WEEK-5 PRESENTED BY NOLLEH

LEARN U ERLANG

CHAPTER. 11

MORE ON MULTI PROCESSING

STATE YOUR STATE

- no huge advantage if they are just functions with message
- process act like fridge - store / taking

```
fridge1() ->
  receive
    {From, {store, _Food}} ->
      From ! {self(), ok},
      fridge1();
    {From, {take, _Food}} ->
      From ! {self(), not_found},
      fridge1();
    terminate -> ok
end.
```

STATE YOUR STATE

- pass all

```
fridge2(FoodList) ->
  receive
    {From, {store, Food}} ->
      From ! {self(), ok},
      fridge2([Food|FoodList]);
```

STATE YOUR STATE

- pass all - 2

```
    {From, {take, Food}} ->
      case lists:member(Food, FoodList) of
        true ->
          From ! {self(), {ok, Food}},
          fridge2(lists:delete(Food, FoodList));
        false ->
          From ! {self(), not_found},
          fridge2(FoodList)
      end;
    terminate ->
      ok
  end.
```

STATE YOUR STATE

- test

```
5> Pid ! {self(), {store, bacon}}.
  {<0.33.0>,{store,bacon}}
6> Pid ! {self(), {take, bacon}}.
  {<0.33.0>,{take,bacon}}
7> Pid ! {self(), {take, turkey}}.
  {<0.33.0>,{take,turkey}}
8> flush().
  Shell got {<0.51.0>,ok}
  Shell got {<0.51.0>,{ok,bacon}}
  Shell got {<0.51.0>,not_found}
  ok
```

STATE YOUR STATE

- makes help functions

```
store(Pid, Food) ->
  Pid ! {self(), {store, Food}},
  receive
    {Pid, Msg} -> Msg
  end.

take(Pid, Food) ->
  Pid ! {self(), {take, Food}},
  receive
    {Pid, Msg} -> Msg
  end.
```

STATE YOUR STATE

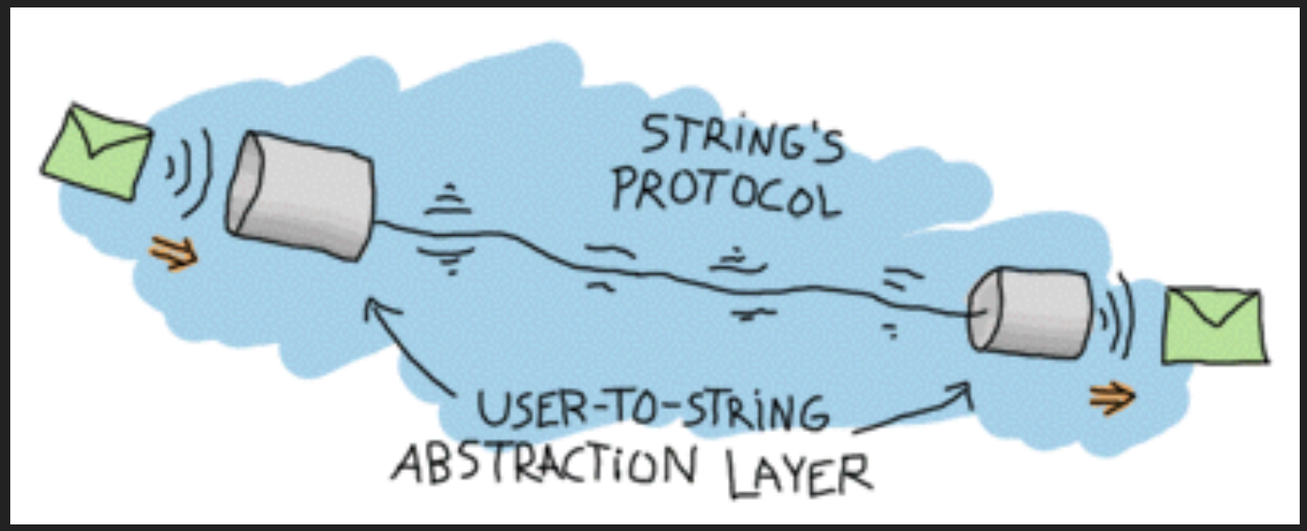
- using it

```
10> f().
  ok
11> Pid = spawn(kitchen, fridge2, [[baking_soda]]).
  <0.73.0>
12> kitchen:store(Pid, water).
  ok
13> kitchen:take(Pid, water).
  {ok,water}
14> kitchen:take(Pid, juice).
not_found
```

