Chalmers | GÖTEBORGS UNIVERSITET
Alejandro Russo, Computer Science and Engineering

Concurrent Programming TDA383

Saturday, October 24, 2015, M (salar i Maskinhuset), 14:00-18:00.

(including example solutions to programming problems)

Behrouz Talebi (Alejandro Russo), tel. 0707 997189 (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 the exam. Väl godkänd is you score 80 points when considering the points of the exam + the points of the labs.

- 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: *by appointment (send email to Alejandro Russo)* on Nov 25th, 12:00-13:00, room 5128 (5th floor), EDIT building.

Question 1. Answer the following questions. You should always justify your reply when answering a Yes/No question. Make it short and to the point.

a) During the lectures we studied the *workers model*. Which problem does this model mitigate? *(2p)*

It mitigates or solves

b) We studied how to make a parallel implementation of the Erlang's function `map`, which we called `pmap` (where the initial `p` comes from parallel). Describe one situation when it is **not** possible to parallelize a `map` function call. *(3p)*

c) How do you know if a function is tail-recursive? *(2p)*

d) What are spurious wake ups in Java? How do they affect your code when using monitors? *(2p)*

e) Can the following code introduce *polling*? *(3p)*

```
while (!sem.tryAcquire()) ;
```

Question 2. The first solution presented for the shared-update problem was Peterson's algorithm:

```
private int turn = 0;
private boolean flag[] = {false, false};
process CS ((int i=0;i<2;i++)) {
   other = (i+1)%2;
   while (true) {
      flag[i] = true;
      turn = other
      while (flag[other] && turn==other) ;
      // Critical section
      flag[i] = false;
   }
}
```

a) If we change `turn == other` by `turn == i`, does the algorithm still achieves mutual exclusion? Justify your answer. *(2p)*

b) If we change `turn = other` by `turn = i`, does the algorithm still achieves mutual exclusion? Justify your answer. *(2p)*

c) If we change `turn = other` by `turn = i` and `turn == other` by `turn == i`, does the algorithm still achieves mutual exclusion? Is there any chance for deadlock? Justify your answer. *(4p)*

Question 3. **The Problem** To deal with the increased number of song requests at a night club, the famous DJ Tupac wants to invest in a new system that allows people to request songs via their smartphones, thus DJ Tupac can focus only on mixing music!

DJ Tupac uses a deck system—the device you see DJs touching all the time when playing music—which internally uses Java. More precisely, the deck uses the following class to administrate a play list.

```
class PlayList {
   private List<Song> playlist = new List<Song>();
   public void addSong (Song title) ;
   public Song CurrentSong() ;
   public Song nextSong() ;
}
```

Your assignment

You should use Java 5 monitors as synchronization primitive. Please, do not care about fairness in your solution. No other synchronization constructs are allowed. 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();
}
```

a) You are given the task to write up the Java class `PlayListThreadSafe` that has exactly the same methods as `PlayList` but it is *thread-safe*, i.e., several threads can call its methods concurrently without damaging the integrity of the playlist. *People at the night club can check out which song is being played or which one comes next as long as the play list is not being modified.*

```
class PlayListThreadSafe {
       // some other fields...
       PlayList pl;
       public void addSong (Song title) ;
       public Song CurrentSong() ;
       public Song nextSong() ;
}
```

(8p)

b) DJ Tupac is now using the deck which includes the class `PlayListThreadSafe` which you wrote! While it works, the DJ is a bit unhappy that sometimes people ask for too many songs in a short time. To make the DJ Tupac happy again, he wants to use a switch on the deck to disallow further requests or queries to the playlist. When the switch is on, the deck gets into **DJMode**. *When the DJMode is on, no more requests and queries to the playlist are accepted until the mode is off.* However, users who have requested songs or queried the playlist during DJMode mode do not need to reissue their requests or queries. You should add the following code to the previous `PlayListThreadSafe` class and modified it accordingly:

```
class PlayListThreadSafe {
   ... // The same methods and private variable as in the point a)
   public void DJmodeON();
```

