huvudprogrammet befinna sig på rad 31 och de två andra trådarna på rad 14 respektive rad 18.

c) Det enklaste sättet att bryta dödlägesrisken är säkert att vända på pilen från A till D via T2. I så fall ska vi ta lås D före vi tar A, dvs byta plats på rad 30 och 31.

2. Först måste vi bestämma prioritetsordningen för trådarna. Eftersom vi ska använda DMS blir prioritetsordningen B, D, A, C.

Därefter bestämmer vi blockeringsfaktorerna:
$B_B = 0,3 + max(0,4, 0,5) = 0,8ms$ (direktblockering i både M1 och M2)
$B_D = 0,5 + 0,3 = 0,8ms$ (direktblockering i både M2 och indirekt blockering via M1)
$B_A = 0,5ms$ (indirekt blockering i M2)
$B_C = 0ms$ (inga lägre prioriterade trådar)

Nu kan vi sätta in våra värden i formeln från formelsamlingen given på tentans första blad. Vi använder iterationsmetoden för att beräkna R för varje tråd.

$R_B = 1 + 0,8 = 1,8$
Eftersom vi inte har något R_B i högerledet finns det ingen anledning att fortsätta iterationen.

$R_D = 2 + 0,8 = 2,8$
$R_D = 2 + 0,8 + \left\lceil \frac{2,8}{5} \right\rceil \cdot 1 = 3,8$
$R_D = 2 + 0,8 + \left\lceil \frac{3,8}{5} \right\rceil \cdot 1 = 3,8$

$R_A = 3 + 0,5 = 3,5$
$R_A = 3 + 0,5 + \left\lceil \frac{3,5}{5} \right\rceil \cdot 1 + \left\lceil \frac{3,5}{10} \right\rceil \cdot 2 = 6,5$
$R_A = 3 + 0,5 + \left\lceil \frac{6,5}{5} \right\rceil \cdot 1 + \left\lceil \frac{6,5}{10} \right\rceil \cdot 2 = 7,5$
$R_A = 3 + 0,5 + \left\lceil \frac{7,5}{5} \right\rceil \cdot 1 + \left\lceil \frac{7,5}{10} \right\rceil \cdot 2 = 7,5$

$R_C = 4 + 0 = 4$
$R_C = 4 + 0 + \left\lceil \frac{4}{5} \right\rceil \cdot 1 + \left\lceil \frac{4}{10} \right\rceil \cdot 2 + \left\lceil \frac{4}{30} \right\rceil \cdot 3 = 10$
$R_C = 4 + 0 + \left\lceil \frac{10}{5} \right\rceil \cdot 1 + \left\lceil \frac{10}{10} \right\rceil \cdot 2 + \left\lceil \frac{10}{30} \right\rceil \cdot 3 = 11$
$R_C = 4 + 0 + \left\lceil \frac{11}{5} \right\rceil \cdot 1 + \left\lceil \frac{11}{10} \right\rceil \cdot 2 + \left\lceil \frac{11}{30} \right\rceil \cdot 3 = 14$
$R_C = 4 + 0 + \left\lceil \frac{14}{5} \right\rceil \cdot 1 + \left\lceil \frac{14}{10} \right\rceil \cdot 2 + \left\lceil \frac{14}{30} \right\rceil \cdot 3 = 14$

Värstafallssvarstidena blir alltså: $R_B = 1,8ms$, $R_D = 3,8ms$, $R_A = 7,5ms$ och $R_C = 14ms$.

3. **Running** Den tråd som just nu kör. Endast en tråd (per CPU-kärna om schemaläggaren använder mer än en CPU-kärna) kan vara i detta tillstånd. Schemaläggaren tar efter en viss tid initiativ till att flytta tråden till tillståndet *runnable*. Om tråden t.ex. anropar `sleep()` flyttas trtråden till tillståndet *blocked*.

Runnable En kö av alla trådar som skulle vilja köra just nu men måste vänta tills CPUn blir ledig. Schemaläggaren flyttar tråden till tillståndet *körande* när det blir denna tråds tur att köra. Tråden kan inte komma till tillståndet *blocked* direkt från detta tillstånd utan måste gå via *running*.

Blocked En tråd i detta tillstånd förbrukar ingen CPU-tid utan väntar på att någon händelse ska inträffa i systemet, till exempel att en viss tid ska ha gått eller någon annan tråd anropar `notify()` i en monitor denna tråd har gjort `wait()`. Då flyttas tråden till tillståndet *Runnable*. Den kan inte flyttas direkt till *running*.

