Chalmers | GÖTEBORGS UNIVERSITET
Alejandro Russo, Computer Science and Engineering

Concurrent Programming TDA381/DIT390

Friday, March 9, 08.30hs (4 hours), M.

(including example solutions to programming problems)

Alejandro Russo, tel. 0705 110896

- Grading scale (Betygsgränser):

  Chalmers: 3 = 20–29 points, 4 = 30–39 points , 5 = 40–50 points
  Chalmers ETCS: E = 20–23, D = 24–29, C = 30–37, B = 38–43, A = 44–50
  GU: Godkänd 20–39 points, Väl godkänd 40–50 points

  Total points on the exam: 50

- Results: within 21 days.
- **Permitted materials (Hjälpmedel):**
  - Dictionary (Ordlista/ordbok)
- **Notes:**
  - Read through the paper first and plan your time.
  - Answer in either Swedish or English.
  - 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.
  - The exams will be shown on Friday March 30 th and Monday 2nd April, 12:00hs - 13:00 hs, Room 5471, EDIT Building, Rännvägen 6B.

Question 1. The Problem

Figure 1: Traghetto crossing the Grand Canal

Venice is one of the most romantic cities in Europe. Plenty of channels and bridges, it creates a unique atmosphere. One of the most traditional methods to cross the Grand Canal in Venice is the traghetto (see Figure 1). Traghetto is just a small boat that crosses from one side to the other and is the cheapest means of transportation. For this exercise, we take the crossing between Fondaco dei Turchi to San Marcuolaen. For simplicity, we assume that

- There is only one traghetto.
- Passengers always want to cross in one direction: from Fondaco dei Turchi to San Marcuolaen.
- The traghetto is initially at San Marcuolaen. Therefore, it needs to firstly reach Fondaco dei Turchi to get passengers.
- The traghetto has space for three passengers.
- Passengers cannot board the traghetto when it is not in Fondaco dei Turchi. Otherwise, they will fall into the water!
- Passengers can board the traghetto at the same time. It is important that if a passenger is too slow to get into the traghetto, the other passengers do not necessarily need to wait to board the gondola.
- The traghetto only crosses to the other side when it is full of passengers.
- We assume a total number of $3P$ passengers that want to cross during the day, where P is a positive natural number.

Your assignment

a) Your task is to implement a simulation of the traghetto using semaphores. The important aspect that your simulation should capture is the synchronization of the traghetto crossing and passengers. Your simulation should only consider what happens with the traghetto during one day. For this part of the exercise, you can assume that passengers get off "instantly and at the same time" at the San Marcuolaen station. Thus, you should only focus on the piece of code related with boarding the traghetto. *(5p)*

b)
```
import edu.ucdavis.jr.JR;
public class Traghetto {
```

```
private int total_crossing = 9;

private sem wait_to_board = 0 ;
private sem boarding_done = 0 ;

process Traghetto {
  int crossed = 0;
  while(crossed < total_crossing){
       System.out.println("At␣San␣Marcoulaen␣-␣Crossing");
       JR.nap(1000) ;
       System.out.println("Arrived␣to␣Fondaco␣dei␣Turchi") ;
       for (int i=0; i < 3; i++)
           V(wait_to_board) ;
       for (int i=0; i < 3; i++)
           P(boarding_done) ;
       JR.nap(300) ;
       System.out.println("Arriving␣to␣San␣Marcoulaen␣with␣passengers!");
       crossed = crossed + 3 ;
       System.out.println("Done␣with␣this␣round!");
  }
}

process Passenger((int i=0;i<total_crossing;i++)) {
    System.out.println("Passenger␣"+i+"␣waiting␣for␣the␣Traghetto!");
    P(wait_to_board) ;
    System.out.println("Boarding␣"+i);
    V(boarding_done) ;
    //
    }

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


}
```

c) Extend your solution from the previous point to simulate what happens at San Marcuolaen. For that, you should consider that every passenger must get off before the traghetto turns back for more passengers. Similarly than for boarding, passengers can get off the traghetto at the same time. It is important that if a passenger is too slow to get off the traghetto, the other passengers do not necessarily need to wait to abandon the gondola. *(5p)*

