Tentamen
Nätverksprogrammering
Lösningsförslag

2017–06–02, 8.00–13.00

Del 1

1. Monitorer
   a) *entry set* innehåller alla trådar som väntar på att få komma in i monitorn, men som inte kan gå in i monitorn då den ägs av en annan tråd. *waiting set* innehåller alla trådar som är blockerade i monitorn, d.v.s. alla trådar som har anropat `wait()`.
   b) Trådar läggs i *waiting set* när de anropat `wait()` i monitorn. Trådar läggs i *entry set* om de anropar en metod i monitorn samtidigt som monitorn ägs av en annan tråd. En tråd tas bort från *entry set* när den släpps in i monitorn (när den blir *ägaren till monitorn*). En/alla trådar från *waiting set* flyttas till *entry set* när `notify()/notifyAll()` anropas av tråden som äger monitorn.

2. HTTP
   a) GET används för att hämta en resurs. PUT används för att skapa en ny instans av en resurs. POST används för att uppdatera en resurs.
   b) Första raden i svaret på en HTTP-operation består alltid av tre delar: 1) protokollversion, 2) statuskod, 3) statustext, t.ex. "HTTP/1.1 200 OK". Statuskoden anger om operationen lyckades, eller indikerar en felkod om den misslyckades. Statuskod i intervallet 200-299 indikerar en framgångsrik operation. Tredje delen av första svarsraden anger en kortfattad förklaring i klartext, "OK" när operationen lyckades.

3. JavaScript
   a) I programmeringsspråk med *function scope* har all kod i en funktion en gemensam namnrymd för variabler. Beteendet när variabler deklareras i ett inre block i en funktion är annorlunda än i t.ex. Java; a) Inre variablerna ersätter tidigare variabler med samma namn (i Java skuggas de yttre variablerna). b) Variablerna är synliga efter det inre blocket. (variabler med *function scope* syns efter blocket, men inte variabler med *block scope*).
   b) *Closure* innebär att funktioner har tillgång till omgivningen de skapades i, d.v.s. variabler deklarerade utanför funktionen. Exempel:

```
var n = 0;

var foo = function(step){
  n = n = step;
}

// n === 0
foo(3);
// n === 3
```

4. a) Falskt (både IP-nummer och port behövs för att skapa en uppkoppling))
   b) Falskt (portnummer 1-1023 är default-portar för kända tjänster, t.ex. 80 för HTTP)
   c) Falskt (en socket har både en input-stream och en output-stream)
   d) Falskt (max en uppkoppling per port, oberoende av protokoll)
   e) Falskt (datan paketeras i TCP-paket)
   f) Sant

5. Hemsidor
   a) HTML innehåller semantisk taggning av sidans innehåll, d.v.s. html-element. CSS innehåller information om sidans layout.
   b) Document Object Model (DOM) är den interna representationen av hemsidan i webbläsaren.
   c) I JavaScript är DOMen synlig via den globala variabeln `Document`. När DOMen modifieras av JavaScript-kod renderar webbläsaren om sidan.

6. Monitorn implementerar en FIFO-kö med bufferstorlek 1. Implementeringen innehåller *kapplöpning*. Då varken läsning eller skrivning till bufferten blockerar måste `get()` anropas mellan två anrop till `put()` för att få det önskade beteendet. Konsumenttråden innehåller även en *busy wait* loop. Detta kan leda till *svält*.

   För korrekt funktion behöver `Monitor.get()` ändras så den blockerar tills det finns ett tal att returnera. På samma sätt behöver `Monitor.put()` blockera när bufferten är full.

Del 2 – Massiva parallella beräkningar

1. a)
```
import java.util.ArrayList;

public class MyJobCache implements JobCache{

    ArrayList<Job> todoList = new ArrayList<Job>();
    ArrayList<Job> resultList = new ArrayList<Job>();

    @Override
    public void addJobs(Job[] jobList) {
        for(Job job : jobList){
            todoList.add(job);
        }
        notifyAll();
    }

    @Override
    public void waitUntilRefillIsNeeded() {
        try {
            while(todoList.size() >= 10){
                wait();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void addResult(Job result) {
        resultList.add(result);
        notifyAll();
    }

    @Override
    public Job getJob() {
        try {
            while(todoList.isEmpty()){
                wait();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        notifyAll();
        return todoList.remove(0);
    }

    @Override
    public Job[] getResults() {
        try {
            while(resultList.size() < 10){
                wait();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        Job[] result = (Job[]) resultList.toArray();
        resultList.clear();
        return result;
    }
```

```
}
```

b)
```
import java.io.BufferedInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.Socket;

public class JobFetcher extends Thread{
    private JobCache jobCache;

    public JobFetcher(JobCache jobCache){
        this.jobCache = jobCache;
    }

    public void run(){
        int nbrJobs = 0;
        while(nbrJobs != -1){
            Job[] jobList = new Job[1];
            jobCache.waitUntilRefillIsNeeded();
            // open socket
            try (Socket s = new Socket("workserver.info", 9090)) {
                InputStream is = new BufferedInputStream(s.getInputStream());
                // get number of jobs offered
                nbrJobs = is.read();
                while( nbrJobs>0 ){
                    nbrJobs--;
                    // fetch one job
                    int bytesRead = 0;
                    int bytesToRead = is.read();
                    if(bytesToRead == -1) { break; }
                    byte[] data = new byte[bytesToRead];
                    while( bytesRead<bytesToRead ){
                        int result = is.read(data, bytesRead,  bytesToRead-bytesRead);
                        if (result==-1) break;
                        bytesRead += result;
                    }
                    jobList[0] = Job.createJob(data);
                    jobCache.addJobs(jobList);
                    jobList[0] = null;
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
            // socket is closed
        }
    }

}

import java.io.BufferedOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;

public class ResultWriter extends Thread{
    private JobCache jobCache;

    ResultWriter(JobCache jobCache){
        this.jobCache = jobCache;
    }
```

```
    public void run(){
        while(true) {
            Job[] resultList = jobCache.getResults();
            // open socket
            try (Socket s = new Socket("workserver.info", 9091)) {
                OutputStream os = new BufferedOutputStream(s.getOutputStream());
                // send the number of result
                os.write(resultList.length);
                for(Job result: resultList){
                    byte[] data = result.getResult();
                    // send this result
                    os.write(data.length);
                    os.write(data);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
            // socket is closed
            // allow garbage collection of old results while waiting for new
            resultList = null;
        }
    }
}
```

c)

```
public class Worker extends Thread{

    private JobCache jobCache;

    public Worker(JobCache jobCache){
        this.jobCache = jobCache;
    }

    public void run(){
        while(true){
            Job job = jobCache.getJob();
            job.compute();
            jobCache.addResult(job);
        }
    }

}
```

d)

```
public class JobClient {

    public static void main(String args[]){
        JobCache jobCache = new MyJobCache();
        JobFetcher fetcher = new JobFetcher(jobCache);
        ResultWriter writer = new ResultWriter(jobCache);
        fetcher.start();
        writer.start();
        for(int i=0; i<4; i++){
            Worker worker = new Worker(jobCache);
            worker.start();
        }
    }

}
```