DIT027 – Assignment 4

The exercises are supposed to be done in pairs. However, it is allowed to discuss the problems with other groups and share ideas (but not code). There are 4 parts in this assignment. Solving all of them is obligatory.

Don't forget to write comments in your code to make your solutions clearer!

The submission deadline is **22 September 2017** at 23:55.

A test suite will be provided to test the modules that you create. It is required to test your solution with the test suite before submitting, and to include the output from running the test suite on your code.

The test suite will also be used to evaluate your code. Passing the test suite is required, but not sufficient to pass the assignment.

In this assignment you are free to use any **library functions**, unless any specific restrictions are mentioned.

Overcomplicated solutions will not be accepted.

Make sure that you follow ALL the instructions closely!

GOOD LUCK!

Problem 0 – Warm up

This problem is not part of the hand-in. Your job here is to run two Erlang shells connected to the same Erlang VM, and perform simple exercises using it.

First, you need to run two instances of the Erlang shell using special command line options.

```
$ erl –name node1@127.0.0.1
(node1@127.0.0.1)1>
```

On Windows you want to use werl.exe with the same options. When werl.exe gets started, you might be prompted to allow access to the network to it. You might need to allow access to the private networks to it.

The second shell should be started with an additional –remsh parameter. The 'hostname' part should correspond to your hostname, as seen in the prompt of the Erlang shell.

```
$ erl –name node2@127.0.0.1 –remsh node1@127.0.0.1
(node1@127.0.0.1)1>
```

Now you have two shells connected to the same VM. You can register a process on the first one and look it up on the other one.

```
(shell 1)
(node1@127.0.0.1)1> self().
<0.46.0>
(node1@127.0.0.1)2> register(process1, self()).
true
(node1@127.0.0.1)3>
(shell 2)
(node1@127.0.0.1)1> whereis(process1).
<0.46.0>
(node1@127.0.0.1)2>
```

Having that, you can send messages from the process running in shell 2 to the one running in shell 1 by using the registered name of the latter.

```
(shell 2)
(node1@127.0.0.1)2> whereis(process1) ! {msg, a}.
{msg, a}
(node1@127.0.0.1)3>
(shell 1)
(node1@127.0.0.1)3> flush().
Shell got {msg, a}
ok
(node1@127.0.0.1)4>
```

You can also receive messages using the receive statement.

```
(shell 2)
(node1@127.0.0.1)3> whereis(process1) ! {msg, b}.
{msg, b}
(node1@127.0.0.1)4>
(shell 1)
(node1@127.0.0.1)4> receive X -> X end.
{msg, b}
(node1@127.0.0.1)5>
```

After that, the variable X in shell 1 is bound to {msg, b}. If you want to use X in pattern matching as a fresh variable, then you need to forget it first using f(X) or f().

You are now ready to do the exercises:

- Execute a selective receive in shell 1 and verify that it works.
- Send a message to a registered process with the PID of the sending process embedded. The registered process should reply to the sending process using that PID.
- Use a list comprehension to execute the receive-and-send routine from the previous task many times on the registered process (say: 5 times). Verify that the other process can repeatedly send requests and get replies from the registered process. You can start the third shell, also connected to the same VM and verify that the third process can also send requests and receive replies. Also, use a list comprehension on the initiating side to send a request and receive a reply many times.
- Create a module and write a recursive function that receives a message and sends back a response, and does that forever. Run this function on the registered process and verify that other processes can communicate with it. To terminate this function you will need to use ^G. As a result, your registered process will terminate as well, and the new process will need to be reregistered.

Problem 1 – Counter server

You are supposed to implement a simple server that holds a counter and supports two operations: incr() and fetch().

A shell interaction with such a server could look as follows:

```
12> S = assignment4_counter:start().
<0.65.0>
13> assignment4_counter:start().
<0.67.0>
```

Two calls to start/0 should start two different instances of the server, and return their PIDs. The returned PID can then be used to perform operations on the server.

```
14> assignment4_counter:incr(S).
ok
15> assignment4_counter:incr(S).
ok
16> assignment4_counter:fetch(S).
2
17> assignment4_counter:reset(S).
ok
18> assignment4_counter:fetch(S).
0
```

The server is initialised with the integer 0. The two calls to incr/1 increment the counter, which leads to the value 2 being returned by fetch/1. After the counter is subsequently reset using reset/1, fetch/1 returns 0. In addition, the operation stop/1 can be used to terminate the server process.

```
19> assignment4_counter:stop(S).
ok
```

Your task is to implement a server that provides the above functionality starting from the skeleton-module **assignment4_counter**. The export list of the submitted module should be unchanged. The functions that need to be implemented are start/0, init/0, incr/1, fetch/1, reset/1 and stop/1. The functions should behave as follows, consistently with the example above:

start() should spawn a new server process and return its PID. Independent calls to start/0 should create different instances of the server. The server process should run the init/0 function, which should initialise its state with the integer 0.

incr(S) should add 1 to the state of the server identified by the PID S. The function should return the atom ok once the update had been performed.

fetch(S) should read the state of the server identified by the PID S and return it.

reset(S) should reset to 0 the state of the server identified by the PID S. The function should return the atom ok once the update had been performed.