1.
```
import edu.ucdavis.jr.JR;
public class TraghettoFull {

 private int total_crossing = 9;

 private sem wait_to_board = 0 ;
 private sem boarding_done = 0 ;
 private sem wait_to_getoff = 0 ;
 private sem got_off   = 0 ;
```

```
process Traghetto {
  int crossed = 0;
  while(crossed < total_crossing){
       System.out.println("At␣San␣Marcoulaen␣-␣Crossing");
       JR.nap(1000) ;
       System.out.println("Arrived␣to␣Fondaco␣dei␣Turchi") ;
       for (int i=0; i < 3; i++)
           V(wait_to_board) ;
       for (int i=0; i < 3; i++)
           P(boarding_done) ;
       JR.nap(300) ;
       System.out.println("Arriving␣to␣San␣Marcoulaen␣with␣passengers!");
       System.out.println("Start␣getting␣off!");
       for (int i=0; i < 3; i++)
           V(wait_to_getoff) ;
       for (int i=0; i < 3; i++)
           P(got_off) ;
       crossed = crossed + 3 ;
       System.out.println("Done␣with␣this␣round!");
  }
}

process Passenger((int i=0;i<total_crossing;i++)) {
    System.out.println("Passenger␣"+i+"␣waiting␣for␣the␣Traghetto!");
    P(wait_to_board) ;
    System.out.println("Boarding␣"+i);
    V(boarding_done) ;

    P(wait_to_getoff) ;
    System.out.println("Descending␣"+i);
    V(got_off) ;
    System.out.println("Goodbye␣"+i);
    //
    }

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


}
```

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

- You must use semaphores for synchronization or mutual exclusion. No other synchronization constructs are allowed.
- You can use either Java or JR. If you choose to use JR, remember that the primitive **sem** is for declaring semaphores (e.g. **sem s**), the primitive **P(s)** for acquiring semaphore *s*, and **V(s)** for signaling semaphore **s**.
- Passengers must all execute the same code.

Question 2. For this exercise, you should implement a server that stores and handles an unbounded buffer in JR. For simplicity, we assume that the buffer consists on integer numbers. Your implementation must serve the operations `put_buffer` and `get_buffer` related to putting and getting numbers from the buffer, respectively. As usual, if any of these operations cannot be completed because some condition was not fulfilled (e.g., the buffer is empty), then the thread should block until that condition is satisfied. Below, you can see an skeleton of a server implementation serving the mentioned operations.

```
op void put_buffer(int);
op int get_buffer();

op void buffer(int);
op int pending();

  while (true)
  {
    inni void put_buffer(int x) st pending.length() == 0 {
            ...
        }

    []  void put_buffer(int x) st pending.length() > 0 {
            ...
        }

    [] int get_buffer() st buffer.length() == 0 {
          ...
       }

    [] int get_buffer() st buffer.length() > 0 {
          ...
       }
  }
}
```

The operation `buffer` represents the buffer of integer numbers. The operation `pending` is a queue of threads waiting for the buffer to get populated in order to fetch a number.

Your assignment Your task is to fill in the dots in the skeleton provided above in order to complete the implementation of the server.

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

- You must use `forward`.
- You must use message passing for synchronization. No other synchronization constructs are allowed.

(10p)

2.
```
public class UnBuff {

public op void put_buffer(int);
public op int get_buffer();
```

```
  private op void buffer(int);
  private op int pending();

  private process server {
    int y = 0 ;
    while (true)
    {
      inni void put_buffer(int x) st pending.length() == 0 {
               send buffer(x);
          }

      []  void put_buffer(int x) st pending.length() > 0 {
               reply ;
               inni int pending() {
                        return x ;
                    }
          }

      [] int get_buffer() st buffer.length() == 0 {
               forward pending() ;
        }

      [] int get_buffer() st buffer.length() > 0 {
               inni void buffer(int x) {
                        y = x ;
                    }
               return y ;
        }
    }
  }

  private process c {
    edu.ucdavis.jr.JR.nap(2000);
    System.out.println("Sending␣ones!");
    send put_buffer(5);
    send put_buffer(42);
  }

  private process d {
    System.out.println("Getting␣one:"+get_buffer());
    System.out.println("Getting␣one:"+get_buffer());
  }


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


}
```

Question 3. Background

The resource/allocation problem is an illustrative example of a common computing problem in concurrency. We have seen this problem during the lectures. We briefly recap the statement of the problem.

