Chalmers | GÖTEBORGS UNIVERSITET
Alejandro Russo, Computer Science and Engineering

Concurrent Programming TDA381/DIT390

Friday, March 18, 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.

Question 1. **The Problem** Sushi places usually present interesting concurrent patterns. For this exercise, we consider a sushi restaurant where a sushi master serves sushi to one or more customers. We assume that the sushi master places the sushi orders in a circular *finite* bar and the customers take their orders from there. Figure 1 illustrates the scenario.

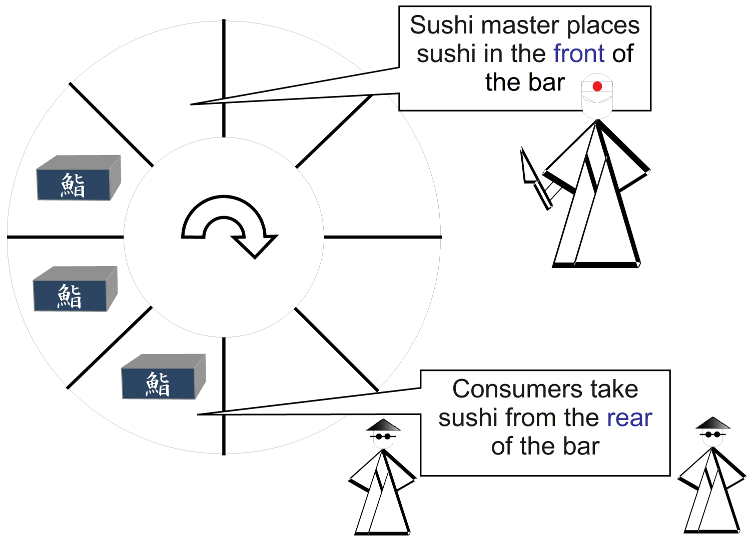
Sushi master places sushi in the front of the bar
鮨
鮨
鮨
Consumers take sushi from the rear of the bar

Figure 1: Sushi restaurant

The sushi master places the sushi in the front of the bar, while customers take sushi from the rear of it. For simplicity, the restaurant has only one dish in its menu, a constant number C of customers per day, and the number of places in the sushi bar is N (for instance, in the picture above $N = 8$).

The rules in the restaurant are the following:

- The restaurant opens at lunch time. Customers can immediately get into the restaurant. However, it is forbidden to go to the sushi bar until the sushi master properly sets up all his (her) elements. Otherwise, it is considered an insult to the sushi master! Once the sushi master is ready to start making sushi, customers are allowed to go to the bar.
- Sushi master must wait if the sushi bar is full
- Customers must wait if the sushi bar is empty

Your assignment Your task is to implement a simulation of the sushi restaurant. The important aspect that your simulation should capture is the synchronization between the sushi master and customers. Your simulation should only consider what happens in the sushi restaurant during one working day. *(10p)*

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

- 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**.
- You must use semaphores for synchronization or mutual exclusion. No other synchronization constructs are allowed.
- Customers must all execute the same code.

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

 private final int N = 5;
 private final int C = 10;

 private final int sushi = 9999 ;

 private sem empty = N ;
 private sem full = 0 ;
 private sem ready = 0 ;
 private sem mutexC = 1 ; // Mutex for customers

 private int [] bar = {0,0,0,0,0} ;

 private int front = 0,
             rear = 0 ;


 process SushiMaster {
   // Preparing tools
   System.out.println("Sushi␣master␣is␣preparing␣his(her)␣tools");
   JR.nap(1000) ;
   for (int i=0; i < C; i++)
       V(ready) ;
   while(true){
      //Produce sushi
      P(empty);
      System.out.println("Making␣sushi!");
      bar[front] = sushi;
      front = (front+1)%N;
      System.out.println("Done␣making␣sushi!");
      V(full);
   }
 }

 process Customer((int i=0;i<C;i++)) {
     int sushi ;
     System.out.println("Customer␣"+i+"␣waiting␣for␣master␣to␣get␣ready");
     P(ready) ;
     while(true){
       P(full);
       P(mutexC);
       System.out.println("Customer␣"+i+"␣grab␣sushi!");
       sushi = bar[rear]; // eat sushi!
```

```
        rear = (rear+1)%N;
        V(mutexC);
        V(empty);
        //Eat sushi
      }
    }


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

Question 2. **The Problem** You have started working at the company Poortify. One product of this company is a music player. The music player allows users to create their own play lists. To keep up with the competitive market, the managers have decided to incorporate a new feature into Poortify. The feature is as follows: users can now specify which other users can add songs to their play lists.