```
   public void DJmodeOFF();
}
```

(4p)

Solution:

```
import java.util.LinkedList;
import java.util.concurrent.locks.*;

class DJMode extends PlayListThreadSafe{
	private final Lock lock = new ReentrantLock();
	private final Condition condVar = lock.newCondition();
	private boolean isBusy = false;

	private PlayList pl = new PlayList();

	public Song CurrentSong() throws InterruptedException {
		lock.lock();
		try{
			while(isBusy) {
				condVar.await();
			}
			return pl.CurrentSong();
		}
		finally { condVar.signal(); lock.unlock();
		}
	}

	public Song nextSong() throws InterruptedException {
		lock.lock();
		try{
			while(isBusy) {
				condVar.await();
			}
			return pl.nextSong();
		}
		finally { condVar.signal(); lock.unlock();
		}
	}

	public void addSong(Song title) throws InterruptedException {
		lock.lock();
		try{
			while(isBusy) {
				condVar.await();
			}
			isBusy = true;
			pl.addSong(title);
			isBusy = false;
			condVar.signal();
		}
		finally { lock.unlock();
		}
	}
```

```
        public void DJmodeON() throws InterruptedException{
               lock.lock();
               try {
               isBusy = true;
               }
               finally { lock.unlock(); }
        }
        public void DJmodeOFF() throws InterruptedException{
               lock.lock();
               try {
                      isBusy = false;
                      condVar.signal();
               }
               finally { lock.unlock(); }
        }
}
```

Question 4. **The Problem** To deal with the heavy traffic over the river Göta Älv, the city of Gothenburg builds a new bridge. The bridge has two car-lanes, one for each direction. Because of construction, the middle section of the bridge is currently a single one-way car lane shared by cars in both directions.

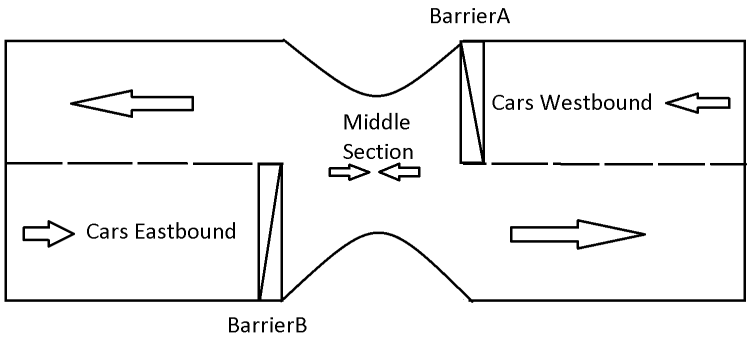

Figure 1: New bridge

We assume the following things:

- Only **one car** is allowed to pass in one direction at a time on the middle section
- Once a car is passing the middle section it always reaches the end of it, i.e., no car is ever stuck in the middle section.
- The barriers can be lifted/lowered by the bridge operator.

Your assignment

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.

a) You have to implement the simulation of the activity that occurs on the bridge. In this part of the task you can ignore the barriers and just write the code for cars and *assume that the barriers are lifted*, i.e., cars can proceed to the middle section as long as there are no other cars there (recall that only 1 car can be in the middle section at a time). The same code needs to be run for cars going Eastbound or Westbound. *(4p)* Solution:

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

public class Bridge {

  static final int N = 10;
  Semaphore middle = new Semaphore(1); //Only one car allowed

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

  public Bridge() {
    for (int i=0; i < 2*N; i++) {
      new Car(i).start();
    }
  }

  public class Car extends Thread {
    private int plate;

    public Car(int plate) {
      this.plate = plate;
    }

    public void run() {
      try{
        System.out.println("Car␣" + plate + "␣waiting␣for␣crossing␣the␣bridge");
        middle.acquire();
        System.out.println("Car␣" + plate + "␣crossing␣the␣bridge");
        this.sleep(500); // It takes some time to cross the bridge
        System.out.println("Car␣" + plate + "␣crossed␣the␣bridge");
        middle.release();
      } catch (InterruptedException e) {
        System.out.println("Exception:␣" + e);
      }
    }
  }
}
```