The operations incr/1, fetch/1 and reset/1 should operate correctly when they are all concurrently called from different processes.

stop(S) should terminate the server process whose PID is S and return ok.

None of the functions incr/1, fetch/1, reset/1 or stop/1 need to perform any error checking.

Problem 2 – Storage server

You are supposed to implement a simple **Term storage server.** The server is supposed to be able to store Erlang terms (values) associated with keys. The server (process) should be registered under the name **sts**, thus there is no need to pass around its Pid. Also, the term storage is **per process**, i.e. two different processes are able to store different values for the same key. The PID of the process calling store/fetch/flush is automatically passed to the server.

A shell interaction with such a server could look like (note, here we are storing terms for a single process – the shell process):

```
34> assignment4_store:start().
{ok, <0.97.0>}
35> assignment4_store:start().
{ok, <0.97.0>}
```

I.e. if it is already started, do nothing but return the already started Pid.

```
36> assignment4_store:store(key1, val1).
{ok,no_value}
```

store/2 stores a new on the server, while returning ok and the old value (or no_value, if the key has not been used before!).

```
37> assignment4_store:store(key2, {a, [more, complex], thing}).
{ok,no_value}
38> assignment4_store:fetch(key2).
{ok,{a,[more,complex],thing}}
39> assignment4_store:fetch(key3).
{error,not_found}
```

fetch/1 returns either {ok, Value} or {error, not_found}. fetch/1 does not change the stored term.

```
40> assignment4_store:store(key1, val1_changed).
{ok,val1}
41> assignment4_store:flush().
{ok,flushed}
42> assignment4_store:fetch(key1).
{error, not_found}
43> assignment4_store:stop().
stopped
44> assignment4_store:stop().
already_stopped
```

Stopping the server twice should not be a problem, and flush/0 removes all terms for the **calling** process (i.e. **not for all processes**!).

The skeleton-module exports the functions: start/0, stop/0, init/0, store/2, fetch/1, and flush/0. You should not change this. Your task is to implement the functions and correctly spawn a process running the init function (in turn calling a local server-loop function) in start/0. start/0 should spawn a new server process unless one is already registered, stop/0 should stop the server, and store/2, fetch/1, and flush/0 are wrapper-functions for the message passing between the server and the caller/client. Make sure that stop/0 is synchronized (i.e. waiting until the server is actually stopped). Your code should handle gracefully any combination of operations invoked at the same time by different client processes.

You also need to write some code to handle the actual storage of terms. Since this exercise is about implementing the server, you may use any back-end you like. For example, you may use the dict_* functions you have implemented in the previous assignment, or the proplists module from the standard library.

A more complex interaction with the term storage, using multiple processes, and showing how they interact (or rather don't interact, as one process should not be able to see terms stored by another process!) is provided by the example below. Be aware that several processes are run concurrently, and the order (but not the content!) of the printouts will vary between different executions!

```
-import(assignment4_store, [fetch/1, store/2, flush/0, start/0]).

test() ->
  start(),
  spawn(fun() ->
            io:format("Process ~p started!\n", [self()]),
            store(key1, val1), store(key2, val2),
            io:format("~p fetch(key1): ~p\n", [self(), fetch(key1)]),
            io:format("~p fetch(key2): ~p\n", [self(), fetch(key2)]),
            flush(),
            io:format("~p fetch(key1): ~p\n", [self(), fetch(key1)])
        end),
  spawn(fun() ->
            io:format("Process ~p started!\n", [self()]),
            store(key1, val2), store(key2, val1), %% Values swapped!
            io:format("~p fetch(key1): ~p\n", [self(), fetch(key1)]),
            io:format("~p fetch(key2): ~p\n", [self(), fetch(key2)]),
            store(key1, blafooo),
            io:format("~p fetch(key1): ~p\n", [self(), fetch(key1)])
        end),
  ok.
```

And running the test/0 function:

```
46> assignment4_store_test:test().
Process <0.126.0> started!
Process <0.127.0> started!
ok
<0.126.0> fetch(key1): {ok,val1}
<0.127.0> fetch(key1): {ok,val2}
<0.126.0> fetch(key2): {ok,val2}
<0.127.0> fetch(key2): {ok,val1}
<0.126.0> fetch(key1): {error,not_found}
<0.127.0> fetch(key1): {ok,blafooo}
48> assignment4_store_test:test().
Process <0.171.0> started!
Process <0.170.0> started!
ok
<0.171.0> fetch(key1): {ok,val2}
<0.170.0> fetch(key1): {ok,val1}
<0.170.0> fetch(key2): {ok,val2}
<0.171.0> fetch(key2): {ok,val1}
<0.170.0> fetch(key1): {error,not_found}
<0.171.0> fetch(key1): {ok,blafooo}
```

Problem 3 – Process handling

In the new era of computation, where normal PC's and laptops are having multi-core processors, parallel computation is crucial for program efficiency. In this problem, we will experiment with one simple way of doing this.

