Erlang assignment 2

Parallel and distributed programming

Isak Samsten

Spring, 2025

Introduction

Introductory assignments for parallel Erlang. This week has three assignments to test your knowledge of the concurrency primitives of Erlang. The tasks are awarded a maximum of 20 points. The grades are distributed according to Table 1.

Table 1: Point to grade conversion table

Point	**Grade**
0 – 8	F
9 – 10	E
11 – 12	D
13 – 15	C
16 – 17	B
18 – 20	A

Submission

The following files should be submitted to iLearn:

Problem 1: `double.erl` and `monitor.erl`

Problem 2: `bank.erl`

Problem 3: `pmap.erl`

Problem 4: `gen_worker.erl`

Oral presentation

You are required to deliver an oral presentation of your solutions at Presentation 2.

Problems

Problem 1 (3 points)

Problem 1.1 (1 points)

Write an Erlang module `double` containing the function `start/0`. The function `start/0` creates a process and registers it under the name `double`. The `double`-process accepts messages on the form `Pid, Ref, N` and returns a message with `N` doubled. If the process receives a non-number it should crash.

Example 1.1

```
1> c(double).
2> double:start().
3> Ref = make_ref(), double ! {self(), Ref, 10}.
4> flush().
Shell got {Ref, 20}
5> double ! {self(), Ref, a}.
=ERROR REPORT==== 18-Oct-2019::08:00:13.392934 ===
Error in process <0.85.0> with exit value:
{badarith,[{erlang,'*',[a,2],[]}]}
```

Problem 1.2 (2 points)

Write an Erlang module `monitor` containing the function `start/0`. The `start/0` function should start the `double` process and monitor it. If the `double` process crashes, the `monitor` process should restart it and continue to monitor its execution. Finally, implement `double:double/1`, which takes as input an Erlang term and tries to double it using the `double` process, retrying if the `double` process is (temporarily) down due to the monitor process restarting it.

Example 1.2

```
1> c(double), c(monitor).
2> monitor:start().
3> Ref = make_ref(), double ! {self(), Ref, 10}.
4> flush().
Shell got {..., 20}
5> double ! {self(), Ref, 8}.
6> flush().
Shell got {..., 16}
7> double ! {self(), Ref, a}.
%% the monitor captures the error and restarts the double process
8> 8 = double:double(4).
```

Problem 2 (7 points)

Write an Erlang module named bank `bank` (in a file `bank.erl`). The module should have the following functions:

Function	**Specification**
`start()`	Start a new bank-server with all accounts having zero balance, returns a `pid` to the server.
`balance(Pid, Who)`	Return the balance of `Who` from the server `Pid`. Return `ok` or `no_account`.
`deposit(Pid, Who, X)`	Deposit `X` amount of money to the account `Who` at the server `Pid`. If no account exists a new account for `Who` is opened. Returns `{ok, NewAmount}`.
`withdraw(Pid, Who, X)`	Withdraw `X` amount of money from the account `Who` at the server `Pid`. Returns `{ok, AmountLeft}` or `insufficient_funds`.
`lend(Pid, From, To, X)`	Lend `X` amounts of money from `From` to `To`. Return `ok`, `insufficient_funds` or `{no_account, Who}`, where `Who` is the account that does not exist or the *atom* `both` if neither account exists.

Your solution should be implemented using Erlang processes and the implementation should be robust using `make_ref()` and `monitor/2`. If `Pid` has stopped (crashed, or does not exist) the functions should return `no_bank` (e.g., `no_bank = bank:balance(pid(1,2,3), bob)`).

Example 2

```
1> S = bank:start().
2> no_account = bank:balance(S, bob).
3> {ok, 110} = bank:deposit(S, bob, 110).
4> {ok, 60} = bank:withdraw(S, bob, 50).
5> 60 = bank:balance(S, bob).
6> {no_account, alice} = bank:lend(S, bob, alice, 30).
7> {ok, 0} = bank:deposit(S, alice, 0).
8> ok = bank:lend(S, bob, alice, 30).
9> {ok, 30} = bank:balance(S, bob).
10> {ok, 30} = bank:balance(S, alice).
```

Problem 3 (5 points)

Implement an Erlang module named `pmap` that provide parallel implementations of the higher-order function `map` (i.e., parallel implementations of the `lists:map(Fun, L)` function). The brave can do Problem 4 first and implement 3.1, 3.2 and 3.3 in terms of it.

