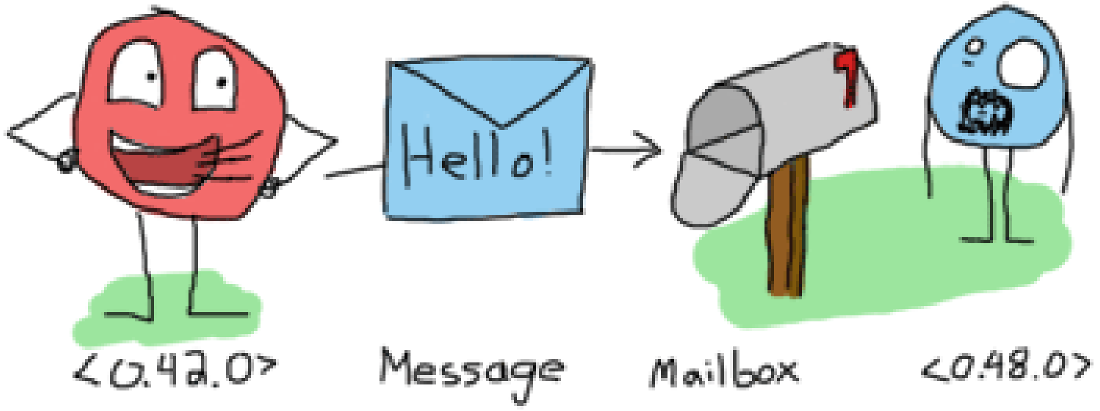
IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

Ioana Leustean



http://learnyousomeerlang.com/the-hitchhikers-guide-to-concurrency#dont-panic

➢Bibliografie

Joe Armstrong, Programming Erlang, Second Edition 2013

Fred Hébert, Learn You Some Erlang For Great Good, 2013
[Varianta online](#)

Jim Larson, Erlang for Concurrent Programming, ACM Queue, 2008

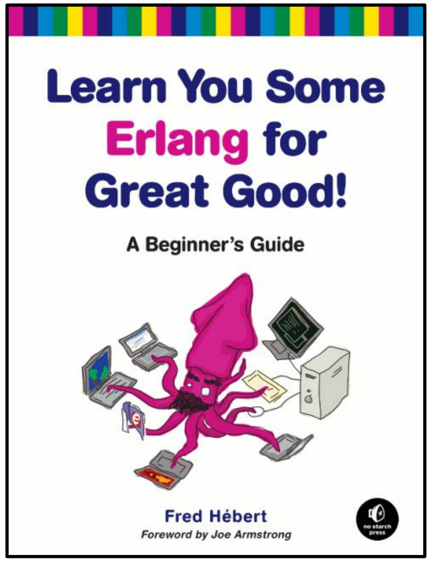
ACTOR MODEL

"Erlang's actor model can be imagined as a world where everyone is sitting alone in their own room and can perform a few distinct tasks. Everyone communicates strictly by writing letters and that's it. While it sounds like a boring life (and a new age for the postal service), it means you can ask many people to perform very specific tasks for you, and none of them will ever do something wrong or make mistakes which will have repercussions on the work of others; they may not even know the existence of people other than you (and that's great).

To escape this analogy, Erlang forces you to write actors (processes) that will share no information with other bits of code unless they pass messages to each other. Every communication is explicit, traceable and safe."

Fred Hébert, Learn You Some Erlang For Great Good

http://learnyousomeerlang.com/introduction#what-is-erlang


Learn You Some Erlang for Great Good!
A Beginner's Guide
Fred Hébert
Foreword by Joe Armstrong

Varianta online

"Messages between Erlang processes are simply valid Erlang terms. That is, they can be lists, tuples, integers, atoms, pids, and so on.

Each process has its own input queue for messages it receives. New messages received are put at the end of the queue. When a process executes a receive, the first message in the queue is matched against the first pattern in the receive. If this matches, the message is removed from the queue and the actions corresponding to the pattern are executed.