4. a) Prioritetsinversion är ett fenomen som kan inträffa när vi schemalägger trådar enligt principen fixed priority scheduling och inte har något prioritetsärvningsprotokoll. Då kan följande scenario uppträda:

a) En lågprioriterad tråd (L) kör och tar ett lås (t.ex. genom att anropa en monitormetod).
b) En tråd med medelhög prioritet (M) avbryter L och börjar köra.
c) En högprioriterad tråd (H) avbryter M och börjar i sin tur köra.
d) H försöker ta låset som L håller och blockeras.
e) Nu får M köra eftersom M är den körbara tråd som har högst prioritet. Tiden som M kan behöva på sig för att exekvera färdigt kan vara väldigt lång om vi har otur.
f) Först när M har kört färdigt får L köra och kan släppa låset som H vill ha.
g) Till slut får H tillgång till låset och kan köra vidare.

Problemet med detta är att värstafallssvarstiden för H kan bli väldigt lång och väldigt svår att beräkna.

b) Dynamiskt prioritetsarv innebär att när A försöker ta låset i scenariot ovan kommer L tillfälligt att ta över den prioritet som H har, vilket medför att det är L som får köra härnäst och inte M. När L släpper låset kommer den att återfå sin ursprungliga prioritet och det är nu H som får köra vidare. På så sätt behöver H bara vänta på L en kort stund och blockeras inte av M.

5. Lösningen består i att vi kapslar in `MessageDispatcher`-objektet, `dispatcher` i en monitor så att vi kan göra operationerna för att lägga till och hämta ut meddelanden synkroniserade. Vi passar dessutom på att ändra beteendet hos `checkForNewMessage()` så att den blockerar tills ett nytt meddelande finns tillgängligt (m.h.a. ett anrop av `wait()`). För att väcka de väntande trådarna krävs då ett `notifyAll()` i metoden `addMessage()`.

Jämfört med den ursprungliga koden har vi alltså:

- Ändrat på rad 5.
- Tagit bort rad 30 och rad 33.
- Infört klassen `MessageDispatcherMonitor` (rad 37-53 i listningen nedan).

```
1  import java.util.Set;
2
3  public class MessageServer {
4
5      private MessageDispatcherMonitor dispatcher = new MessageDispatcherMonitor();
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                 conn.sendMessage(m);
31                 previous = m;
32             }
33         }).start();
34     }
35 }
36
37 class MessageDispatcherMonitor {
38     private MessageDispatcher dispatcher = new MessageDispatcher();
39
40     public synchronized void addMessage(Message m) {
41         dispatcher.addMessage(m);
42         notifyAll();
43     }
44
45     public sychronized checkForNewMessage(Set<String> followedUsers, Message previous) {
46         Message m;
47         while ((m = dispatcher.checkForNewMessage(followedUsers,previous))==null )
48             try {
49                 wait();
50             } catch(InterruptedException e) { throw new Error(e); }
51         }
52     }
53 }
```

6. För att garantera att `waitingJobs` inte används av mer än en tråd åt gången behöver vi en mutex-semafor. I lösningen nedan väljer vi att använda ett `ReentrantLock`, men vi skulle lika gärna ha kunnat använda en `Semaphore` med startvärdet noll. För att undvika busy-wait inför vi också en räknande semafor som håller reda på hur många jobb som ligger och väntar i kön.

Skillnaden mot ursprungsprogrammet blir då att följande rader i lösningen nedan är nya: 2-3, 9-10, 21-26, 28, 36, 38-39. I den ursprungliga versionen av lösningen har vi dessutom strukit rad 17 och 20.

```
1  import java.util.ArrayList;
2  import java.util.concurrent.Semaphore;
3  import java.util.concurrent.locks.*;
4
5  public class ThreadPool {
6
7      private ArrayList<Runnable> waitingJobs;
8      private int noOfWorkers;
9      private Lock mutex = new ReentrantLock();
10     private Semaphore available = new Semaphore(0);
11
12     public ThreadPool(int n) {
13         waitingJobs = new ArrayList<>();
14         noOfWorkers = n;
15     }
16
17     public void init() {
18         for(int i=0;i<noOfWorkers;i++) {
19             new Thread(() -> {
20                 while (true) {
21                     try {
22                         available.acquire();
23                     } catch(InterruptedException e) {
24                         throw new Error(e);
25                     }
26                     mutex.lock();
27                     Runnable job = waitingJobs.remove(0);
28                     mutex.unlock();
29                     job.run();
30                 }
31             }).start();
32         }
33     }
34
35     public void submitJob(Runnable job) {
36         mutex.lock();
37         waitingJobs.add(job);
38         mutex.unlock();
39         available.release();
40     }
41
42     // For testing purpose only
43     public static void main(String [] args) {
44         ThreadPool pool = new ThreadPool(4);
45         pool.init();
46         for(int i=0;i<100;i++) {
47             pool.submitJob(() -> {
48                 System.out.println("Starting...");
49                 try {
50                     Thread.sleep(2000);
51                 } catch(InterruptedException e) {
52                     throw new Error(e);
53                 }
54                 System.out.println("Finished!");
55             });
56         }
57     }
58 }
```