The problem

A controller controls access to some resources. For simplicity, we assume that we have only one kind of resources. Clients make requests to take or return any number of those resources. The controller only grants the resources if there are sufficiently many resources available, otherwise the client requesting those resources must block. More specifically, clients have the following interface:

- `request(n)`, which request `n` copies of the resource and returns a list with the `n` resources.
- `release(rs)`, which returns the list of resources `rs` to the controller.

Your assignment

a) Your task is to implement the primitives `request` and `release` using **transactional memory**. *(8p)*

b) Can deadlock occur in your solution? If so, show under which situation(s) it might occur. Otherwise, justify why your solution is deadlock free. *(2p)*

c) Can starvation occur in your solution? If so, show under which situation(s) it might occur. Otherwise, justify why your solution is starvation free. *(2p)*

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

- You must use pseudo-code. In particular, you must use

  ```
  atomic {
        statements
  }
  ```

  to introduce a transaction. Command **`retry`** is used to abort a transaction and rerun it at a later time.
- To model lists in pseudo-code, which you need for this exercise, you must use the following convention:
  - The empty list is written `[]`.
  - `rs + ys` represents the concatenations of lists `rs` and `ys`, e.g. `[1,2,3]+[4,5,6]` returns the list `[1,2,3,4,5,6]`.
  - Function `take(n,rs)` returns a list with the first `n` elements of `rs`, e.g. `take(2,[1,2,3])` returns the list `[1,2]`. If the length of the list `rs` is shorter than `n`, `take` returns the whole list `rs`.
  - Function `drop(n,rs)` returns the list `rs` without the first `n` elements, e.g. `drop(2,[1,2,3])` returns the list `[3]`. If the `n` is bigger than the length of the list, `drop` returns the empty list.
- You must use transactional memory. No other synchronization constructs are allowed.

a)
```
list_integer resources ; // This is a transactional variable

list_integer request (int n) {
  list_integer return_list ; // This is a local variable, no considered by the
                             // transaction.

  atomic {
   if (length(resources) < n)
        retry ;

   return_list = take(n, resources)
   resources = drop(n,resources)
  }

  return return_list ;
}


void release (list_integer rs) {

  atomic {
     resources = rosources + rs ;
  }

  return ;
}
```

b) Transactional memory guarantees that there are no deadlocks.

c) It might occur. It depends on the scheduler used by the implementation of the transactional memory.

Question 4. **The Problem** You have started working at the company Morzilla. One product of this company is a web browser. The web browser has a cache where web pages are stored for further access and to determine if the user has already visit or not a certain webpage. To keep up with the competitive market, the managers have decided to incorporate a concurrency into the web browser. Now, every tab in the browser runs in a separate thread.

The previous (sequential) version of the browser has class which implements the access to the cache. It has the following interface:

```
class Cache where
    public boolean visitURL (URL url) ;
    public void addURL (URL url) ;
```

Function `visitURL` receives as an argument an URL and returns true when the webpage has been visited previously. Function `addURL` is invoked everytime that a web site needs to be added to the cache. Observe that while `visitURL` reads from the cache, `addURL` writes into it.

Your Assignment Your task is to implement a new class `ThreadSafeCache` which utilizes the old class but that can be safely used by different threads at the same time. Your class should have the same interface as the old one. *It is important to guarantee that several threads can obtain information about the presence of an URL in the cache (i.e., by calling `visitURL`) at*

the same time. There must be no more than one instance of `addURL` *running at a given time.* Please, do not care about fairness in your solution.