However, if the first pattern does not match, the second pattern is tested. If this matches, the message is removed from the queue and the actions corresponding to the second pattern are executed. If the second pattern does not match, the third is tried and so on until there are no more patterns to test. If there are no more patterns to test, the first message is kept in the queue and the second message is tried instead. If this matches any pattern, the appropriate actions are executed and the second message is removed from the queue (keeping the first message and any other messages in the queue). If the second message does not match, the third message is tried, and so on, until the end of the queue is reached. If the end of the queue is reached, the process blocks (stops execution) and waits until a new message is received and this procedure is repeated."

http://erlang.org/doc/getting_started/conc_prog.html

- Concurenta in Erlang este implementata folosind urmatoarele primitive:

```
Pid = spawn (fun)

Pid = spawn (module, fct, args)

Pid ! Message

receive … end
receive … after … end
```

➢ Erori in progamarea concurenta

"Imagine a system with only one sequential process. If this process dies, we might be in deep trouble since no other process can help. For this reason, sequential languages have concentrated on the prevention of failure and an emphasis on *defensive programming*.

Error handling in concurrent Erlang programs is based on the idea of *remote detection and handling of errors*. Instead of handling an error in the process where the error occurs, we let the process die and correct the error in some other process."

Joe Armstrong, Programming Erlang, Second Edition 2013

➢ Erori in progamarea concurenta

"When we design a fault-tolerant system, we assume that errors will occur, that processes will crash, and that machines will fail. Our job is to detect the errors after they have occurred and correct them if possible.

The Erlang philosophy for building fault-tolerant software can be summed up in two easy-to-remember phrases: “Let some other process fix the error” and “Let it crash.”

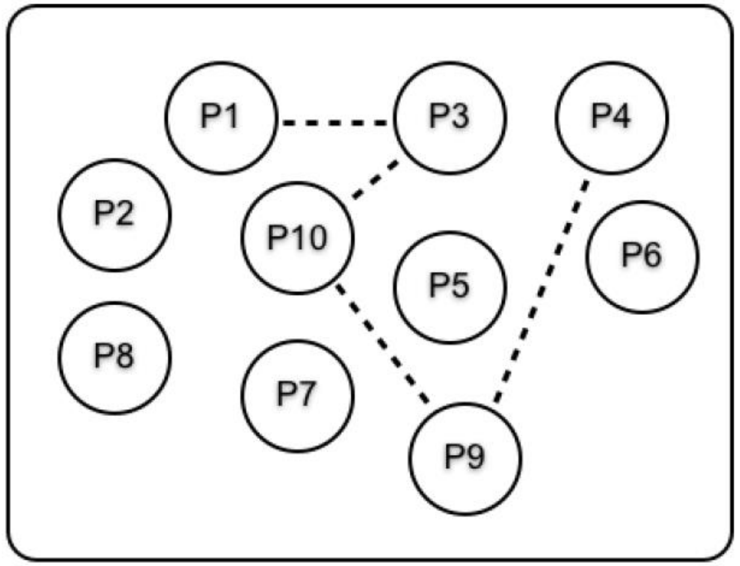
Joe Armstrong, Programming Erlang, Second Edition 2013

➢ Erori in programarea concurenta

- procesele pot fi legate, iar legatura intre procese se creaza cu functia

link(Pid)

- legatura create cu link este bidirectionala
- cand un proces se termina, el trimite un semnal de eroare tuturor proceselor legate de el



Joe Armstrong, Programming Erlang, Second Edition 2013

➢ Mesaje si semnale de eroare

- procesele comunica prin mesaje si semnale de eroare
- mesajele sunt trimise cu **send**
- cand un proces se termina (normal sau cu eroarea) trimite automat
  un semnal de eroare tuturor proceselor de care este legat
- cand un process se termina, el emite un *exit reason*
- un process se termina normal cand *exit reason* este atomul **normal**

exit/1 exit/2

- un process se poate termina singur prin apelul functiei **exit(reason)** ; in acest caz
  el va trimite un semnal de eroare tuturor mesajelor de care este legat
- un proces poate trimite un semnal de eroare fals apeland
  exit(Pid, Reason); in acest caz, Pid va primi semnalul de eroare dar procesul initial
  nu se termina.

