Chalmers | GÖTEBORGS UNIVERSITET
Alejandro Russo, Computer Science and Engineering

Concurrent Programming TDA382

Friday, March 14, 14:00 PM.

(including example solutions to programming problems)

Alejandro Russo, tel. 0705 110896

- The maximum amount of points you can score on the exam: 68 points. To pass the course, you need to pass each lab, and get at least 24 points on the exam.

  The grade for the exam is as follows:

  Chalmers: grade 3: 24 - 38 points, grade 4: 39 - 53 points, grade 5: 54 - 68 points.
  GU: Godkänd 24-53 points, Väl godkänd 54-68 points

  The grade for the whole course is based on the points obtained in the exam and the labs. More specifically, the course grade (exam + lab points) is determined as follows.

  Chalmers: grade 3: 40 - 59 points, grade 4: 60 - 79 points, grade 5: 80 - 100 points.
  GU: Godkänd if passed the labs and get Godkänd in the exam. Väl godkänd is you passed the labs and got Väl godkänd in the exam.
- Results: within 21 days.
- **Permitted materials (Hjälpmedel):** Dictionary (Ordlista/ordbok)
- **Notes:**
  - Read through the paper first and plan your time.
  - Answers preferably in English, some assistants might not read Swedish yet.
  - If a question does not give you all the details you need, you may make reasonable assumptions. Your assumptions must be clearly stated. If your solution only works under certain conditions, state them.
  - Start each of the questions on a new page.
  - The exact syntax of each programming language you are going to use is not so important as long as the graders can understand the intended meaning. If you are unsure just put in an explanation of your notation.
  - Points will be deducted for solutions which are unnecessarily complicated.
  - As a recommendation, consider spending around 45 minutes per exercise. However, this is only a recommendation.
  - To see your exam:
    - April 9th 2014, 12:15-13:00, room 5128 (5th floor), EDIT building.
    - April 11th 2014, 12:15-13:00, room 5128 (5th floor), EDIT building.

Question 1. **The Problem** Even though this winter is not very snowy this year, still there is a considerable amount of people who have decided to go skiing. When we go there, it is very common to use a cable car to go up to the mountain. There are quite a lot of different types of cable cars, for the exercise we assume one which has the following features:

- There is only one car in the line.
- Skiers only take it from the bottom to the top. They go down skiing!
- The car is not at the bottom initially. So, it has to get to the bottom station and wait for the skiers.
- The car has space for N skiers, where N is a positive natural number.
- Skiers can get in the cable car at the same time. It is important that a beginner skier (who takes quite a lot of time to get in) does not slow down another more proficient skiers when they want to get in.
- The cable car only departs when is full of skiers.
- We assume $N * P$ skiers per day, where P is a positive natural number.

Your assignment

a) You have to implement the simulation of a cable car used by skiers to go up the top of the mountain. The cable car have to wait at the bottom of the mountain until it is full of skiers. Then it goes up and once it reaches the top the skiers can get off. You can assume that the skiers get off instantly from the cable car once it reaches the top. The cable car goes down when is empty. *(8p)*

b) Extend the previous solution without the assumption that the skiers get instantly off the cable car. Therefore, the cable car has to wait until all the skiers get off before it goes down. It is also possible that the beginner skiers also take more time for getting off and it should not interfere when the more proficient skiers want to get of. *(4p)*

To get full points your solution must fulfill the following criteria:

- You must use Java
- You must use semaphores for synchronization or mutual exclusion. No other synchronization constructs are allowed.
- Every skier in the simulation must execute the same code.