As company policy at Morzilla, all programmers must 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();
}
```

(8p)

```
import java.util.concurrent.*;
class ThreadSafePlayList {
  private PlayList playlist;
  private final Lock lock = new ReentrantLock();
  private final Condition waitingToRead = lock.newCondition();
  private final Condition waitingToWrite = lock.newCondition();
  private int readers = 0;
  private int writers = 0;
  public ThreadSafeCache() {
    cache = new Cache();
  }

  public int visitURL(URL url) {
    int result ;
    lock.lock();
    while (writers > 0)
      waitingToRead.await();
    readers++;
    lock.unlock();
    result = cache.visitURL(url);
    lock.lock();
    readers--;
    if (readers == 0)
      waitingToWrite.signal();
    lock.unlock();
    return result;
  }

  public void addURL(URL url) {
    lock.lock();
    while (readers > 0 || writers > 0)
      waitingToWrite.await();
    writers++;
    lock.unlock();
```

```
    cache.addURL(url);
    lock.lock();
    writers--;
    waitingToWrite.signal();
    waitingToRead.signalAll();
    lock.unlock();
  }
}
```

Question 5. Background

We introduce here a simplified version of a common concurrent pattern used in practice. The task is to distribute work tasks among a number of workers. A worker is a process that can perform a computation. Workers can be active or passive. An active worker is a process already performing computations related to a task(s), while passive is a process waiting to be assigned to a task(s). We have a server that keeps track of the tasks to be performed and has a fix number of workers willing to take those tasks. More specifically,

- The **initial state** of the server is a queue of tasks and a list of passive workers.
- A worker can take more than a single task.
- An active worker becomes passive after finishing with the assigned task(s). Similarly, a passive worker becomes active when being assigned a task(s).
- The server **finishes** execution when the queue of tasks is empty and there are no active workers.
- The server waits for a worker to return a result when the queue is empty or there are no more passive workers.
- When the task queue is not empty, the server gets a passive worker and assign some chunk of tasks to perform, i.e., the worker is now active.

How to get a chunk of work from the queue and how to combine results from a single worker with the result from the others is something specific for each problem. However, in order to not make you worry about that, we assume that there exists the following functions.

```
combine_result(Result, Results)
obtain_tasks(Tasks)
```

Function `combine_result` takes a newly produce result and a list of previously produced results and returns a list with the combined results. Function `obtain_tasks` takes a list of tasks `Tasks` and returns a tuple. The first component of the tuple is a chunk of tasks, written `Chunk`, and the second component is the list `Tasks` without the tasks described in `Chunk`.

Your assignment Your task is to provide an implementation of the server that distributes tasks among workers in Erlang. For instance, if the server is implemented as the function `work_load`, then the initial call to that function must be `work_load(Tasks, Passive, ...)`, where `Tasks` is the list of tasks to be computed and `Passive` is a list of PID of the passive workers. We use `...` to denote that there are more arguments in the server that you should think about. The result of calling `work_load` must be the result produced when the server finishes execution. Every worker executes the following code, which might give you a hint about how the code for `work_load` looks like.

```
worker() ->
    receive {Pid, Tasks} ->
            Result = compute(Tasks),
            Pid ! {self(), Result},
            worker()
    end.
```

Function `compute` just computes a result with the given tasks. Even though it sounds difficult, do not be afraid, the solution of this exercise is just a few lines in Erlang! *(10p)*

```
-module(server_workers).
-export([work_load/4, worker/0]).

work_load([], _, [], Results) -> Results;

work_load([Task | Tasks], [Worker | Passive], Active, Results) ->
  {Chunk, TTasks} = obtain_tasks([Task | Tasks]),
  Worker ! {self(), Chunk},
  work_load(TTasks, Passive, [Worker | Active], Results).

work_load(Tasks, Passive, Active, Results) ->
  receive {Worker, Result} ->
    work_load (Tasks,
      [Worker | Passive],
      lists:delete(Worker, Active),
      combine_result(Result, Results))
  end;
```