STATE YOUR STATE

- uh - hum. let's wrap spawn.

```
start(FoodList) ->
  spawn(?MODULE, fridge2, [FoodList]).
```



STATE YOUR STATE

- using it

```
15> f().
  ok
16> c(kitchen).
  {ok,kitchen}
17> Pid = kitchen:start([rhubarb, dog, hotdog]).
  <0.84.0>
18> kitchen:take(Pid, dog).
  {ok,dog}
19> kitchen:take(Pid, dog).
  not_found
```

STATE YOUR STATE

- in normal case, what happen?

```
20> kitchen:take(pid, dog).
```

  - A message to take food is sent from you (the shell) to the fridge process;
  - Your process switches to receive mode and waits for a new message;
  - The fridge removes the item and sends it to your process;
  - Your process receives it and moves on with its life.

STATE YOUR STATE

- what happen?

```
20> kitchen:take(pid(0,250,0), dog).
```

  - A message to take food is sent from you (the shell) to an unknown process;
  - Your process switches to receive mode and waits for a new message;
  - The unknown process either doesn't exist or doesn't expect such a message and does nothing with it;
  - Your shell process is stuck in receive mode.

TIMEOUT

- timeout

```
receive
   Match -> Expression1
   after Delay ->
      Expression2
end.
```

TIMEOUT

- use timeout 3000

```
store2(Pid, Food) ->
  Pid ! {self(), {store, Food}},
  receive
    {Pid, Msg} ->  Msg
    after 3000 -> timeout
  end.
take2(Pid, Food) ->
  Pid ! {self(), {take, Food}},
  receive
    {Pid, Msg} -> Msg
    after 3000 -> timeout
  end.
```

TIMEOUT

- using it

```
User switch command
  --> k
  --> s
  --> c
Eshell V5.7.5  (abort with ^G)
1> c(kitchen).
  {ok,kitchen}
2> kitchen:take2(pid(0,250,0), dog).
  timeout
```

TIMEOUT

- special cases

```
sleep(T) ->
  receive
  after T -> ok
  end.
```

```
flush() ->
  receive
    _ -> flush()
  after 0 ->
    ok
  end.
```

SELECTIVE RECEIVES

- give priority

```
important() ->
  receive
    {Priority, Message} when Priority > 10 ->
      [Message | important()]
after 0 ->
  normal()
end.
```

SELECTIVE RECEIVES

- give priority

```
normal() ->
  receive
    {_, Message} ->
      [Message | normal()]
after 0 ->
  []
end.
```