1.
```
import java.util.concurrent.Semaphore;

public class Skiers {

    static final int CAPACITY = 3;
    static final int ROUNDS_PER_DAY = 3;
    static final int SKIERS = CAPACITY*ROUNDS_PER_DAY;

    //--------- SECTION a) -----------------
    Semaphore waitingForPassager;
    Semaphore boarding;
    //--------- SECTION a) -----------------
```

```
//--------- SECTION b) -----------------
Semaphore wait_to_getoff;
Semaphore gettingOff;
//--------- SECTION b) -----------------


public static void main(String[] args) {
    new Skiers();
}

public Skiers() {
    waitingForPassager = new Semaphore(0);
    boarding = new Semaphore(0);
    wait_to_getoff = new Semaphore(0);
    gettingOff = new Semaphore(0);

    CableCar cableCar = new CableCar(CAPACITY);
    cableCar.start();

    for(int i=0; i<SKIERS; i++){
        new Skier(i).start();
    }

}

public class Skier extends Thread {
    private int id;

    public Skier(int id) {
        this.id = id;
    }

    public void run(){
        try{

            //--------- SECTION a) -----------------
            System.out.println("Skier␣" + id + ":␣Waiting␣for␣the␣cable␣car!␣\n");
            //Notifying that I am in the station
            waitingForPassager.release();

            //Waiting for my turn to get in the cable car
            boarding.acquire();

            System.out.println("Skier␣" + id + ":␣In␣the␣cable␣car!␣:)␣\n");
            //--------- SECTION a) -----------------

            //--------- SECTION b) -----------------
            //Waiting in the cable car
            wait_to_getoff.acquire();

            //Simulating the time they spend getting off
            int random = (int)(Math.random() * ( 5000 - 1000 ));
            System.out.println("Skier␣" + id + ":␣It␣is␣going␣to␣take␣me␣"
                               + random +"␣milisec␣to␣get␣off␣\n");
            Thread.sleep(random);

            //Notifying to the cable car that I am outside
            gettingOff.release();
            //--------- SECTION b) -----------------

            System.out.println("Skier␣" + id
                               + ":␣Finally␣at␣the␣top!␣\n");


        } catch(InterruptedException e){
            System.out.println("Exception:␣" + e + "␣\n");
        }
    }


}

public class CableCar extends Thread {

    private int roundsPerDay;

    public CableCar(int roundsToday) {
        roundsPerDay = roundsToday;
    }

    public void run() {
        try {
            for (int j = roundsPerDay; j>0; j--) {


                System.out.println("Cable␣car:␣At␣the␣bottom␣station\n");
                Thread.sleep(1000);


                //--------- SECTION a) -----------------

                //Waiting until there are CAPACITY skiers
                for(int i=0; i<CAPACITY; i++){
                    waitingForPassager.acquire();
                }
                //Allowing skiers to get in
                for(int i=0; i<CAPACITY; i++){
                    boarding.release();
                }
                //--------- SECTION a) -----------------

                Thread.sleep(1000);
                System.out.println("Cable␣car:␣Going␣up␣!\n");
```

```
                    //--------- SECTION b) -----------------
                    for(int i=0; i<CAPACITY; i++){
                       //Notifying that they can get off
                       wait_to_getoff.release();
                    }

                    for(int i=0; i<CAPACITY; i++){
                       //Waiting until everyone gets off
                       gettingOff.acquire();
                    }
                    //--------- SECTION b) -----------------

                    System.out.println("Cable␣car:␣Everyone␣is␣off,␣going␣back\n");

                 }
              } catch(InterruptedException e){
                 System.out.println("Exception:␣" + e + "␣\n");
              }

           }

        }


     }
```

Question 2. Consider a savings account shared by several people. Each person associated with the account may deposit or withdraw money from it. The current balance in the account is the sum of all deposits to date less the sum of all withdrawals to date. Clearly, the balance must never become negative. A person making a deposit never has to delay (except for mutual exclusion), but a withdrawal has to wait until there are sufficient funds.

Your assignment You are giving the task to write up a Java class to implement shared accounts. More specifically, you need to provide the following interface.

```
class SharedSavingsAccount {
   public SharedSavingsAccount(long initialBalance) ;
   public void deposit(int amount) ;
   public withdraw(int amount) ;
}
```

Please, do not care about fairness in your solution. You should use Java 5 monitors as synchronization primitive. Here is a short reference of what you will need from `java.util.concurrent.locks`.

```
class ReentrantLock {
  public ReentrantLock();
  public Condition newCondition();
  public void lock();
  public void unlock();
}
class Condition {
  public void await();
  public void signal();
  public void signalAll();
}
```

(10p)