Problem 3.1 (1 point)

Implement the function `pmap:unordered(Fun, L)` which takes as input a `function Fun` and a list `L`. The function then for each element i in `L` spawns one *worker* process and evaluates `Fun` on the element. The worker respond to the caller which gather the results and return it. To clarify, the result of `pmap:unordered(Fun, L)` is the same as that of `lists:map(Fun, L)` but without guaranteed order.

Example 3.1

```
1> pmap:unordered(fun (X) -> X * 2 end, [1,2,3]).
[2,4,6] % in undefined order
```

Problem 3.2 (2 points)

Implement the function `pmap:unordered(Fun, L, MaxWorkers)` which is functionally equivalent to `pmap:unordered/2` but only spawns `MaxWorkers` *worker*-processes. If `MaxWorkers` is larger than `length(L)`, `MaxWorkers` is set to `length(L)`.

Example 3.1

```
1> pmap:unordered(fun (X) -> X * 2 end, [1,2,3], 2).
[2,4,6] % in undefined order
```

Problem 3.3 (2 points)

Implement the function `pmap:ordered(Fun, L, MaxWorkers)` which is functionally equivalent to `pmap:unordered/3` but the order of the returned list is the same as `L`.

Example 3.1

```
1> [2, 4, 6] = pmap:ordered(fun (X) -> X * 2 end, [1,2,3], 2).
```

Problem 4 (5 points)

Write a behavior-module `gen_worker` that can be used to create work-pools with custom behavior. The `gen_worker` module defines a `handle_work/1`-callback[1] which receives a value and returns {`result, Result`} or `no_result`. The `gen_worker`-module should also handle errors in the user-defined callback function to avoid killing the worker processes. The module also exports the following functions:

Function	**Description**
`Pid = start(Callback, Max)`	Start a work-pool with `Callback` and `Max` processes handling the work. `Pid` is the process identifier of the work-pool.
`stop(Pid)`	Stop `Pid` and all its workers.
`Ref = async(Pid, W)`	Schedule `W` for processing one worker at `Pid`. `Ref` is a unique reference that can be used to `await(Ref)` the result.
`await(Ref)`	Await the result with the unique reference `Ref` created by `async(Pid, W)`. Returns `no_result`, `error` or {`result, Result`}.
`await_all(Refs)`	Await the work for all references in the list `Refs` and return a (possibly empty) list of results. Work resulting in `no_result` should *not* be included in the list.

More specifically, `gen_worker` encapsulate the behavior of a work-pool, i.e., it defines a *master* process that receives and distributes work to *k worker* processes. The worker processes in turn call `Callback:handle_work(Work)` and sends back the result to the *master* process. The *master* process then sends back the result to the caller of `gen_worker:async/2` (note that `async/2` does not wait for the result but return a reference that can be used by `await/1` to wait for the result). The caller can then use `gen_worker:await/1` to `receive` the result.

[1]Implementors should receive a warning if the callback is not exported, i.e., you should use `-callback` to define your callback function.

On the next page, we have a small module `pmap` which uses the `gen_worker` module to implement `pmap:ordered/2`, which can the be used as in Example 4.

Example 4

```
1> c(gen_worker), c(pmap).
2> pmap:ordered(fun (X) -> X*2 end, [1,2,3]).
[2, 4, 6]
```

`pmap.erl`

```
-module(pmap).
-behaviour(gen_worker).
-export([handle_work/1, ordered/2]).

%% Simple handle_work: apply the function to the value
handle_work({Fun, V}) ->
   {result, Fun(V)}.

ordered(Fun, L) ->
   %% Start a work-pool with 2 workers
   WorkPool = gen_worker:start(?MODULE, 2),

   %% Schedule the work asynchronously
   Refs = [gen_worker:async(WorkPool, {Fun, V}) || V <- L],

   %% Await the result
   Result = gen_worker:await_all(Refs),

   %% Stop our work pool
   gen_worker:stop(S),

   %% Return the result
   Result.
```