Advanced Programming

Erlang for Robust Systems

Ken Friis Larsen
kflarsen@diku.dk

Department of Computer Science
University of Copenhagen

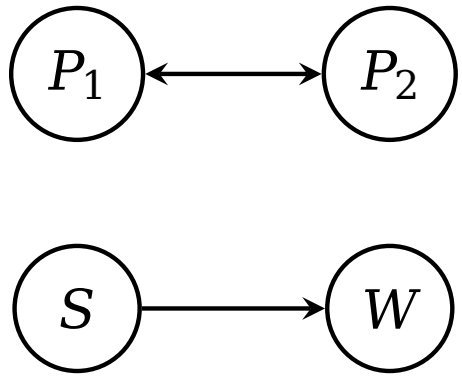
October 10, 2017

Today's Menu

- Recap Linking processes
- Supervisors
- Library code for making robust servers
- Open Telecom Platform (OTP)

Robust Systems

- We need at least two computers(/nodes/processes) to make a robust system: one computer(/node/process) to do what we want, and one to monitor the other and take over when errors happens.
- `link(Pid)` makes a symmetric link between the calling process and `Pid`.
- `monitor(process, Pid)` makes an asymmetric link between the calling process and `Pid`.



Linking Processes

- If we want to handle when a linked process crashes then we need to call `process_flag(trap_exit, true)`.
- Thus, we have the following idioms for creating processes:
  - Idiom 1, I don't care:

    ```
    Pid = spawn(fun() -> ... end)
    ```
  - Idiom 2, I won't live without her:

    ```
    Pid = spawn_link(fun() -> ... end)
    ```
  - Idiom 3, I'll handle the mess-ups:

    ```
    ...
    process_flag(trap_exit, true),
    Pid = spawn_link(fun() -> ... end),
    loop(...).

    loop(State) ->
      receive
        {'EXIT', Pid, Reason} -> HandleMess, loop(State);
        ...
      end.
    ```

Example: Keep ~~Trucking~~ Looping

Suppose that we really most have a phonebook server running at all times. How do we monitor the phonebook server and restart it if (when?) it crashes.

Example: Keep Looping

```
start() -> keep_looping().
request_reply(Pid, Request) -> Ref = make_ref(),
                      Pid ! {self(), Ref, Request},
                      receive {Ref, Response} -> Response end.
keep_looping() ->
    spawn(fun () ->
            process_flag(trap_exit, true),
            Worker = spawn_link(fun() -> loop(#{}) end),
            supervisor(Worker)
          end).
supervisor(Worker) ->
    receive
        {'EXIT', Pid, Reason} ->
            io:format("~p exited because of ~p~n", [Pid,Reason]),
            Pid1 = spawn_link(fun() -> loop(#{}) end),
            supervisor(Pid1);
        Msg -> Pid ! Msg, supervisor(Pid)
    end.
```

Generic Servers

- Goal: Abstract out the difficult handling of concurrency to a generic library
- The difficult parts:
  - The `start-blocking(/async)-loop` pattern
  - Supervisors
  - Hot-swapping of code

Basic Server Library

```
start(Name, Mod) ->
  register(Name, spawn(fun() -> loop(Name, Mod, Mod:init())
                           end)).
blocking(Pid, Request) ->
  Pid ! {self(), Request},
  receive
    {Pid, Reply} -> Reply
  end.
loop(Name, Mod, State) ->
  receive
      {From, Request} ->
          {Reply, State1} = Mod:handle(Request, State),
          From ! {Name, Reply},
          loop(Name, Mod, State1)
  end.
```

Example: Phonebook Callback Module, 1

```
-module(pb).
-import(basicserver, [blocking/2]).

%% Interface
start()          -> basicserver:start(phonebook, pb).
add(Contact)     -> blocking(phonebook, {add, Contact}).
list_all()       -> blocking(phonebook, list_all).
update(Contact) -> blocking(phonebook, {update, Contact}).
```

Example: Phonebook Callback Module, 2

```
%% Callback functions
init() -> #{}.

handle({add, {Name, _, _} = Contact}, Contacts) ->
    case maps:is_key(Name, Contacts) of
        false -> {ok, Contacts#{Name => Contact}};
        true  -> {{error, Name, is_already_there},
                  Contacts}
    end;
handle(list_all, Contacts) ->
    List = maps:to_list(Contacts),
    {{ok, lists:map(fun({_, C}) -> C end, List)},
     Contacts};
handle({update, {Name, _, _} = Contact}, Contacts) ->
    {ok, Contacts#{Name => Contact}}.
```

Hot Code Swapping

```
swap_code(Name, Mod) -> blocking(Name, {swap_code, Mod}).
blocking(Pid, Request) ->
    Pid ! {self(), Request},
    receive {Pid, Reply} -> Reply
    end.
loop(Name, Mod, State) ->
    receive
        {From, {swap_code, NewMod}} ->
            From ! {Name, ok},
            loop(Name, NewMod, State);
        {From, Request} ->
            {Reply, State1} = Mod:handle(Request, State),
            From ! {Name, Reply},
            loop(Name, Mod, State1)
    end.
```

Open Telecom Platform (OTP)

- Library(/framework/platform) for building large-scale, fault-tolerant, distributed applications.
- A central concept is the OTP *behaviour*
- Some behaviours
  - `supervisor`
  - `gen_server`
  - `gen_statem` (or `gen_fsm`)
  - `gen_event`
- See `proc_lib` and `sys` modules for basic building blocks.

Using gen_server

Step 1: Decide module name

Step 2: Write client interface functions

Step 3: Write the six server callback functions:

- `init/1`
- `handle_call/3`
- `handle_cast/2`
- `handle_info/2`
- `terminate/2`
- `code_change/3`

(you can do it by need.)

Flamingo!

Summary

- To make a robust system we need two parts: one to do the job and one to take over in case of errors
- Structure your code into the infrastructure parts and the functional parts.
- Use `gen_server` for building robust servers.
- This week's assignment: Flamingo

Exam

- One week take-home project (3/11–10/11)
- Hand in via Digital Exam
- Max group size is **1** (one)
- The University has a zero-tolerance policy against exam fraud (including assisting).