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class SharedSavingsAccount {

    private final Lock lock = new ReentrantLock();
    private final Condition sufficientBalance = lock.newCondition();

    protected long balance;

    public void SharedSavingsAccount(long initialBalance) {
              balance = initialBalance;
    }

    public void deposit(int amount) {
         lock.lock() ;

         balance += amount;

         sufficientBalance.signalAll() ;

         lock.unlock() ;
    }

   public void withdraw(int amount) throws InterruptedException {
         lock.lock() ;

         while (amount > balance)
               sufficientBalance.await();
         balance -= amount;

         lock.unlock() ;
    }
}
```

Question 3. **The Problem** We introduce here a simplified version of a common concurrent pattern used in practice. Consider a client-server architecture implemented in Erlang, where there is one process (the *server*) that serves requests for other processes in the system (the *clients*). These requests involve some computation that may take some time to complete. In the simplest version of this architecture, the clients make requests to the server by sending a message to the server process, which will sit in its mailbox until the server is ready to process it.

This model is convenient because it is easy to implement. However, it is possible for clients to send more requests than the server can handle at a given time, which will fill its mailbox, possibly exceeding the memory capacity of the server. We will refer to this situation as *message flooding*. As a consequence of message flooding, the server's performance can be severely hampered.

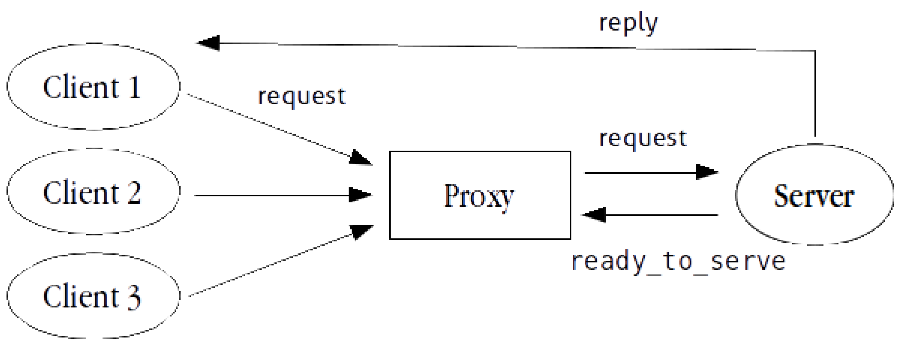

Figure 1: Client-server architecture with a message proxy

Your assignment In this assignment, your task will be to implement a form of mitigation for message flooding, which consists in a *proxy* process for the server, i.e. a process that serves as an intermediary between the clients and the server. The clients should send messages to the proxy process instead of to the server. The proxy process will forward messages from the clients to the server, as long as the total number of active requests does not exceed a fixed amount, given when the proxy starts. Once this bound is overstepped, all subsequent messages are dropped until the server has reduced its load. Dropping these messages mitigates the message-flooding problem, since they will not have to be retained in the server's memory.

To implement this functionality, we must assume that the server sends the atom `ready_to_serve` to the proxy, whenever a request has been fulfilled. The proxy must receive these atoms, using them to maintain a count of the number of active requests in the server. A request counts as active if it has been already sent to the server, but no corresponding `ready_to_serve` atom has been received by the proxy. Figure 1 shows a simple diagram of the proposed architecture. Note that the replies from the server to the clients do not go through the proxy, so you do not have to worry about them.

You are required to write the code of the proxy process in a function called `proxy`, and assume that the server and clients have already been implemented. In particular, you can assume that the server knows the process id of the proxy process (it somehow keeps it in its own state), so the code for serving a request will be of the form

```
server(ProxyPid) ->
   ...
   receive
       ...
       Request -> serve_request(Request),
                  ProxyPid ! ready_to_serve,
                  server(ProxyPid)
   end.
```

Symbol ... denotes some Erlang code which is not relevant for the point we try to make.

Your `proxy` function should take three arguments, i.e., `proxy(Server,C, MaxReq)`, where `Server` is the process id of the server process; `C` is the number of requests being handle by the server; and `MaxReq` is the maximum number of requests that can be served by the server at any given time. *(12p)*

To get full points your solution must fulfill the following criteria:

- You must use Erlang
- You must use Erlang's message passing model. No other synchronization constructs are allowed.

Hint You might consider to use selective **`receive`**, i.e., using **`receive`** with the **`when`** guard.

```
-module(bounded).

-export([bounded_server/2, run/0]).