Imagine that we have a (very) long list of computations that we need to perform. (Or a list of data where we need to perform a computation for each element in the list.) A simple way to parallelize this would be to split the list in two, and give half of the list to a separate process. (Or in four if you happen to have four cores, etc...) With the lightweight processes in Erlang, we can go a step further: We can create a process for each element in the list. (**Note:** this is overly naïve, and if each computation is small the overhead is too much even for Erlang processes!)

In the skeleton module **assignment4.erl** you are given the mysterious function **task/1**, that we use as an example. **task/1** takes an integer parameter (range 0..100), and performs a long computation, and returns an integer value.

The first task is to implement a function **dist_task/1**. The function should, given this list of numbers, spawn a process for each computation and collect all the computation results. Using this function could look like:

```
133> Data = [17, 9, 3, 12, 11, 24, 27].

[17,9,3,12,11,24,27]

134> assignment4:dist_task(Data).

[375,460,358,511,494,494,324]
```

Here 7 computation processes were spawned and each process calculated one task.

Hint: Make sure that you collect the results in the right order (i.e. that you maintain the same order for the results as for the input).

This computation pattern is a specific example of a general pattern, it is often called ***parallel_map*** or ***pmap***. It works very much like the **lists:map/2** in Erlang. With the difference that each function application is evaluated in a separate process. (**Hint:** the similarity with **lists:map/2** could be used to test your implementation!) You may reuse the code for **pmap/2** that you have seen in the lecture.

Your second task is to implement the more general **pmap/2**, where the first argument is the function to apply and the second argument is the list of data. You should take into account the possibility of a function crashing, i.e. your implementation should work also for the provided **faulty_task/1**, and not hang.

```
135> assignment4:pmap(fun assignment3:task/1, Data).
[375,460,358,511,494,494,324]
136> assignment4:pmap(fun assignment3:faulty_task/1, Data).
[375,460,358,511,494,494,unexpected_error]
137> assignment4:pmap(fun assignment3:faulty_task/1, Data).
[375,460,358,511,494,494,324]
138> assignment4:pmap(fun assignment3:faulty_task/1, Data).
[375,unexpected_error,358,511,494,494,324]
```

Note that this version of **pmap** handles errors differently than the one presented in the lecture.

Hint: Make sure that you actually do things concurrently, that is, dist_task/1 should be more efficient that sequentially calculating the tasks. You could use **timer:tc/3** to measure the performance (be aware that you need to have a multi-core processor in order to actually gain any performance).

Problem 4 – Reasoning backwards

In this problem you are not supposed to submit any code, but write your answer in comments of **assignment4.erl**.

Consider the following module defining a server.

```
-module(a4p4).
-export([start/0, init/0, read/1, add/2, f/1, g/1, test/0]).

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

init() -> loop(0).

loop(N) ->
  receive
    {read, Pid} ->
       Pid ! {value, self(), N},
       loop(N);
    {add, Pid, M} ->
       Pid ! {add_reply, self()},
       loop(N + M)
end.

read(Serv) ->
  Serv ! {read, self()},
  receive {value, Serv, N} -> N end.

add(Serv, N) ->
  Serv ! {add, self(), N},
  receive {add_reply, Serv} -> ok end.
```

The following code that uses the server is also part of the module.

```
f(P) ->
  add(P, 1),
  io:format(”f() read ~p~n”, [read(P)]).

g(P) ->
  add(P, 2),
  io:format(”g() read ~p~n”, [read(P)]).

test() ->
  P = start(),
  spawn(a4p4, f, [P]),
  add(P, 3),
  spawn(a4p4, g, [P]),
  io:format(”test() read ~p~n”, [read(P)]).
```

When your colleague ran the test() function, three lines were printed.

Unfortunately, he dropped the first line during copying from the terminal, so you only know the two last lines:

```
g() read 5
test() read 3
```

You would like to use your reasoning to deduce what the first line was. The VM that ran the code was heavily loaded at that time, so you need to consider all possible orders of execution of code in different processes. Please state what was the first line printed, or list all possibilities if more than one is possible. Please include a short explanation of how you arrived at your answer.