Concurrence, Parallelism and Distributed Systems (CPDS)
Module I: Concurrency
Facultat d'Informàtica de Barcelona
Final Exam
November 11, 2014

Answer the questions concisely and precisely
Answer each problem in a separate page (remember to put your name)
Closed-book exam
Duration: 2 hour

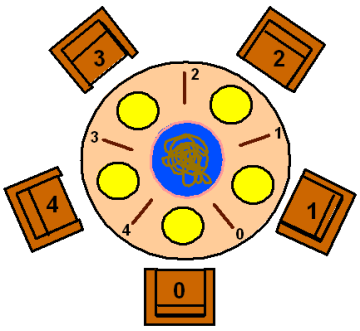
Exercise 1 (2 Points)
Nice Manchester Lunch. Given the processes

```
SMILE = ( smile -> STOP).
LUNCH = ( eat -> drink -> STOP).
||LUNCH_SMILE=(LUNCH||SMILE).
||JOINT_LUNCH = (a:LUNCH||b:LUNCH).
```

do the following

- (1/2 Point) Draw the LTS for `LUNCH_SMILE`.
- (1/2 Point) What are the possible traces of actions that can happen in `LUNCH_SMILE`?
- (1 Point) Draw the LTS for `JOINT_LUNCH`.

Exercise 2 (4 Points)
We borrow from M&K's slides the following explanation and picture.
Five philosophers sit around a circular table. Each philosopher spends his life alternately thinking and eating. In the centre of the table is a large bowl of spaghetti. A philosopher needs two forks to eat a helping of spaghetti. One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left.



Consider the basic FSP description given in M&K of the Dining Philosophers problem.

```
FORK = (get -> put -> FORK).
PHIL = (sitdown -> right.get -> left.get->
        eat-> left.put -> right.put-> arise -> PHIL).
||DINERS(N=5)
     = forall [i:0..N-1]
           (phil[i]:PHIL || {phil[i].left, phil[((i-1)+N)%N].right}::FORK).
```

Note that `DINERS` models, for the default parameter value, five concurrent `PHIL` processes and five concurrent `FORK` processes. Each `FORK` is a shared resource that can be used by two philosophers.

Consider the following points:

- (1 Point) The process `DINERS(N=5)` has a deadlock. Explain in a short sentence the deadlock and give a (shortest) trace to deadlock.
- (2 Points) To avoid the deadlock we introduce a parametrized 'butler' process that allows at most `N-1` philosophers to sit down simultaneously.

  ```
  BUTLER(...) = ...

  ||SITTING(N=5) = (....::BUTLER(...)).

  ||BUTLER_DINERS(N=5) = ...
  ```

- (1 Point) The `Fork` LTS process corresponds to a `Fork` monitor in Java. Complete the following snipped code:

  ```
  class Fork {

    private boolean taken=...;
    private int identity;

    Fork(int id)
      {...;}

    synchronized void put() {
      taken=...;
      ...
    }

    synchronized void get()
       throws java.lang.InterruptedException {
     ...
    }
  }
  ```

Exercise 3 (4 Points)
Matrix product. Let us develop a `matrix_product` module to compute the matrix product.

- (1 Point) Remind that given two vectors `X` and `Y`, for instance

  ```
  2> X= [1.0, 2.0, 3.0, 4.0].
  [1.0,2.0,3.0,4.0]
  4> Y= [1.0, 4.8, 9.8, 16.0].
  [1.0,4.8,9.8,16.0]
  ```

  the dot product is `dot_prod(X,Y) = 1.0*1.0 +...+4.0*16.0`. Define a function

  ```
  % Pre: both input lists have the same length
  dot_prod([], []) -> 0;
  dot_prod([... | ...], ...) -> ....
  ```

  such that

```
6> D= matrix_product:dot_prod(X,Y).
104.0
```

- (1 Point) In Erlang we give matrix row by row

```
7> M = [[1.0, 2.0, 3.0, 4.0], [1.0, 4.0, 9.0, 16.0] ,[1.0, 8.0, 27.0, 64.0] ].
```

Corresponds to the matrix

$$M = \begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 \\ 1.0 & 4.0 & 9.0 & 16.0 \\ 1.0 & 8.0 & 27.0 & 64.0 \end{bmatrix}$$

Given `M` the `transpose(M)` changes rows into columns, for instance:

$$\texttt{transpose}(M) = \begin{bmatrix} 1.0 & 1.0 & 1.0 \\ 2.0 & 4.0 & 8.0 \\ 3.0 & 9.0 & 27.0 \\ 4.0 & 16.0 & 64.0 \end{bmatrix}$$

In Erlang

```
11> matrix_product:transpose(M).
[[1.0,1.0,1.0],[2.0,4.0,8.0],[3.0,9.0,27.0],[4.0,16.0,64.0]]
```

Complete the following `transpose(M)` function

```
transpose([]) -> [];
transpose([[]|_]) -> [];
transpose(M) -> [ [H || [...| ...] <- M ] | transpose([ ... || ... ]) ].
```

- (1 Point) Finally complete the following function to compute the matrix product

```
mult(A, B) ->
   BT = transpose(...),
   [[ dot_prod(RowA, ...) || ...] || RowA <- ...].
```

such that (shematically):

```
11> M=[[1.0,2.0,1.0],[2.0,0.0,3.0]].
12> N=[[1.0,2.0,3.0],[2.0,1.0,2.0],[3.0,2.0,1.0]].
13> c(matrix_product).
14> P=matrix_product:mult(M, N).
[[8.0,6.0,8.0],[11.0,10.0,9.0]]
```

- (1 Point) Remind the `server5` given in the section 16.1 of Armstrong book.

```
-module(server5).
-export([start/0, rpc/2]).

start() -> spawn(fun() -> wait() end).

wait() ->
   receive
     {become, F} -> F()
   end.

rpc(Pid, Q) ->
    Pid ! {self(), Q},
    receive
        {Pid, Reply} -> Reply
    end.
```

started with

```
Pid=server5:start().
```

Suppose that `server5` is currently running as a factorial server (as in the book). Imagine that you need it to become a matrix product server.

- Design a module `my_mult_server` to do the job.
- Complete the following instruction in order to update the server.

```
Pid!{... , ...}
```