The previous version of the music player had a class which implements the store and administration of a play list. It has the following interface:

```
class PlayList where
     public SongInfo checkSong (SongName song) ;
     public void addSong (SongInfo songInfo) ;
```

Function `checkSong` receives as an argument the name of the song and returns its information (singer, album, record year, etc). If the song is not present in the list, `checkSong` returns **`null`**.

Function `addSong` adds a new song to the play list.

Your Assignment Your task is to implement a new class `ThreadSafePlayList` 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 songs (i.e., by calling `checkSong`) at the same time. Please, do not care about fairness in your solution.

As company policy at Poortify, 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 ThreadSafePlayList() {
  playlist = new PlayList();
 }

 public SongInfo checkSong(SongName song) {
  SongInfo songInfo ;
  lock.lock();
  while (writers > 0)
   waitingToRead.await();
  readers++;
  lock.unlock();
  songInfo = playlist.checkSong(song);
  lock.lock();
  readers--;
  if (readers == 0)
   waitingToWrite.signal();
  lock.unlock();
  return songInfo;
 }

 public void addSong(SongInfo songInfo) {
  lock.lock();
  while (readers > 0 || writers > 0)
   waitingToWrite.await();
  writers++;
  lock.unlock();
  playlist.addSong(songInfo);
  lock.lock();
  writers--;
  waitingToWrite.signal();
  waitingToRead.signalAll();
  lock.unlock();
 }
}
```

Question 3. **Background** The dining philosophers 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 The dining philosophers problem consists of, for instance, four silent philosophers sitting at a circular table (see Figure 2). Philosophers only perform two activities: eat and think. While eating, they are not thinking, and while thinking, they are not eating. A large bowl of rice is placed in front of each of them. A chopstick is placed in between each pair of adjacent philosophers, and each philosopher must only use the chopsticks that are located to his left and to his right. In order to eat, a philosopher must have two chopsticks. So, each philosopher should pick up the chopsticks to his left and right, eat, put the chopsticks down again and then think (and repeat).

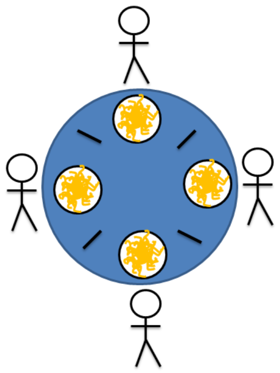

Figure 2: Illustration of the dining philosophers problem

Your assignment

a) Your task is to implement a simulation of the dining philosophers problem using **transactional memory**. You should assume that there are *N* philosophers sit around a table with *N* chairs instead of four as described above. Philosophers should be modelled by processes and nothing else.

(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 can use pseudo-code or Haskell. If you use pseudo-code, you must use

  ```
  atomic {
          statements
  }
  ```

  to introduce a transaction. Command **retry** is used to abort a transaction and rerun it at a later time.

- You must use transactional memory. No other synchronization constructs are allowed.

a)
```
process Philosopher ((int i=0;i<N;i++)) {
   int left = i,
       right = (i+1)%N;
   while (true) {
      //Think
      atomic
      {
```

```
            if (forks[left] == forks[right]==0)
            {
              forks[left] = 1 ;
              forks[right] = 1 ;
            } else retry ;
        }
        //Eat
        atomic()
        {
           forks[left] = 0 ;
           forks[right] = 0 ;
        }
     }
  }
```

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 Consider the following operations:

```
op void syncSend(int x)
op int[] syncReceive()
```

which allow processes to send and receive integers. Operation `syncSend` sends one integer, and operation `syncReceive` receives N integers (from N senders) in one go. For simplicity, we assume that there is only one process receiving data and N processes who send integers. A receiver blocks until all N senders have sent an integer. Senders block until the receiver is ready to get some information and all the other $N-1$ senders have sent their integers. For example, assume that $N = 2$, and given three processes P_1, P_2, and P_3. If P_1 firstly executes `call syncSend(42)`, then it blocks since there is no receiver and the other sender has not yet called `call syncSend`. Then, P_2 executes `syncReceive()`, then it blocks since there is only one sender who wants to send data. Finally, P_3 executes `call synSend(10)`. Then, P_2 receives an array with the elements 42 and 10. Processes P_1, P_2, and P_3 are now capable to continue with their respective computations.

Your assignment Your task is to write a server process that implements the operations `syncSend` and `syncReceive`.

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

- You must use JR (Hint: you might consider the use of `forward`, but it is not strictly necessary here).
- You must use message passing for synchronization. No other synchronization constructs are allowed.

(8p)

A solution in JR.