SELECTIVE RECEIVES

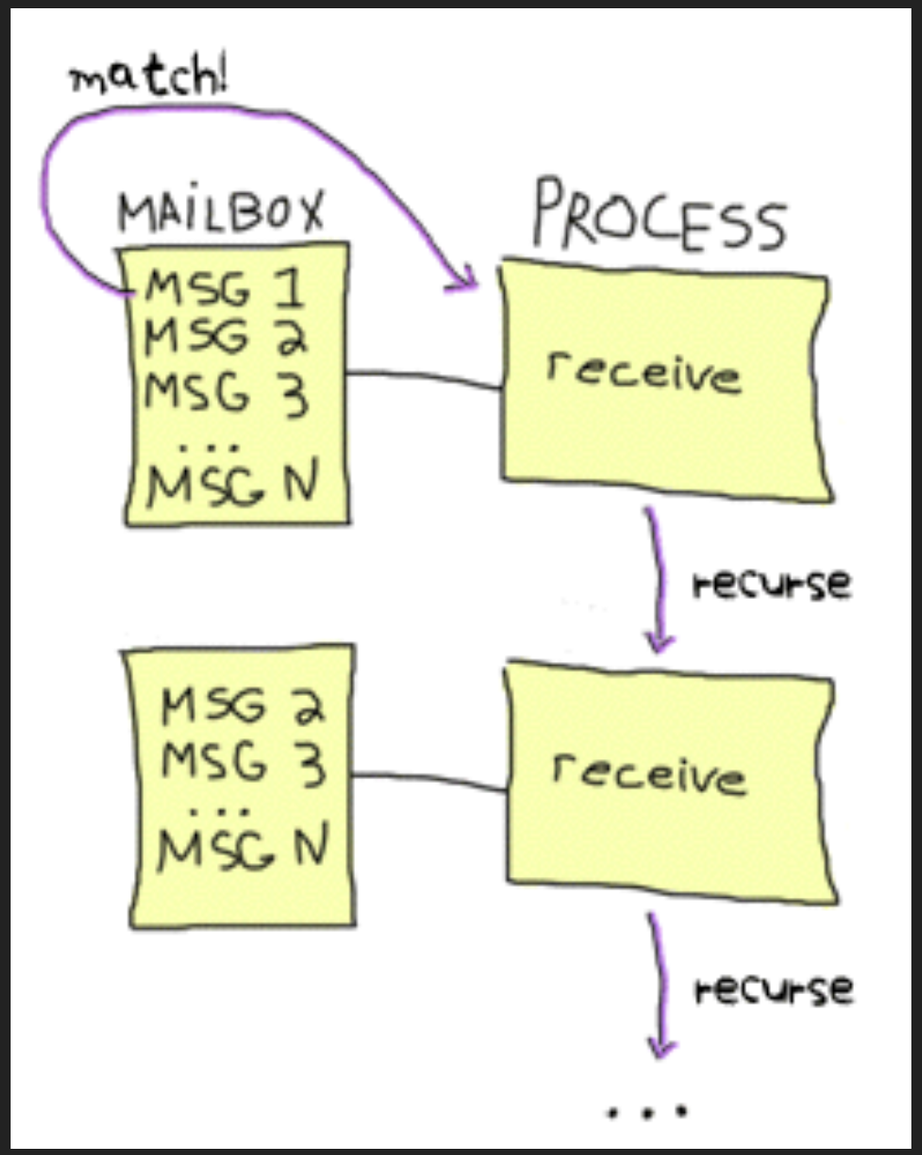
- give priority

```
1> c(multiproc).
  {ok,multiproc}
2> self() ! {15, high}, self() ! {7, low}, self() ! {1, low}, self() ! {17, high}.
  {17,high}
3> multiproc:important().
  [high,high,low,low]
```

- before after routine is started, try to grab every message..