%%%
%%% Load balancer
%%%

% This process acts as a proxy for the server.
%
% The load balancer keeps track of the current number of requests C,
% and when this number becomes Max, it starts dropping messages until
% the Server informs it that it is ready to take more requests.
proxy(Server, C, Max) ->
    receive
        % decrements the counter of active requests if > 0
        ready_to_serve ->
            io:format("~p␣ready␣to␣serve␣a␣new␣request\n", [Server]),
            proxy(Server, if C > 0 -> C-1; true -> 0 end, Max);

        % prints out debugging information (counter and max)
        debug -> io:format("dbg:␣~p␣~p~n", [C, Max]),
                 proxy(Server, C, Max);

        % A message must be forwarded to the server if the number of
        % active requests is between 0 and Max
        Msg when ((C >= 0) and (C < Max)) ->
            Server ! Msg,
            proxy(Server, C+1, Max);

        % We only reach this case if there are too many active
        % requets, so we drop this message
        Msg -> io:format("message␣dropped:␣~p~n", [Msg]),
               proxy(Server, C, Max)

    end.
```

Question 4. In this exercise, we consider how to handle an *unbounded buffer* in Erlang. To use the buffer, processes have access to the following API.

```
put(Buffer, Item)
get(Buffer)
```

The argument `Buffer` denotes the process id (`Pid`) of the process managing the unbounded buffer.

To put an element in the buffer, processes call **`put`**`(Buffer, Item)`, where `Item` is the value to place in the front of the buffer. This function always succeed, i.e., it never blocks the caller process. Function **get**(Buffer) extracts the last element from the buffer. If the buffer is empty, the caller must block until elements in the buffer are available.

Your assignment

a) You have to implement the process handling the unbounded buffer as well as the functions **put** and **get**. *(12p)*

```
% Never blocks
put(Buffer, Item) ->
    Buffer ! {put, Item},
    ok.

% Blocks when empty
get(Buffer) ->
    Buffer ! {get, self()},
    receive
        X -> X
    end.

% Main server loop (part a)
server({Buffer, Requests}) ->
    receive
        {put, Item} ->
            server(send(Buffer ++ [Item], Requests)) ;
        {get, Pid} ->
            server(send(Buffer, Requests ++ [Pid]))
    end.

send([], Requests) -> {[], Requests} ;

send(Buffer, []) -> {Buffer, []} ;

send([Item|Buffer], [Pid|Requests]) ->
    Pid ! Item,
    {Buffer, Requests}.
```

b) Extend the previous solution with the following variation: **get** calls are only serviced when the number of items in the buffer is exactly equal to the number of waiting get calls. *(4p)*

```
server({Buffer, Requests}) ->
    receive
        {put, Item} ->
            server(sendB(Buffer ++ [Item], Requests)) ;
        {get, Pid} ->
            server(sendB(Buffer, Requests ++ [Pid]))
    end.

sendB(Buffer, Requests) ->
    if
        length(Buffer) == length(Requests) ->
            [ Pid ! Item || {Item, Pid} <- lists:zip(Buffer, Requests)],
            {[],[]} ;
        true ->
```

```
            {Buffer,Requests}
    end.
```

To get full points your solution must fulfill the following criteria:

- You must use Erlang
- You must use Erlang's message passing model. No other synchronization constructs are allowed.

Question 5. Answer the following questions. You should always justify your reply when answering a Yes/No question. Make it short and to the point.

a) In a Java 5 monitor, is it guarantee that the interaction between the boundary queue and the queues associated with conditional variables provides *fairness* for threads? *(2p)*

b) Why workers are useful? *(2p)*

c) In which situations unique references, created by calling `make_ref()`, are needed in Erlang? *(2p)*

d) Describe two differences between messages passing and Linda. *(2p)*

Question 6. Deadlock is a classic problem in concurrent systems. Deadlock is often explained using the Dining Philosopher's Problem. In this Java code snippet, each fork/chopstick is represented by a lock (binary semaphore):

```
int left = i ;
int right = (i+1)%MAX;
while (true) {

  try {
      chops[left].acquire() ;
      chops[right].acquire() ;
  } catch (InterruptedException e) {}

  chops[left].release() ;
  chops[right].release() ;
}
```

a) Describe one systematic manner to avoid deadlock (We saw more than one in the course). *(5p)*

b) Describe minor modifications to the above code such that philosophers can be fed not only safely, but also deadlock-free. *(5p)*