```
public class Nto1 {

  public op void syncSend(int x);
  public op int[] syncReceive();

  private op void senderQ(int);
  private int N = 2;

  private int [] to_send = {42,10}; // Data to be sent

  private process server {
    int [] data = {0,0};

    while (true)
    {
      inni void syncSend(int x) {
              forward senderQ(x);
           }
      [] int[] syncReceive() st senderQ.length() == N {
          for(int i=0; i<N; i++)
            inni void senderQ(int x) {
                data[i]=x;
                return ;
            }
          return data ;
        }
    }
  }

  // Just testing code below
  private process c {
    int [] data_receive = {0,0} ;
    System.out.println("Going␣to␣receive...");
    data_receive = syncReceive();
    for(int i=0; i<N; i++)
        System.out.println("Received:␣"+data_receive[i]);
  }

  private process sender((int id=0 ; id < N; id++)) {
    edu.ucdavis.jr.JR.nap(2000*id);
    System.out.println("Sending:"+to_send[id]);
    syncSend(to_send[id]);
    System.out.println("Sent:"+to_send[id]);
  }


  public static void main(String[] args) {
    new Nto1();
  }


}
```

Question 5. **Background** There are several approaches to make parallel sequential code. In this exercise, we explore one of them.

Consider the definition of function `map` in Erlang.

```
map(_, []) -> [] ;
map(F, [H|T]) -> [F(H)|map(F,T)].
```

This function sequentially applies function `F` to every element in the list. For example, calling `map(fun (X) ->X*X end, [1,2,3])` gives the result `[1,4,9]`. Here, some examples of other functions that use `map` in their definitions.

```
sum_sqrt(List) -> lists:sum(map(fun (X) -> X*X end, List)).
min_sqrt(List) -> lists:min(map(fun (X) -> X*X end, List)).
suffix_sqrt(List1, List2) ->
        lists:suffix(List1, map(fun (X) -> X*X end, List2)).
```

Function `sum_sqrt` adds the square values of the elements in the list (e.g. `sum_sqrt([1,2,3])` is `14`). Function `min_sqrt` returns the minimum of the square values of the elements in the list (e.g. `min_sqrt([1,2,3])` is `1`). Function `suffix_sqrt` determines if `List1` is a suffix of the list of the square values of `List2` (e.g. `suffix_sqrt([4,9], [1,2,3])` is **`true`**, while `suffix_sqrt([5,9], [1,2,3])` is **`false`**).

Problem Computing the result of function `F` applied to an argument could be an intensive computation, i.e., it could take a considerable amount of time. Under certain conditions, however, it is possible to speed up the code that uses `map` with a parallel version of it, called `pmap`.

Function `pmap` works like `map`, but when we call `pmap(F,L)`, it creates one parallel process for each application of `F` to an element of `L`. For simplicity, we consider only those uses of `map` where the order of elements in the resulting list is not important. For instance, we could change the use of `map` by `pmap` in the functions `sum_sqrt` and `min_sqrt`, but not in `suffix_sqrt`.

Your assignment Your task is to provide an implementation of `pmap` in Erlang. To achieve that, you need to implement a series of auxiliary functions. Even though it sounds difficult, do not be afraid, the solution of this exercise is just a few lines in Erlang!

a) Implement a function called `gather`. This function returns a list of `N` elements and takes a positive number `N` and an unique identifier `Ref` as arguments. The process calling `gather(N,Ref)` waits to get `N` messages of the form {`Ref, Value`} and puts all the received values into a list. For instance, if a process calls `gather(2,123456)` and it has the messages {`123456, 20`}, {`555555, 30`}, and {`123456, 99`} in its mailbox, then the function returns the list `[20,99]` if message {`123456, 20`} is received before than {`123456, 99`}. Otherwise, the function returns the list `[99,20]`. *(5p)*

b) Implement a function called `distribute`. This function takes a list `List`, a function `F`, and a unique reference `Ref` as arguments. When calling `distribute(List,F,Ref)`, the function creates, for each element `X` of the list, a process to compute `F(X)`. Each of these processes will send a message to the process executing `distribute`. The messages will have two components. The first component is the unique identifier `Ref` and the second component being `F(X)`. *(5p)*

c) Implement function `pmap` using `gather` and `distribute`. *(2p)*

```
-module(pmap).
-export([gather/2, distribute/3, pmap/2]).
```

```
gather(0,_) -> [] ;
gather(N,Ref) ->
    receive
      {Ref, Value} -> [Value | gather(N-1, Ref)]
    end.


distribute([], _, _) -> true ;
distribute([Value|Rest], F, Ref) ->
    S = self(),
    spawn( fun() -> S ! {Ref, F(Value)}

            end ),
    distribute(Rest, F, Ref).

pmap(L,F) ->
    Ref = erlang: make_ref(),
    distribute(L,F,Ref),
    gather(length(L),Ref).
```