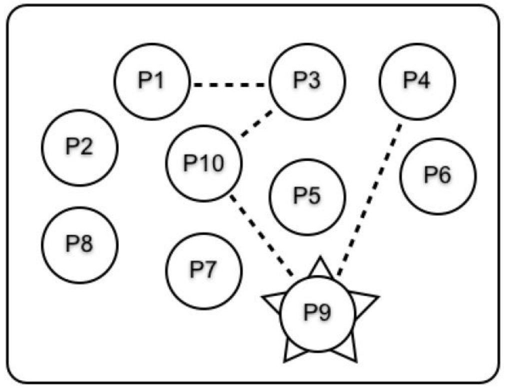
➢Procese normale si procese sistem

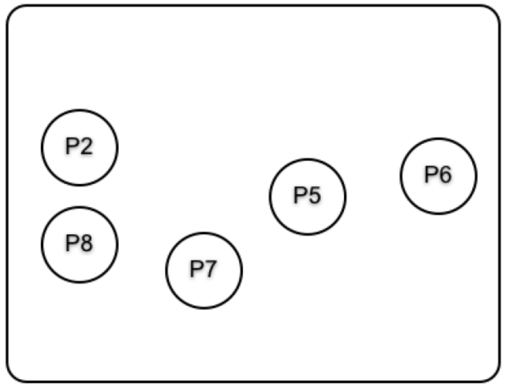
- exista doua tipuri de procese: procese normale si procese system
- un process normal devine process system prin evaluarea expresiei

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

➢Primirea semnalelor de eroare

Cand un proces normal primeste un semnal de eroare si *exit reason* nu este **normal** atunci procesul se termina si trimite semnalul de eroare proceselor cu care este legat.





Joe Armstrong, Programming Erlang, Second Edition 2013

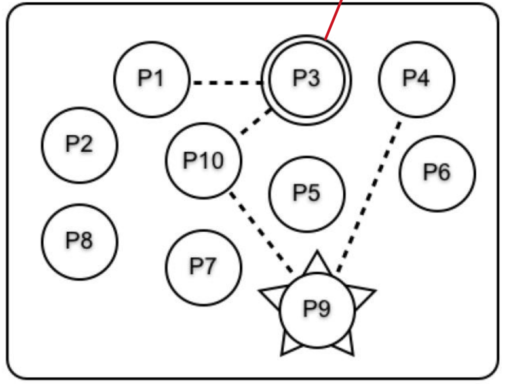
➢Primirea semnalelor de eroare

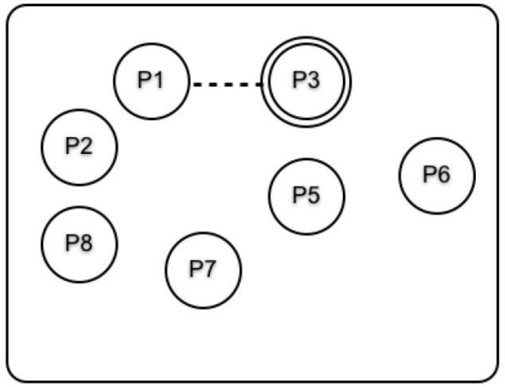
process_flag(trap_exit,true)
-- trap the exit signals

- Cand un proces sistem primeste un semnal de eroare, il transforma intr-un mesaj de forma

  {'EXIT', Pid, Why}

  Unde Pid este identitatea procesului care s-a terminat si Why este *exit reason* (cand procesul nu se termina cu eroare, Why este **normal**)





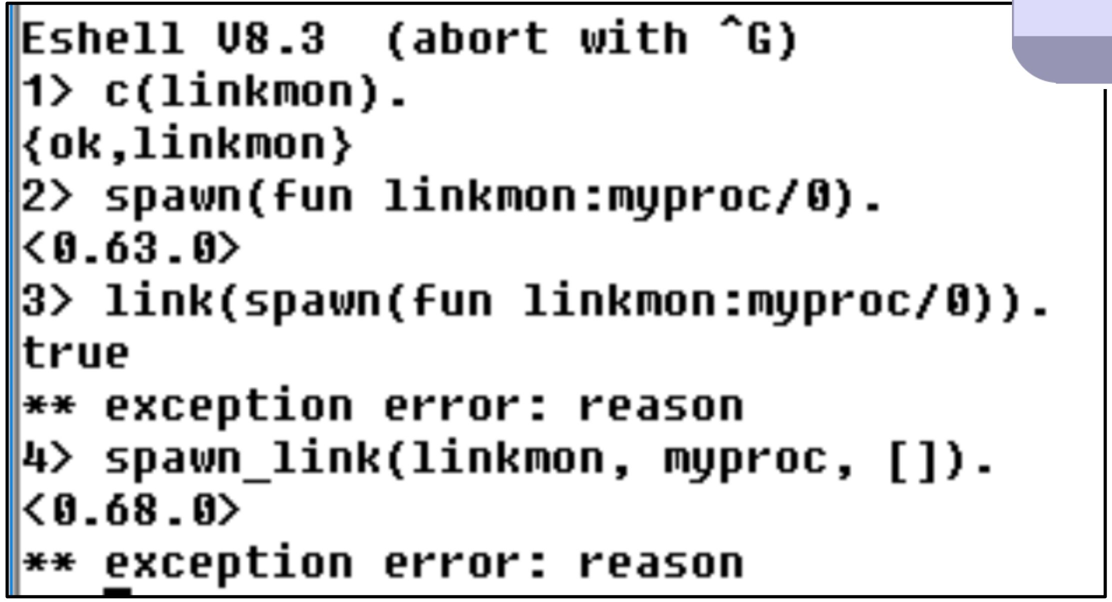
- Procesele sistem opresc propagarea erorilor.