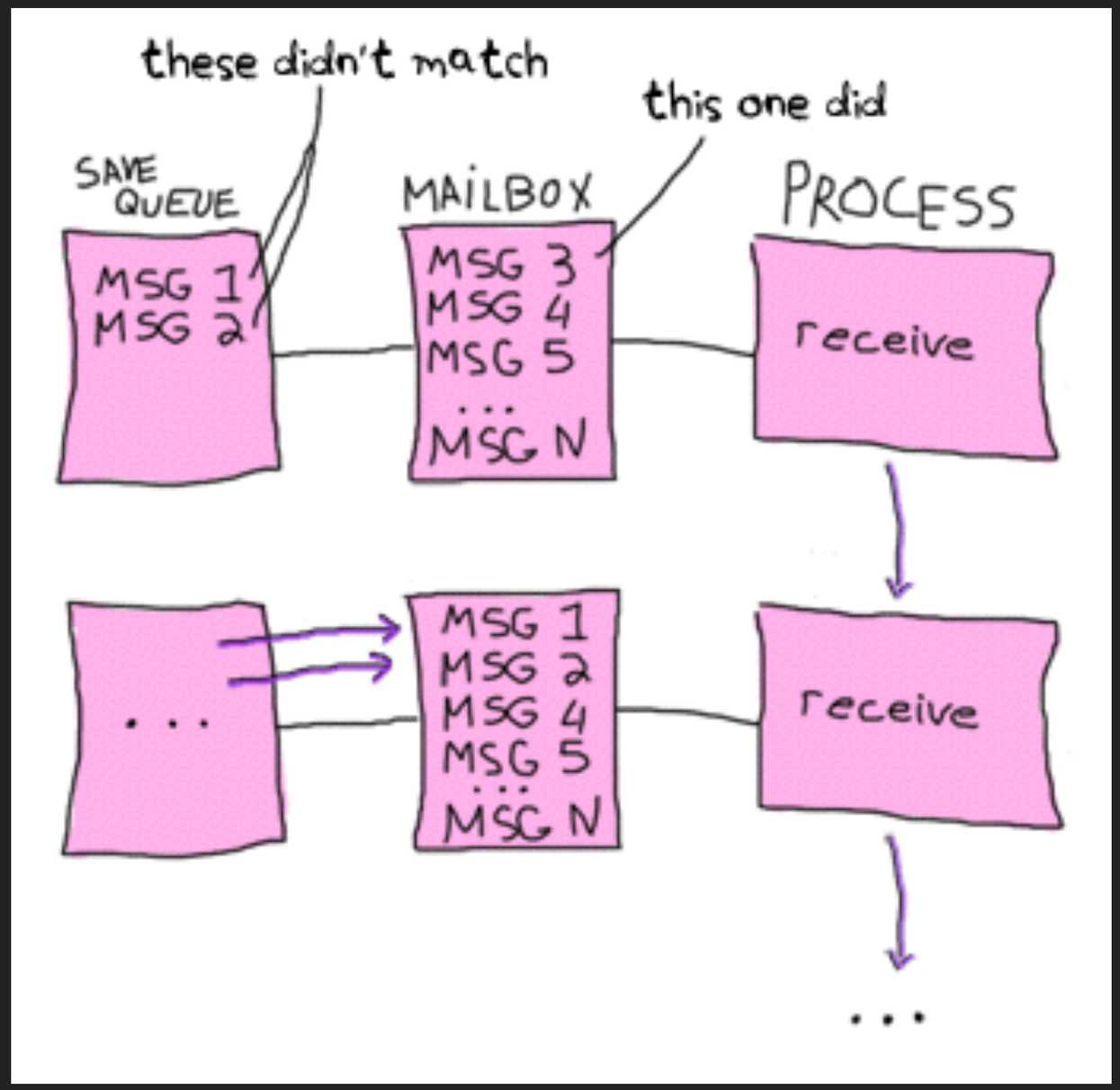
SELECTIVE RECEIVES

- be care!



SELECTIVE RECEIVES

- be care!



SELECTIVE RECEIVES

- be care!
  - if your process has a lot of messages you never care about, reading useful messages will actually take longer and longer (and the processes will grow in size too).
- ask your self
  - why you are getting messages you do not want ?

SELECTIVE RECEIVES

- defensive measure

```
receive
  Pattern1 -> Expression1;
  Pattern2 -> Expression2;
  Pattern3 -> Expression3;
  ...
  PatternN -> ExpressionN;
  Unexpected ->
    io:format("unexpected message ~p~n", [Unexpected])
end.
```

SELECTIVE RECEIVES

- unexpected message out of the mailbox and show a warning
- you can logging it
- more smarter way to implement priority message,..
  - using min_heap / gb_tree (smallest.. largest..)
    - but what if most of message has high-priority… ??
      - optimized/1, make_ref() > R14A

