Lösningar, EDAF85 Realtidssystem

2023–04–20, 08.00–13.00

1. Prioritetsinversion är ett fenomen som inträffar när en högprioriterad tråd blir blockerad av en lägre prioriterad tråd utan att denna direkt blockerar en resurs som den första tråden försöker låsa och som leder till långa fördröjningar som kan vara svåra eller till och med omöjliga att beräkna i förväg.

   Exempel: Tre trådar, A, B, och C, där A har högst prioritet och C lägst. A och C kommunicerar med varandra genom en monitor M. C kör och går in i M (låser monitorn). B börjar köra och avbryter C i kraft av sin högre prioritet. A börjar köra och avbryter i sin tur B på grund av sin ännu högre prioritet. A försöker gå in i M, men blockerar eftersom den är låst av C. B fortsätter köra eftersom den nu är den tråd som har högst prioritet av de körbara trådarna (B och C). Först när B kört klart får C köra och kan lämna M så att A kan fortsätta sin körning. Resultatet är att A har blockerats inte bara av att C behövde lämna C, men också av att B skulle köra färdigt trots sin lägre prioritet.

   Prioritetsinversion kan undvikas genom att införa prioritetsarv. Det kan t.ex. göras genom att låta C tillfälligt få As prioritet så länge som A väntar på att C ska lämna en resurs som A behöver.

2. a) Vi börjar med att bestämma prioritetsordningen. Enligt RMS ska den bli C – A – B (från högst till lägst prioritet).

      C: Bara direkt blockering kan förekomma (i M1): $B_C = 0,7ms$ (från `b()`).

      A: A kan blockeras *antingen* av att B håller M2 (i 0,1 ms) *eller* av att B håller M1 (i 0,7 ms, pga indirekt blockering). Båda kan inte förekomma samtidigt eftersom det stod att endast en monitoroperation anropas samtidigt. Då får vi: $B_A = max(0,1\ ;\ 0,7) = 0,7ms$

      B: B har lägst prioritet och kan alltså inte bli blockerad av några trådar med lägre prioritet. $B_B = 0ms$

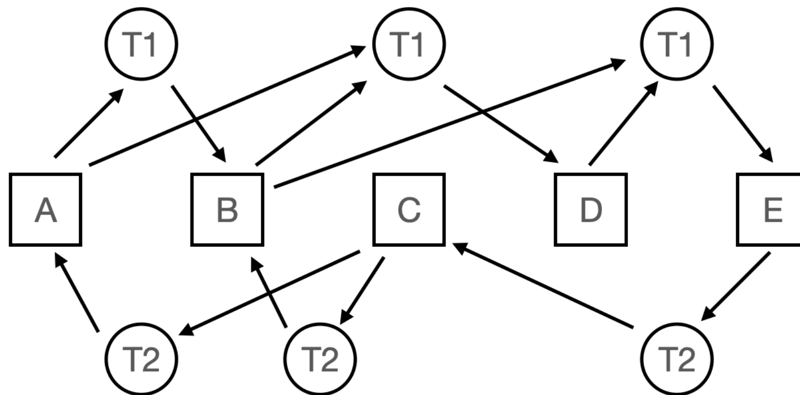
   b) $R_C^0 = 3 + 0,7 = 3,7$
      $R_C^1 = 3 + 0,7 = 3,7$

      $R_A^0 = 4 + 0,7 = 4,7$
      $R_A^1 = 4 + 0,7 + \left\lceil \frac{4,7}{10} \right\rceil \cdot 3 = 7,7$
      $R_A^2 = 4 + 0,7 + \left\lceil \frac{7,7}{10} \right\rceil \cdot 3 = 7,7$

      $R_B^0 = 8 + 0 = 8$
      $R_B^1 = 8 + 0 + \left\lceil \frac{8}{10} \right\rceil \cdot 3 + \left\lceil \frac{8}{25} \right\rceil \cdot 4 = 15$
      $R_B^2 = 8 + 0 + \left\lceil \frac{15}{10} \right\rceil \cdot 3 + \left\lceil \frac{15}{25} \right\rceil \cdot 4 = 18$
      $R_B^3 = 8 + 0 + \left\lceil \frac{18}{10} \right\rceil \cdot 3 + \left\lceil \frac{18}{25} \right\rceil \cdot 4 = 18$

      Svar: $R_A = 7,7ms$, $R_B = 18ms$ och $R_C = 3,7ms$

3. a) Resursallokeringsgraf:



   b) Det finns flera möjliga loopar i grafen, men den loop som minimerar antalet instanser av trådarna är den som går från B via T1 till E, från E via T2 till C och från C tillbaka till B via T2. Det behövs alltså minst en instans av T1 och två instanser av T2 för att dödläge ska kunna uppstås.