b) At the beginning of rush hour, the bridge operator decides to lower both barriers. Then, the operator allows N Westbound cars to proceed by raising up the Westbound BarrierA. After that, the operator does the same thing but in the other direction, i.e., he/she lowers the Westbound BarrierA, lifts the Eastbound BarrierB, allows N cars to proceed and then lowers BarrierB. This behavior repeats until the rush hour is over. For simplicity, we will assume that the amount of cars during that time is a multiple of $2N$. In this exercise, you should implement the code for the operator and cars going East- and Westbound. Recall that we are

still assuming that only one car can be at the middle point at a given time. *(8p)* Solution:

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

public class Bridge {

  static final int N = 10;
  Semaphore middle = new Semaphore(1); //Only one car allowed

  Semaphore east = new Semaphore(0);
  Semaphore west = new Semaphore(0);
  Semaphore eastDone = new Semaphore(0);
  Semaphore westDone = new Semaphore(0);

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

  public Bridge() {
    for (int i=0; i < 2*N; i++) {
      if(i < N) {
        new Car(i, east, eastDone).start();
      } else {
        new Car(i, west, westDone).start();
      }
    }
    new Operator().start();
  }

  public class Car extends Thread {
    private int plate;
    private Semaphore barrierWait;
    private Semaphore barrierDone;

    public Car(int plate, Semaphore wait, Semaphore done) {
      this.plate = plate;
      this.barrierWait = wait;
      this.barrierDone = done;

    }

    public void run() {
      try{
        System.out.println("Car␣" + plate + "␣waiting␣for␣barrier");
        barrierWait.acquire();
        System.out.println("Car␣" + plate + "␣waiting␣for␣crossing␣the␣bridge");
        middle.acquire();
        System.out.println("Car␣" + plate + "␣crossing␣the␣bridge");
        this.sleep(100); // It takes some time to cross the bridge
        middle.release();
        System.out.println("Car␣" + plate + "␣notifying␣barrier");
        barrierDone.release();
      } catch (InterruptedException e) {
```

```
        System.out.println("Exception␣in␣Car" + plate +"␣:␣" + e);
      }
    }
  }

  public class Operator extends Thread {

    public void run() {
      try{
        System.out.println("Openning␣east␣barrier");
        for (int i=0; i<N ; i++) {
          east.release();
        }

        for (int i=0; i<N ; i++) {
          eastDone.acquire();
        }
        System.out.println("All␣east␣cars␣passsed,␣closed␣east␣barrier");

        System.out.println("Openning␣west␣barrier");
        for (int i=0; i<N ; i++) {
          west.release();
        }

        for (int i=0; i<N ; i++) {
          westDone.acquire();
        }
        System.out.println("All␣west␣cars␣passsed,␣closed␣west␣barrier");

      } catch (InterruptedException e){
        System.out.println("Exception␣in␣Operator:␣" + e);
      }
    }
  }
}
```

Question 5. During the course, we have seen different kind of primitives for concurrent programming: semaphores, monitors, message passing, etc. In fact, we mentioned that these primitives are equally expressive, i.e., what you can do with semaphores, you can do with monitors, what you can do with monitors, you can do with semaphores, and so on.

Your assignment
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.

In this exercise, we are going to show that message passing is able to encode semaphores. For that, you need to implement the following Erlang module.