CHAPTER. 12

ERRORS AND PROCESSES

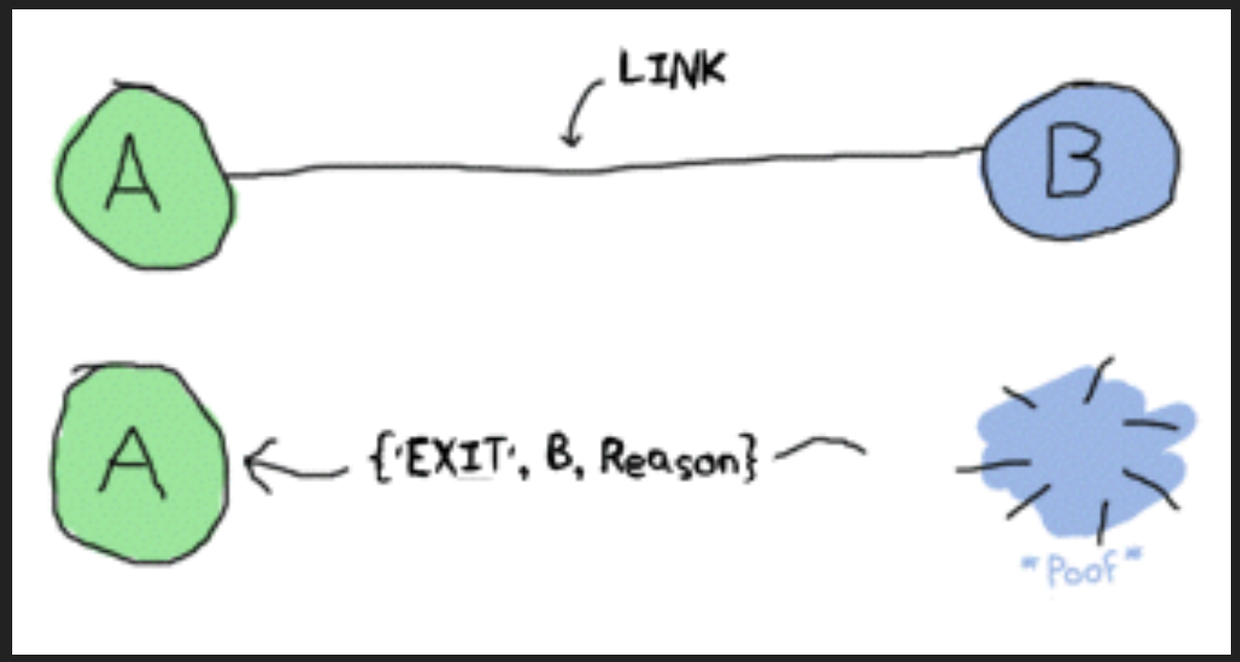
LINKS

- relation

```
myproc() ->
  timer:sleep(5000),
  exit(reason).
```

```
1> c(linkmon).
  {ok,linkmon}
2> spawn(fun linkmon:myproc/0).
  <0.52.0>
3> link(spawn(fun linkmon:myproc/0)).
  true
  ** exception error: reason
```

LINKS



- {'EXIT', B, Reason} message can not be caught with a try ... catch as usual

LINKS

- relation

```
chain(0) ->
  receive
    _ -> ok
  after 2000 ->
    exit("chain dies here")
  end;
chain(N) ->
  Pid = spawn(fun() -> chain(N-1) end),
  link(Pid),
  receive
    _ -> ok
  end.
```

LINKS

- using it

```
4> c(linkmon).
{ok,linkmon}
5> link(spawn(linkmon, chain, [3])).
true
** exception error: "chain dies here"
```

[shell] == [3] == [2] == [1] == [0]
[shell] == [3] == [2] == [1] == *dead*
[shell] == [3] == [2] == *dead*
[shell] == [3] == *dead*
[shell] == *dead*
dead, error message shown
[shell] <-- restarted

LINKS

- link/1 happens more than one step

```
spawn_link/1-3
```

- atomic

IT'S A TRAP!

- error propagation is similar with message passing, but it is special type of message, signal.
- killing part is link. how about quick restarting ?
- system processes

```
process_flag(trap_exit, true)
```

IT'S A TRAP!

- example

```
1> process_flag(trap_exit, true).
  true
2> spawn_link(fun() -> linkmon:chain(3) end).
3> receive X -> X end.
```

```
[shell] == [3] == [2] == [1] == [0]
[shell] == [3] == [2] == [1] == *dead*
[shell] == [3] == [2] == *dead*
[shell] == [3] == *dead*
[shell] <-- {'EXIT,Pid,"chain dies here"} -- *dead*
[shell] <-- still alive!
```

IT'S A TRAP!

- Let's first set the bases to experiment without a system process

```
Exception source: spawn_link(fun() -> ok end)
  Untrapped Result: - nothing -
  Trapped Result: {'EXIT', <0.61.0>, normal}
  The process exited normally, without a problem. Note that this looks
  a bit like the result of catch exit(normal),
  except a PID is added to the tuple to know what processed failed.
```

IT'S A TRAP!

- Let's first set the bases to experiment without a system process

```
Exception source: spawn_link(fun() -> exit(reason) end)
  Untrapped Result: ** exception exit: reason
  Trapped Result: {'EXIT', <0.55.0>, reason}
  The process has terminated for a custom reason. In this case, if there
  is no trapped exit, the process crashes. Otherwise, you get the above
  message
```

IT'S A TRAP!

- Let's first set the bases to experiment without a system process

blah blah...
see Your Book. they are listed in same way..

