Lösningar, Tentamen EDAF85 – Realtidssystem

2023-08–24, 14.00–19.00

1. Man kan nämna åtminstone tre olika orsaker:

- Metoden `sleep()` är garanterad att sova minst angivet antal millisekunder, men javastandarden säger inget om hur lång tid den högst får sova.
- Även om `sleep()` sov exakt 1000 ms så kommer det dessutom att krävas tid för att exekvera de andra satserna i loopen. Denna tid måste adderas till varvtiden.
- Om programmet kör på en dator med ett modernt operativsystem med tidsdelning mellan olika program så kan datorn vara upptagen med att köra processer som hör till andra program vilket gör att vårt program måste vänta.

2.
a) Kapplöpning

b) Passagerartråden kör `monitor.awaitLiftStillAndOpenAtFloor(f)` och får veta att hissdörren är öppen (metoden returnerar), men innan, eller under, den anropar `passenger.enterLift();` så sker ett trådbyte och hisstråden börjar exekvera. Hisstråden beslutar sig för att stänga dörren innan passageraren har hunnit gå in i hissen. *(1p)*

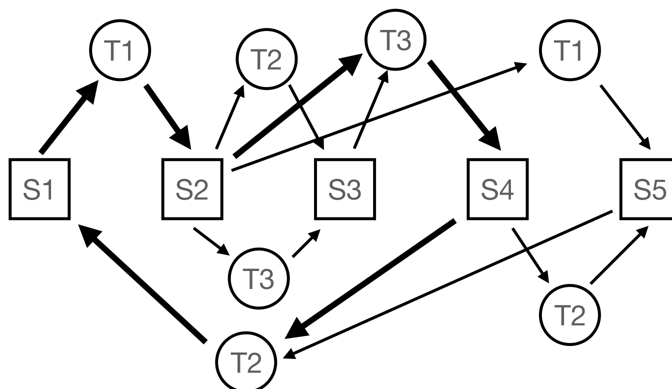
c) Vi måste på något sätt berätta för hissen att det finns passagerare som håller på att gå in i hissen så att den inte stänger dörren för tidigt. Enklaste sättet att göra detta är kanske att införa en räknare inuti monitorn som håller reda på antalet passagerare som håller på att gå in i hissen. Metoden `awaitLiftStillAndOpenAtFloor()` måste modifieras så att den räknar upp denna räknare innan den returnerar. När passageraren har avslutat sitt anrop av `enterLift()` behöver hen informera monitorn om att hen är inne i hissen så att räknaren kan räknas ner igen. Vi behöver alltså en ny metod i monitorn för detta som passagerartråden kan anropa. Hisstråden i sin tur kan i sin tur kontrollera att räknaren är noll innan den stänger dörrarna och går till nästa våning.

3.
a) Kortas periodtid (T) ska ha högst prioritet, så prioritetsordningen blir: T1–T3–T2

b) Vi kan börja med att konstatera att trådarna är oberoende av varandra (ingen blockering), att deadline är lika med periodtiden och att RMS används. Då kan vi tillämpa Liu & Laylands schemaläggningstest. Om vi summerar CPU-utnyttjandegraden för de tre trådarna får vi en total CPU-utnyttjandegrad på: $0,429 + 0,077 + 0,182 = 0,688$. Detta är mindre än gränsvärdet för schemaläggningsbarhet vid oändligt antal trådar (se formelsamlingen på tentans titelsida) som är $\approx 0,693147$. Vi kan därför dra slutsatsen att systemet är schemaläggningsbart. Notera att om värdet hade varit större än gränsvärdet för tre trådar ($\approx 0,780$) så hade vi inte kunnat dra någon slutsats om schemaläggningsbarheten med denna metod. Då hade vi fått göra en axakt analys där vi räknade ut svarstiden för varje tråd och jämförde med dess deadline.

4.

a)



b) Det finns en cykel i resursallokeringsgrafen (markerad med breda pilar) sådan att varje tråd i cykeln bara förekommer en gång (detta är ett krav eftersom det stod att vi bara hade tre, olika, trådar).

c) Byt ordning på `s1.lock();` och `s2.lock();` på kodrad 10–11 i tråd T1.

5. T1: $0,5 + 0,7 = 1,2ms$ (direkt blockering i M1 och M2)
T2: $max(0,2\ ,\ 0,7) = 0,7ms$ (*antingen* direkt blockering i M3 *eller* indirekt blockering via M2)
T3: $0ms$ (inga lägre prioriterade trådar)

6.

```
public class DelayQueue<E> {

    private class Node implements Comparable<Node> {
        private final E element;
        private final long releaseTime;

        private Node(E element, long delay) {
            this.element = element;
            releaseTime = System.currentTimeMillis() + delay;
        }

        @Override
        public int compareTo(Node other) {
            return Long.compare(releaseTime, other.releaseTime);
        }
    }

    private final Queue<Node> q = new PriorityQueue<>();

    // you may add attributes if you want to (but you don't need to)

    /**
     * Adds an element with a delay to the queue. The element
     * will not be returned (in take() below) until 'delay'
     * milliseconds after it was added.
     */
    public synchronized void add(E e, long delay) {
        // TODO: implement this method
        q.add(new Node(e,delay));
        notifyAll();
    }

    /**
     * Blocks until the queue contains at least one element whose
     * release time is met. If more than one such element is
     * available, the one with the earliest release time is returned.
     */
    public synchronized E take() throws InterruptedException {
        // TODO: implement this method
        Node n = q.peek();
        long now = System.currentTimeMillis();

        while (n == null || n.releaseTime >= now) {
            if (n == null) {
                wait();
            } else {
                wait(n.releaseTime - now);
            }
            now = System.currentTimeMillis();
            n = q.peek();
        }

        return q.poll().element;
    }
}
```

7.

```
class Dispatcher {
    // TODO: put your own private declarations here
    POOL_SIZE = 5;
    DelayQueue<Runnable> q;

    /**
     * Constructor
     */
    public Dispatcher() {
        // TODO: implement this method
        q = new DelayQueue<>();
        for(i=0;i<POOL_SIZE;i++) {
            new Thread(()->{while(true) { q.take().run() }}).start();
        }
    }

    /**
     * Adds the task r to the dispatcher. The dispatcher will
     * automatically call the tasks run() method as soon as
     * possible after the delay (in milliseconds) given by delay.
     */
    public addTask(Runnable r,long delay) {
        // TODO: implement this method
        q.add(r,delay);
    }
}
```