Advanced Programming

Erlang for Robust Systems

Ken Friis Larsen
kflarsen@diku.dk

Department of Computer Science
University of Copenhagen

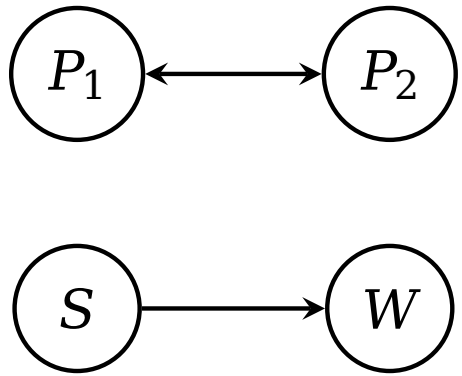
October 11, 2016

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().
blocking(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(dict:new()) end),
            supervisor(Worker)
          end).
supervisor(Worker) ->
    receive
        {'EXIT', Worker, Reason} ->
            io:format("~p exited because of ~p~n", [Worker,Reason]
            Pid1 = spawn_link(fun() -> loop(dict:new()) end),
            supervisor(Pid1);
        Msg -> Worker ! Msg, supervisor(Worker)
    end.
```

Generic Servers

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

Basic Server

```
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() -> dict:new().

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

Server With Transactions

```
blocking(Pid, Request) ->
    Pid ! {self(), Request},
    receive {Pid, {throw, Why}} -> throw(Why)
            {Pid, Reply} -> Reply;
    end.
loop(Name, Mod, State) ->
    receive
        {From, Request} ->
            try Mod:handle(Request, State) of
                {Reply, State1} ->
                    From ! {Name, Reply},
                    loop(Name, Mod, State1)
            catch
                throw : Why ->
                    From ! {Name, {throw, Why}},
                    loop(Name, Mod, State)
            end
    end.
```

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.
```

Transactions and Hot Code Swapping

- Can we combine transactions and hot code swapping?
- How about the supervisor model?
- You have X minutes

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.)

Using gen_statem

Step 1: Decide module name

Step 2: Write client interface functions

Step 3: Write following callback functions:

- `init/1`
- `callback_mode/0` should return `state_functions` or `handle_event_function`
- `terminate/3`
- `code_change/4`
- `handle_event/4` or some `StateName/3`functions

(you can do it by need.)

Callback module for gen_statem, part 1

```
-module(door).
-behaviour(gen_statem).
-export([...]).

start(Code) ->
    gen_statem:start({local, door}, door, lists:reverse(Code

button(Digit) ->
    gen_statem:cast(door, {button, Digit}).

stop() ->
    gen_statem:stop(door).
```

Callback module for gen_statem, part 2

```
% STATEM state functions

locked(cast, {button, Digit}, {SoFar, Code}) ->
    case [Digit|SoFar] of
        Code ->
            do_unlock(),
            {next_state, open, {[], Code}, 5000};
        Incomplete when length(Incomplete)<length(Code) ->
            beep(),
            {next_state, locked, {Incomplete, Code}};
        _Wrong ->
            thats_not_gonna_do_it(),
            {keep_state, {[], Code}}
    end.


open(timeout, _, State) ->
    do_lock(),
```

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.
- Use `gen_statem` (or `gen_fsm`) for servers that can be in different states.
- This week's assignment: Implement Map-Reduce (lecture on Thursday) using OTP behaviours.
- Keep an eye out for arrangement on October 25, 13–16, about elective courses