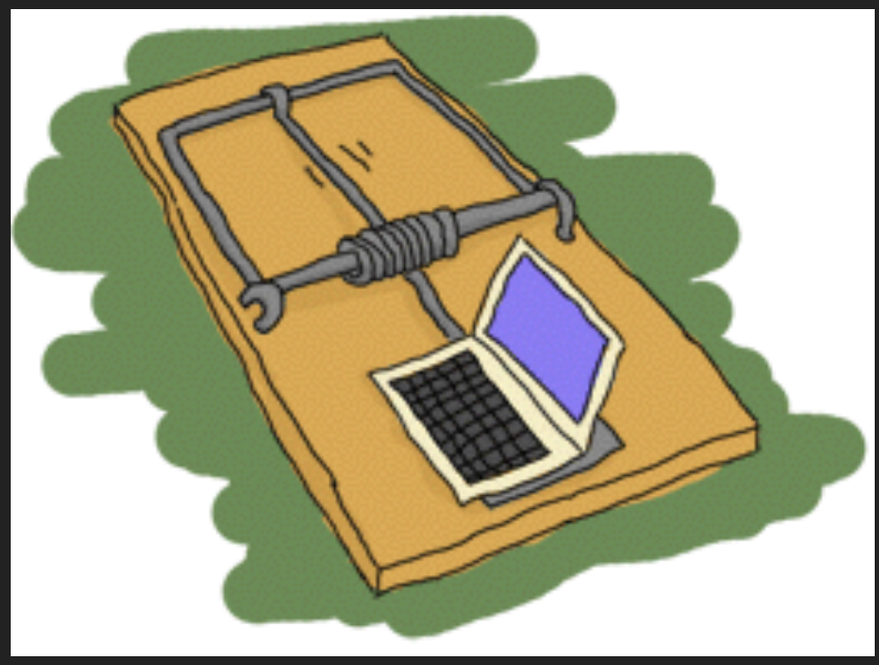
IT'S A TRAP!

- exit/2 - kill another one from a distance, safely

```
exit(self(), normal) % same
exit(spawn_link(fun() -> timer:sleep(50000) end), normal) % no effect

exit(spawn_link(fun() -> timer:sleep(50000) end), kill)
% Trapped Result: {'EXIT', <0.58.0>, killed}

exit(self(), kill)
% Trapped Result: ** exception exit: killed

Exception source: spawn_link(fun() -> exit(kill) end)
% Untrapped Result: ** exception exit: killed
% Trapped Result: {'EXIT', <0.67.0>, kill}
```

IT'S A TRAP!

- kill
  - you might want to brutally murder a process
- killed
  - prevent cascading die



MONITORS

- monitor - special type of link
  - they are unidirectional;
  - they can be stacked.
- if you have 2 or 3 different libraries that you call and they all need to know whether a process is alive or not?
  - not stackable.. (you unlink one, you unlink them all)

MONITORS

- example

```
1> erlang:monitor(process, spawn(fun() -> timer:sleep(500) end)).
  #Ref<0.0.0.77>
2> flush().
  Shell got {'DOWN',#Ref<0.0.0.77>,process,<0.63.0>,normal}
  ok
```

MONITORS

- recv

```
{'DOWN', MonitorReference, process, Pid, Reason}
```

  - reference : to demonitor

- atomic operation

```
spawn_monitor/1-3
```

MONITORS

- demonitor

demonitor/1

```
3> {Pid, Ref} = spawn_monitor(fun() -> receive _ -> exit(boom) end
   end).
   {<0.73.0>,#Ref<0.0.0.100>}
4> erlang:demonitor(Ref).
   true
5> Pid ! die.
   die
6> flush().
   ok
```

MONITORS

- demonitor

demonitor/2

```
8> {Pid, Ref} = spawn_monitor(fun() -> receive _ -> exit(boom) end
end).
  {<0.35.0>,#Ref<0.0.0.35>}
9> Pid ! die.
  die
10> erlang:demonitor(Ref, [flush, info]).
  false
11> flush().
  ok
```

NAMING PROCESSES

- another problem 1 - 1

```
start_critic() ->
  spawn(?MODULE, critic, []).

judge(Pid, Band, Album) ->
  Pid ! {self(), {Band, Album}},
  receive
    {Pid, Criticism} -> Criticism
  after 2000 ->
    timeout
  end.
```

NAMING PROCESSES

- another problem 1- 2

```
critic() ->
  receive
    {From, {"Rage Against the Turing Machine", "Unit Testify"}} ->
      From ! {self(), "They are great!"};
    {From, {"System of a Downtime", "Memoize"}} ->
      From ! {self(), "They're not Johnny Crash but they're good."};
    {From, {"Johnny Crash", "The Token Ring of Fire"}} ->
      From ! {self(), "Simply incredible."};
    {From, {_Band, _Album}} ->
      From ! {self(), "They are terrible!"}
  end,
  critic().
```