4. Vid påtvingad tidsdelning kommer systemet med jämna mellanrum skifta mellan att exekvera de trådar som är körbara så att alla får en del av tiden, eller om prioriteter förekommer byta till en tråd med högre prioritet så att den inte behöver vänta tills den för tillfället exekverande tråden kört färdigt. Detta vill vi normalt ha i ett realtidssystem för att garantera att trådar klarar sina tidskrav.

   I samband med dödläge betyder preemption att vi i förtid tar ifrån en tråd en resurs den har reserverat innan den är klar med resursen. Det kan visserligen göra att dödläge skulle kunna brytas, men det skulle också kunna leda till synkroniseringsfel vilket gör att det är något vi inte vill ha i ett realtidssystem.

5.
   1. Falskt. Det är bara synkroniserade anrop i samma *objekt* som blockerar varandra. Inte anrop i olika objekt.
   2. Sant. Enligt föregående.
   3. Falskt. Eftersom `getAbs` inte är synkroniserad kommer den heller aldrig att blockera när den anropas. Låset på C1 påverkar inte anropet av metoden.
   4. Falskt. Vi kan få felaktiga värden som returneras från metoden ifall `re` och/eller `im` skulle uppdateras under tiden beräkningen av formeln utförs.

6. a) Ett kontextbyte innebär att all information om den körande tråden sparas undan och i stället hämtas den tidigare sparade information om en annan tråd som får fortsätta sin körning. Vi byter alltså vilken tråd som körs.

   b) Statisk schemaläggning innebär att vi i förväg gör upp ett detaljerat schema för vilken aktivitet som ska utföras när och i vilken ordning. Detta schema styr sedan exekveringen av koden och upprepas när man kommit till slutet av schemat.

   c) Rate Monotonic Scheduling innebär att trådar schemaläggs dynamiskt enligt givna prioriteter. Prioriteten sätts efter hur ofta tråden körs. Ju oftare den körs desto högre prioritet.

   d) Vid Earliest Deadline First låter man alltid den tråd köra som har närmast till sin deadline.

7.
   1. Tag bort nyckelordet `synchronized` från metoden `compareTo()`.
   2. Ersätt första kodraden i metoden `compareTo()` med:

```
int r = Double.compare(getBonus(),o.getBonus());
```

   Vi låser då bara ett objekt i taget (medan vi hämtar ut bonusvärdet) i stället för båda objekten. Jämförelsen av namnen är ofarlig eftersom namnattributet är deklarerat final och därför oföränderligt (immutable). I det fallet är samtidig access inget problem.

8.
```
public abstract class LivenessTracker {
        private final PriorityQueue<Timeout> timeouts = new PriorityQueue<>();
        private final Map<Long,Timeout> lastActivityReported = new HashMap<>();

        private class Timeout implements Comparable<Timeout> {
            final Thread thread;
            final long timeout;
            public Timeout(Thread thread, long timeout) {
                this.thread = thread;
                this.timeout = timeout;
            }
            public int compareTo(Timeout other) {
                return Long.compare(this.timeout, other.timeout);
            }
        }

        private class AlertThread extends Thread {
            public void run() {
                while(true) {
                    LivenessTracker.this.timeoutAlert(LivenessTracker.this.awaitTimeout());
                }
            }
        }

        public LivenessTracker() {
            new AlertThread().start();
        }

        public synchronized void reportActivity(long timeout) {
              long now = System.currentTimeMillis();
              Thread t = Thread.currentThread();
              Timeout previous = lastActivityReported.get(t.getId());
              if (previous != null) {
                  timeouts.remove(previous);
              }
              if (timeout!=0) {
                  Timeout e = new Timeout(t, now + timeout);
                  timeouts.add(e);
                  lastActivityReported.put(t.getId(), e);
                  notifyAll();
              }
          }

          private synchronized Thread awaitTimeout() throws InterruptedException {
              long now = System.currentTimeMillis();
              while (true) {
                  Timeout next = timeouts.peek();
                  if (next == null) {
                      wait();
                  } else if (next.timeout > now) {
                      wait(next.timeout - now);
                  }
                  now = System.currentTimeMillis();
                  next = timeouts.peek();
                  if (next.timeout <= now) {
                      timeouts.remove(next);
                      return next.thread;
                  }
              }
          }

          protected abstract void timeoutAlert(Thread.t);
}
```