Concurrency, Parallelism and Distributed Systems (CPDS)
Module I: Concurrency
Facultat d'Informàtica de Barcelona
Final Exam, April 22, 2016

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

Exercise 1 *Sugarolas* (5 Points).
The recent launch of the drink *Sugarola* has been a success. The process `SUPERMARKET` below models a supermarket where the number of Sugarola bottles that a customer can buy is only limited by the number of available bottles on the shelf. For instance, action `get[2]` means buying 2 bottles of Sugarola in a purchase. The process `WORKER` refills the shelf when Sugarola bottles are scarce (smaller or equal than `Min`). At any time, the maximum numbers of bottles in the shelf is `Max`.

```
const Min = some number                    //defines the threshold for filling
const Max = some number greater than Min   //shelf capacity

SHELF = BOT[0],
BOT[i:0..Max] = (when (i > 0) get[....].....
                |when (i<= Min)......
                )

WORKER = (fill->WORKER).
CLIENT = (get[1..Max]->CLIENT).

||SUPERMARKET = (SHELF || WORKER || CLIENT).
```

1. Complete the `SHELF` process. At the initial state the shelf has no bottles.
2. Assume in this question (and only here) that `Max = 3` and `Min = 1`. Draw the `SUPERMARKET` process.
3. Define a safety property `NOFILLCHAINED` to show that there are no traces of `SUPERMARKET` issuing two chained fill actions. Simpler property definitions will obtain better grades.
4. Write a Java monitor for `SHELF` that avoids disturbing unnecessarily the threads on the waiting set. Assume there are several `WORKER` threads and several `CLIENT` threads running concurrently.
5. A new definition of the process `SUPERMARKET` models two types of client, one of them greedy

   ```
   ||NEW_MARKET = (SHELF || WORKER || {a,b}:CLIENT)/{{a,b}.get/get} << {a.get}.
   ```

   Draw `NEW_MARKET` when `Max = 3` and `Min = 1`. Think about the following progress properties concerning `NEW_MARKET`. Are they true? Explain your answer.

   (a) `progress FILL = {fill}`
   (b) `progress B_GET = {b.get[1..Max]}`

Exercise 2 *More on Parallel Sorting.* (5 Points).

Suppose that all the following programs are in the jh_pqs module. Let us remind the sequential recursive version of the quicksort:

```
qsort([]) -> [];
qsort([X|Xs]) ->
   qsort([Y || Y <- Xs, Y<X])
   ++ [X]
   ++ qsort([Y || Y <- Xs, Y>=X]).
```

The following holds:

```
3> L= [1,2,3, 83, 117, 114, 112, 114, 105, 115, 101].
[1,2,3,83,117,114,112,114,105,115,101]
4> jh_pqs:qsort(L).
[1,2,3,83,101,105,112,114,114,115,117]
5> jh_pqs:qsort("hello word").
" dehlloorw"
```

In order to parallelize let us develop the idea of *sort the second half in parallel* and apply recursion.

1. (1 Point). In a first try, we ask to complete the following incorrect code (do not worry, you will correct later),

   ```
   psort([]) -> [];
   psort([X|Xs]) ->
      Parent = ...,
      spawn(fun() -> ... ! psort([Y || Y <- Xs, Y >= X]) end),
      psort([Y || Y <- Xs, Y < X])
     ++ [X]
     ++ receive Ys -> Ys end.
   ```

   As we see, psort is not correct!

   ```
   13> jh_pqs:psort(L).
   [1,2,3,83,101,105,112,114,117,114,115]
   14> jh_pqs:psort("hello word").
   " edhlloowr"
   ```

2. (1 Point). It happens that psort is slower that qsort (in a quad core, intel CORE i5).

   ```
   16> jh_pqs:test_qsort(5000000).
   11.43500000005588
   17>
   17> jh_pqs:test_psort(5000000).
   14.288999999989755
   ```

   To solve that, let us control the granularity by a parameter "D". Please complete the following (yet incorrect) psort2 program:

   ```
   psort2(Xs) -> psort2(5,Xs).

   psort2(0,Xs) -> qsort(Xs);
   psort2(_,[]) -> [];
   psort2(D,[X|Xs]) ->
      Parent = ...,
      spawn(fun() -> ... ! psort2(D-1,[Y || Y <- Xs, Y >= X]) end),
      psort2(...,[Y || Y <- Xs, Y < X]) ++
      [X] ++
      ....
   ```

The program is yet incorrect, but at least, is faster..

```
20> jh_pqs:psort2("hello word").
" edhlloorw"

26> jh_pqs:test_psort2(5000000).
5.599999999976717
```

Explain shortly and informally the relationship between `D` and the number of cores.

3. (1 Point). Explain why `psort2` code is incorrect.

   Hint: unfold the call `psort2(...,[Y || Y <- Xs, Y < X])`. If are not sure about the answer, go the the next item and get back later...

4. (1 Point). Let us remain briefly how to tag messages uniquely:

   - `Ref = make_ref()`, create a globally unique reference,
   - `Parent !{Ref, Msg}`, send the message tagged with the reference and
   - `receive {Ref, Msg} -> ... end`, match the reference on receipt... picks the right message from the mailbox.

   Please complete the following code

```
psort3(Xs) -> psort3(5,Xs).

psort3(0,Xs) -> qsort(Xs);
psort3(_,[]) ->[];
psort3(D,[X|Xs]) ->
   Parent = ...,
   Ref = make_ref(),
   spawn(fun() -> ... !{... ,psort3(... , ...)}end),
   psort3(.... , ...)
   ++ ...
   ++ receive {... ,Greater} -> ... end.
```

   Note that (if you complete in the right way) it works correctly, without time degradation,

```
29> jh_pqs:psort3("hello word").
" dehlloorw"

32> jh_pqs:test_psort3(5000000).
5.258000000030734
33>
```

   Explain why `psort3` is correct.

5. (1 Point). Let us consider a final improvement adding a new variables `Grtr` and `Smlr`. Complete the following final version

```
psort4(D,[X|Xs]) ->
   Parent = ...,
   Ref = make_ref(),
   Grtr = [Y || Y <- Xs, Y >= X],
   Smlr =
   spawn(fun() -> ... ! {...,psort4(..., ...)} end),
   ...
```

   It happens,

```
40> jh_pqs:test_psort4(5000000).
5.725000000093132
```

   Explain shorty by it could make makes sense to use `Grtr` and `Smlr` in the preceding code even if the improvement is not clear.