```
module(sem)
-export([createSem/1, acquire/1, release/1]).
```

```
createSem(InitialValue) ->
   ...

acquire(Semaphore) ->
   ...

release(Semaphore) ->
   ...
```

Erlang's processes can, for instance, use this module in the following manner:

```
Mutex = createSem(0),
acquire(Mutex),
%% critical section
release(Mutex),
%% rest of the program
```

Acquiring or releasing a semaphore should not delay acquiring or releasing another one—every semaphore minds its own business.

(12p)

Question 6. The Problem In mathematics, the multiplication of a n-dimensional vector of the form

$$v = \begin{bmatrix} x_1 & x_2 & x_3 & \dots & x_n \end{bmatrix}$$

and a quadratic matrix A

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix}$$

is another vector if the form

$$v \cdot A = \begin{bmatrix} (x_1 a_{11} + x_2 a_{21} \cdots + x_n a_{n1}) & (x_1 a_{12} + x_2 a_{22} \cdots + x_n a_{n2}) & \dots & (x_1 a_{1n} + x_2 a_{2n} \cdots + x_n a_{nn}) \end{bmatrix}$$

Your assignment
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.

Calculating each cell in the vector $v \cdot A$ sequentially is not necessary since the cells are not dependent on each other. This present a great opportunity to boost performance by introducing concurrency. So one could calculate each cell in $v \cdot A$ independent of each other and if you have a multicore machine, your code could even run in parallel! *Your task is to write Erlang code which given a vector v and a quadratic matrix A, calculates the multiplication $v \cdot A$ exploiting concurrency as much as possible!*

Assume the following:

- A matrix is represented as a list of lists of integers, i.e., `A = [[1, 2], [3, 4]]` where each list in the list is always a *full row*.

- A vector is represented as a list of numbers.
- We assume that the vector v and matrix A can be always multiplied, i.e., the vector has n elements and the matrix is quadratic and has n rows and n columns.
- You have the following functions at your hand:
  - `transpose(A)` which returns the transposed matrix (where rows are converted into columns). For instance, `transpose([[1,2],[3,4]])` returns `[[1,3][2,4]]`.
  - `pmap(F, Xs)` which returns the list where every element in `Xs` has been applied to function `F`. Importantly, `pmap` concurrently applies function `F` for every element in `Xs`, i.e., it creates a process which computes `F(X)` for every element `X` in `Xs`. We saw how to implement this function in the course, but here you can just use it!
  - `sum(Xs)` which returns the sum of all the elements in the list. For instance, `sum([1,2,3])` returns `6`.

a) Assume that you have the function `sum_of_products(Xs, Ys)` which takes lists `Xs` and `Ys` of equal length, and produces the sum of the point-wise multiplication of the list. In other words, it produces the number `X_1*Y_1 + X_2*Y_2 + ... + X_n*Y_n`, where `X_i` is the i-element of `Xs`, and `Y_i` is the i-element of `Ys`.

Implement the function `mult_parallel(Vector,Matrix)` which produces the result of multiplying a `Vector` of n elements with a quadratic matrix of size $n \times n$ using `sum_of_product` and exploiting concurrency as much as possible.

Solution:

```
mult_parallel(Vector, Matrix) ->
   Columns = transpose(Matrix),
   pmap(fun (Column) -> sum_of_product(Vector, Column) end,
        Columns)
```

(6p)

b) In this point, we want to boost performance even more by implementing concurrently the *multiplication of numbers* in `sum_of_products(Xs, Ys)`—this is useful for multiplying rather big numbers. The way that we are going to implement that is by first spawning every multiplication in a Erlang process and summing up the results produced by them. As a help, we give you a code skeleton and you should complete the dots (...). *(6p)*

```
sum_of_product(Xs, Ys) -> sum_of_product_aux(Xs, Ys,length(Xs)).
sum_of_product_aux([],_,N) ->
    Products = [ receive {res, Result} -> Result end
                 || Number <- seq(1,N)],
    sum(Products) ;
sum_of_product_aux([X|XS], [Y|YS],N) ->
          ...
```

Solution:

```
sum_of_product(Xs, Ys) -> sum_of_product_aux(Xs, Ys,length(Xs)).
sum_of_product_aux([],_,N) ->
    sum [ receive {res, Result} -> Result || Number <- seq(1,N) ] ;
sum_of_product_aux([X|XS], [Y|YS],N) ->
          Myself = self(),
          spawn(fun () -> Myself ! {res,X*Y} end)
          sum_of_product_aux(XS,YS,N).
```