Concurrent Programming

Exercise Booklet 8: Message Passing

Solutions to selected exercises (◊) are provided at the end of this document. Important: You should first try solving them before looking at the solutions. You will otherwise learn **nothing**.

Exercise 1. Implement the turnstile example given at the beginning of the course. You should have a counter thread and two turnstile threads. All reads and writes to the counter are managed through its dedicated thread.

Exercise 2. Implement a server that concatenates strings. It should follow the protocol given by the following atoms (which are sent to the server):

- `{start}`: client wishes to send a number of strings to be concatenated;
- `{add,S}`: concatenate string `S` to the current result;
- `{done}`: done sending strings, send back result.

Exercise 3. We wish to define a server that receives two types of messages from a client: "continue" and "counter".

1. Each time the server receives the message "counter" it should display on the screen the number of times that it received the "continue" message since the last time it received the "counter" message.
2. Modify your solution so that the server, instead of printing the number on the screen, sends it back to the client which then prints the number of the screen himself.

Exercise 4. (◊) We wish to model a *Timer* that, when spawned, received the frequency with which it should generate *ticks* and the set of pids which should receive ticks (a message represented as the atom `tick`). For example, `spawn(timer,timer,[100, [pid1,pid2,pid3,pid4]])` will initiate the `timer` process that will send tick messages every 100 milliseconds and send them to pids `pid1`, `pid2`, `pid3` and `pid4`.

Exercise 5. Modify your solution so that the *Timer* is initially created with an empty list of pids and that clients have to register in order to receive ticks. For that they need to communicate with the *Timer* sending it a `register` message.

Exercise 6. Write a server that receives a number `n` and returns a boolean indicating whether it is prime or not. Provide two solutions, one where you define an ad-hoc client and server and another using the generic server seen in class.

Exercise 7. (◊) Implement the bar exercise (Exercise 1) from EB5. On Fridays the bar is usually full of men, therefore the owners would like to implement an access control mechanism in which one man can enter for every two women.

1. Complete the template below:

```
-module(bar).
-compile(export_all).

start(W,M) ->
    S=spawn(?MODULE,server,[0,0]),
    [spawn(?MODULE,woman,[S]) || _ <- lists:seq(1,W)],
    [spawn(?MODULE,man,[S]) || _ <- lists:seq(1,M)].

woman(S) ->  % Reference to PID of server
    error(not_implemented).

man(S) ->   % Reference to PID of server
    error(not_implemented).

server(Women,Men) ->   % Counters for Women and Men
    error(not_implemented).
```

2. Consider how to address that extension where it can get late (see details in EB5). Complete the template below:

```
-module(bar2).
-compile(export_all).

start(W,M) ->
    S=spawn(?MODULE,server,[0,0,false]),
    [spawn(?MODULE,woman,[S]) || _ <- lists:seq(1,W)],
    [spawn(?MODULE,man,[S]) || _ <- lists:seq(1,M)],
    spawn(?MODULE,itGotLate,[3000,S]).

itGotLate(Time,S) ->
    timer:sleep(Time),
    R=make_ref(),
    S!{self(),R,itGotLate},
    receive
    {S,R,ok} ->
        done
            end.

woman(S) ->  % Reference to PID of server
    error(not_implemented).

man(S) ->   % Reference to PID of server
    error(not_implemented).

server(Women,Men,false) ->   % Counters for Women and Men, false=it
                             % did not get late yet
    error(not_implemented);

server(Women,Men,true) ->   % Counters for Women and Men, true= it got late
    error(not_implemented);
```

Exercise 8. You are asked to implement a guessing game. A server receives requests to play the game from clients. These requests are of the form `{From,Ref,start}`, where `From` is the Pid of the client, `Ref` is a reference number and `start` is an atom. The server should then:

1. spawn a "servlet" process that plays the game with the client; and
2. then receive new client requests.

Note that by spawning a servlet the server is always responsive to new game requests. The servlet should behave as follows:

- generate a pseudorandom number in the range [0,10];

- wait for guesses from the client of the form `{Pid,Ref,Number}`, where `Pid` is its Pid, `Ref` is a reference number and `Number` is the number the client is guessing.
- answer each message, indicating whether the client has guessed (`gotIt`) or not (`tryAgain`).

The client should keep guessing random numbers. Once it has guessed correctly, both client and servlet simply ends their execution.

You can use the function `rand:uniform(N)` for generating random numbers between 1 and N. Also, you may include helper functions.

```
-module(gg).
-compile(export_all).

start() ->
    spawn(fun server/0).

server() ->
    exit(incomplete).

client(S) ->
    exit(incomplete).
```

Exercise 9. We wish to model a network of temperature sensors. Each sensor maintains a value (representing a temperature). This value may be modified in two ways: it can receive a direct reading (that updates the value of that sensor) or it can update its value based on the values of its neighboring sensors following the protocol which we describe next. Each sensor has a list of its neighboring sensors (you may assume this list is fixed when the sensor is created). All sensors are synchronized by means of the *Timer*: every time the timer ticks, each sensor sends its value to all its neighbors each of which update their own value by taking the average of all received values.

1 Solutions to Selected Exercises

Answer to exercise 4

```
-module(tick).
-compile(export_all).


start(T) ->
    L = [spawn(?MODULE,client,[]) || _ <- lists:seq(1,10)],
    spawn(?MODULE,timer,[T,L]).

timer(Frequency,L) ->
    timer:sleep(Frequency),
    [ Pid!{tick} || Pid <- L],
    timer(Frequency,L).

client() ->
    receive
    {tick} ->
        client()
    end.
```

Answer to exercise 7

```
-module(b).
-compile(export_all).

start(W,M) ->
    S=spawn(?MODULE ,server ,[0]),
    [spawn(?MODULE,woman,[S]) || _ <- lists:seq(1,W)],
    [spawn(?MODULE,man,[S]) || _ <- lists:seq(1,M)].
```

```
8
woman(S) -> % Reference to PID of server
10      S!{self(),woman}.

12  man(S) -> % Reference to PID of server
        Ref = make_ref(),
14      S!{self(),Ref,man},
        receive
16      {S,Ref,ok} ->
            ok
18      end.

20  server(Women) ->
        receive
22      {_From,woman} ->
            server(Women+1);
24      {From, Ref, man} when Women>1 ->
            From!{self(),Ref,ok},
26          server(Women-2)
        end.
```