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 process and two turnstile processes. The turnstile thread should send a `{bump}` message to the counter thread so that the counter can increment its local counter. The counter should also support a `{read,From}` that sends back the value of the counter to the process with pid `From`.

```
-module(ex1).
-compile(export_all).

start(N) ->  %% Spawns a counter and N turnstile clients
    C = spawn(?MODULE,counter_server,[0]),
    [ spawn(?MODULE,turnstile,[C,50]) || _ <- lists:seq(1,N)],
    C.

counter_server(State) ->  %% State is the current value of the counter
    error(not_implemented).

turnstile(C,N) ->  %% C is the PID of the counter, and N the number of
    error(not_implemented).      %% times the turnstile turns
```

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,From}`: done sending strings, send back result to `From`.

```
start() ->
    S = spawn(?MODULE,server,[]),
    [ spawn(?MODULE,client,[S]) || _ <- lists:seq(1,100000)].

client(S) ->  %%
    S!{start,self()},
    S!{add,"h",self()},
    S!{add,"e",self()},
    S!{add,"l",self()},
    S!{add,"l",self()},
    S!{add,"o",self()},
    S!{done,self()},
    receive
    {S,Str} ->
        io:format("Done: ~p~s~n",[self(),Str])
    end.

server() ->
    error(not_implemented).
```

Discuss

- What happens if multiple clients want to use the service?
- Modify your solution so that the server spawns "servlets" that help out individual clients. In this case the interaction would be as follows:

```
start() ->
    S = spawn(?MODULE,server,[]),
    [ spawn(?MODULE,client,[S]) || _ <- lists:seq(1,100000)].

client(S) ->  %%
S!{start,self()},
    receive
      {Servlet} ->
          ok
    end,
    Servlet!{add,"h",self()},
    Servlet!{add,"e",self()},
    Servlet!{add,"l",self()},
    Servlet!{add,"l",self()},
    Servlet!{add,"o",self()},
    Servlet!{done,self()},
    receive
    {Servlet,Str} ->
        io:format("Done: ~p~s~n",[self(),Str])
    end.

server() ->
    error(not_implemented).
```

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, receives 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 a tick message every 100 milliseconds 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 sends back 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 Jets fans. Since the owners are Patriots fans they would like to implement an access control mechanism in which one Jets fan can enter for every two Patriots fans.

1. Complete the template below:

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

start(P,J) ->
```

```
6       S=spawn(?MODULE,server,[0,0]),
7       [spawn(?MODULE,patriots,[S]) || _ <- lists:seq(1,P)],
8       [spawn(?MODULE,jets,[S]) || _ <- lists:seq(1,J)].
9
10  patriots(S) ->  % Reference to PID of server
11      error(not_implemented).
12
13  jets(S) ->   % Reference to PID of server
14      error(not_implemented).
15
16  server(Patriots) ->   % Counters for Patriots available for justifying ingress of Jets
17      error(not_implemented).
```

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

```
1   -module(bar2).
2   -compile(export_all).
3   -compile(nowarn_export_all).
4
5   start(P,J) ->
6       S=spawn(?MODULE,server,[0,0,false]),
7       [spawn(?MODULE,patriots,[S]) || _ <- lists:seq(1,P)],
8       [spawn(?MODULE,jets,[S]) || _ <- lists:seq(1,J)],
9       spawn(?MODULE,itGotLate,[3000,S]).
10
11  itGotLate(Time,S) ->
12      timer:sleep(Time),
13      R=make_ref(),
14      S!{self(),R,itGotLate},
15      receive
16      {S,R,ok} ->
17          done
18              end.
19
20  patriots(S) ->  % Reference to PID of server
21      error(not_implemented).
22
23  jets(S) ->   % Reference to PID of server
24      error(not_implemented).
25
26  server(Patriots,false) ->   % Counters for Patriots available for justifying ingress of Jets, false=it
27                               % did not get late yet
28      error(not_implemented);
29
30  server(Patriots,true) ->   % Counters for Patriots available for justifying ingress of Jets, true= it
31      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() ->
    error(not_implemented).

client(S) ->
    error(not_implemented).
```

Exercise 9. We wish to model a network of processes, called *nodes*, each of which holds some number of *tokens*. Each node has a list of the pids of its neighboring nodes. A timer process sends a tick to all nodes in the system. Upon receiving this tick, each node passes on a copy of all its tokens to all its neighbors.

1 Solutions to Selected Exercises

Answer to exercise 4

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


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

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

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

Answer to exercise 7

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

start(P,J) ->
    S=spawn(?MODULE ,server ,[0]),
    [spawn(?MODULE,patriots,[S]) || _ <- lists:seq(1,P)],
    [spawn(?MODULE,jets,[S]) || _ <- lists:seq(1,J)].

patriots(S) -> % Reference to PID of server
    S!{self(),patriots}.

jets(S) -> % Reference to PID of server
    Ref = make_ref(),
    S!{self(),Ref,jets},
    receive
    {S,Ref,ok} ->
```

```
17          ok
18      end.
19
20  server(Patriots) ->
21      receive
22      {_From,patriots} ->
23          server(Patriots+1);
24      {From, Ref, jets} when Patriots>1 ->
25          From!{self(),Ref,ok},
26          server(Patriots-2)
27      end.
```