Concurrency, Parallelism and Distributed Systems (CPDS)
Module I: Concurrency
Facultat d'Informàtica de Barcelona
Final Exam
April 26, 2018

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 *Client Server extension* (2 Point).
Remind the basic `CLIENT_SERVER` architecture,

```
CLIENT = (call -> wait -> continue -> CLIENT).
SERVER = (request -> service -> reply -> SERVER).
||CLIENT_SERVER = (CLIENT || SERVER)/ {call/request, wait/reply}.
```

We would like to extend the client server system `CLIENT_SERVER` so that two clients can use the server.

1. (1 Point) In order to do so, complete `TWO_CLIENT_SERVER` in the following schema:

   ```
   const N = 2
   CLIENT = (call -> wait -> continue -> CLIENT).
   SERVER = (request -> service -> reply -> SERVER).
   ||TWO_CLIENT_SERVER = (client[...]:CLIENT || SERVER)
                              / {client[....]..../request, .../reply}.
   ```

2. (1 Point) Consider the following traces and say if they are correct or not (argue briefly your answer).

   (a) The first one is:

   ```
   client.1.call, service, client.1.wait, client.2.call,
   client.1.continue, service, client.2.wait
   ```

   (b) The second one is:

   ```
   client.1.call, service, client.2.call,  client.1.wait,
   client.1.continue, service, client.2.wait
   ```

Exercise 2 *Counter* (3 Points).
Consider the `COUNT` process,

```
COUNT (N=5)   = COUNT[0],
COUNT[i:0..N] = (when(i<N) inc->COUNT[i+1]
                |when(i>0) dec->COUNT[i-1]
                ).
```

1. (1 Point) Translate this process into a Java Monitor `Count` following the Magee & Kramer rules.
2. (1 Point) Encode a class `CountNoWait` following the aproach develop in the lab. Remind that in this case there is no `wait()` or `notifyAll()`.
3. (1 Point) Compare both approaches.

Exercise 3 *Warring Neighbors again* (3 Points).
Consider the following FSP model for the warring neighbors problem.

```
CARDVAR = CARDVAR[1],
CARDVAR[j:1..2] = (set1 -> CARDVAR[1]
                  |set2 -> CARDVAR[2]
                  |[j] -> CARDVAR[j]
                  ).

set CardActions = {set1, set2, [1], [2]}

||TURN = (turn: CARDVAR).

NEIG1 = (turn.set2 -> turn[t:1..2]
                   -> if (t == 2)
                      then NEIG1
                      else (enter -> exit -> NEIG1)
        ) + {turn.CardActions}.

NEIG2 = (turn.set1 -> turn[t:1..2]
                  -> if (t == 1)
                     then NEIG2
                     else (enter -> exit -> NEIG2)
        ) + {turn.CardActions}.

property MUTEX = ....

||FIELD = (alice:NEIG1 || bob: NEIG2 || {alice, bob}::TURN || MUTEX).

progress ALICE_ENTER = {alice.enter}
progress BOB_ENTER = {bob.enter}

||GREEDY = FIELD << {alice.turn.set2, bob.turn.set1}.
```

1. (1/2 Point) Complete the description of `MUTEX` in order to check mutual exclusion on enter–exit actions.
2. (1/2 Point) Has `FIELD` the `MUTEX` property? Argue your answer.
3. (1 Point) Has `GREEDY` progress properties `ALICE_ENTER` and `BOB_ENTER`? Argue your answer.
4. (1 Point) What is your opinion about this model? Compare it with the Peterson's solution.

Exercise 4 *my_counter* (2 Points).
Complete the following code,

```
-module(my_counter).
-export([start/0,loop/1,increment/1,value/1,stop/1]).

%% (1 Point) The interface functions.
```

```
start() -> spawn(...).

increment(Counter) -> Counter ! increment.

value(Counter) ->  ... ! {self(),value},
                   receive
                        {..., ...} -> ....
                   end.

 stop(Counter) -> Counter ! stop.

%% (1 Point) The counter loop.

loop(Val) -> receive
                 increment -> loop(...);
                 {From,value} -> From ! {self(),Val}, ...;
                 stop -> true;
                  _ -> % All other messages, recursive call
                          ...
             end.
```

in order to have the following behaviour,

```
20> c(my_counter).
{ok,my_counter}
21> Counter = my_counter:start().
<0.89.0>
22>
22> my_counter:value(Counter).
0
23> my_counter:increment(Counter).
increment
24> my_counter:value(Counter).
1
25> my_counter:increment(Counter).
increment
26> my_counter:value(Counter).
2
27> my_counter:stop(Counter).
stop
28>
```