NAMING PROCESSES

- another problem 1- 3

```
1> c(linkmon).
  {ok,linkmon}
2> Critic = linkmon:start_critic().
  <0.47.0>
3> linkmon:judge(Critic, "Genesis", "The Lambda Lies Down on
Broadway").
  "They are terrible!”

4> exit(Critic, solar_storm).
  true
5> linkmon:judge(Critic, "Genesis", "A trick of the Tail Recursion").
  timeout
```

NAMING PROCESSES

- another problem 1- 4

```
start_critic2() ->
  spawn(?MODULE, restarter, []).
restarter() ->
  process_flag(trap_exit, true),
  Pid = spawn_link(?MODULE, critic, []),
    receive
      {'EXIT', Pid, normal} -> % not a crash
        ok;
      {'EXIT', Pid, shutdown} -> % manual termination, not a crash
        ok;
      {'EXIT', Pid, _} ->
  restarter()
end.
```

NAMING PROCESSES

- there is no way to find the Pid of the critic !!!

```
start_critic2() ->
  spawn(?MODULE, restarter, []).
restarter() ->
  process_flag(trap_exit, true),
  Pid = spawn_link(?MODULE, critic, []),
    receive
      {'EXIT', Pid, normal} -> % not a crash
        ok;
      {'EXIT', Pid, shutdown} -> % manual termination, not a crash
        ok;
      {'EXIT', Pid, _} ->
  restarter()
end.
```

NAMING PROCESSES

- there is no way to find the Pid of the critic !!!

```
erlang:register/2
```

- If the process dies, it will automatically lose its name or you can also use unregister/1 to do it manually

```
erlang:unregister/2
```

- get list all registered process

```
erlang:registered/0 ..... in shell, regs()
```

NAMING PROCESSES

- solve problem with register

```
restarter() ->
  process_flag(trap_exit, true),
  Pid = spawn_link(?MODULE, critic, []),
  register(critic, Pid),
  receive
    {'EXIT', Pid, normal} -> % not a crash
      ok;
    {'EXIT', Pid, shutdown} -> % manual termination, not a crash
      ok;
    {'EXIT', Pid, _} ->
  restarter()
end.
```

NAMING PROCESSES

- solve problem with register

```
judge2(Band, Album) ->
  critic ! {self(), {Band, Album}},
  Pid = whereis(critic),
  receive
    {Pid, Criticism} -> Criticism
  after 2000 ->
    timeout
  end.
```

NAMING PROCESSES

- no, it ISN'T solving problem, completely.

1. critic ! Message
2. critic receives
3. critic replies
4. critic dies
5. whereis fails
6. critic is restarted
7. code crashes

1. critic ! Message
2. critic receives
3. critic replies
4. critic dies
5. critic is restarted
6. whereis picks up wrong pid
7. message never matches

- critic: shared state / race condition

NAMING PROCESSES

- solve problem considering pid's differentiable 1

```
judge2(Band, Album) ->
  Ref = make_ref(),
  critic ! {self(), Ref, {Band, Album}},
  receive
    {Ref, Criticism} -> Criticism
  after 2000 ->
    timeout
  end.
```

NAMING PROCESSES

- solve problem considering pid's differentiable 2

```
critic2() ->
  receive
    {From, Ref, {"Rage Against the Turing Machine", "Unit Testify"}} ->
      From ! {Ref, "They are great!"};
  {From, Ref, {"System of a Downtime", "Memoize"}} ->
    From ! {Ref, "They're not Johnny Crash but they're good."};
  {From, Ref, {"Johnny Crash", "The Token Ring of Fire"}} ->
    From ! {Ref, "Simply incredible."};
  {From, Ref, {_Band, _Album}} ->
    From ! {Ref, "They are terrible!"}
end,
critic2().
```

NAMING PROCESSES

- solve problem considering pid's differentiable 3

```
6> c(linkmon).
  {ok,linkmon}
7> linkmon:start_critic2().
  <0.55.0>
8> linkmon:judge2("The Doors", "Light my Firewall").
  "They are terrible!"
9> exit(whereis(critic), kill).
  true
10> linkmon:judge2("Rage Against the Turing Machine",
"Unit Testify").
  "They are great!"
```