Joe Armstrong, Programming Erlang, Second Edition 2013

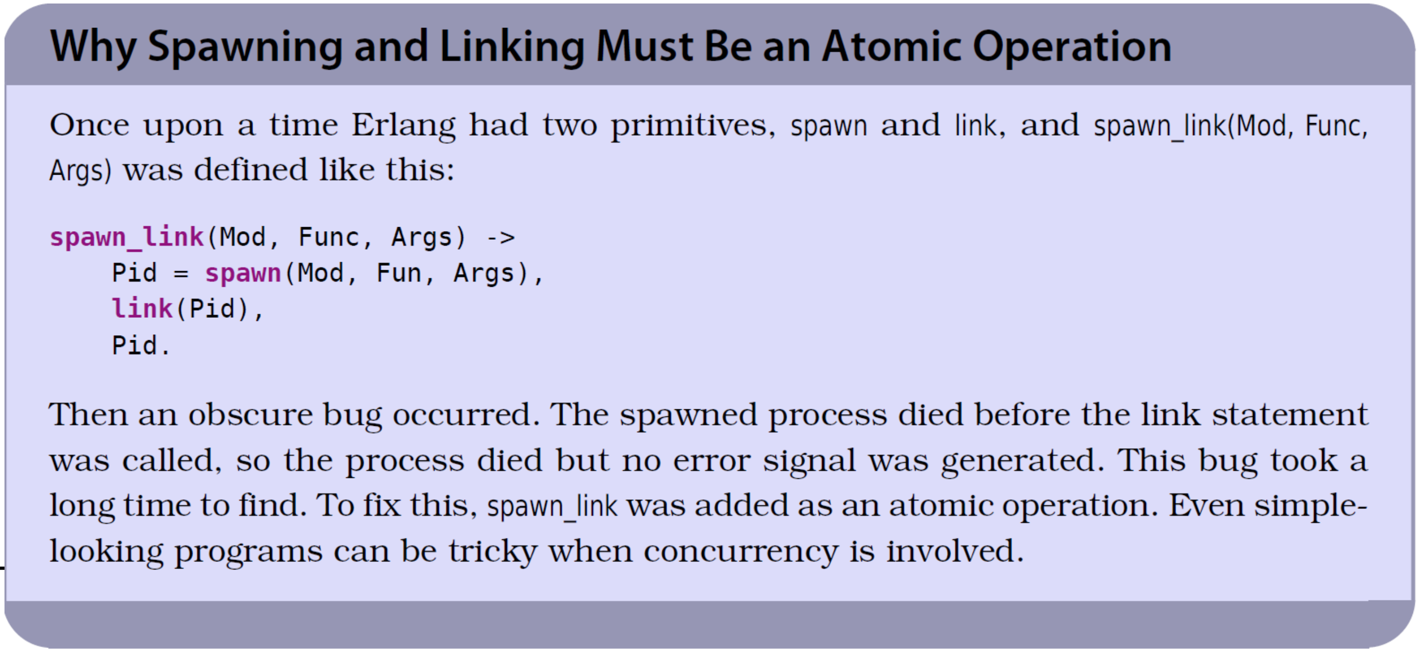
Daca un proces system pimeste mesajul **kill** atunci se termina. Mesajele **kill** sunt generate prin apeluri **exit(Pid, kill)**

➢ link() si spawn_link()

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



```
Eshell V8.3  (abort with ^G)
1> c(linkmon).
{ok,linkmon}
2> spawn(fun linkmon:myproc/0).
<0.63.0>
3> link(spawn(fun linkmon:myproc/0)).
true
** exception error: reason
4> spawn_link(linkmon, myproc, []).
<0.68.0>
** exception error: reason
```



Why Spawning and Linking Must Be an Atomic Operation

Once upon a time Erlang had two primitives, spawn and link, and spawn_link(Mod, Func, Args) was defined like this:

```
spawn_link(Mod, Func, Args) ->
    Pid = spawn(Mod, Fun, Args),
    link(Pid),
    Pid.
```

Then an obscure bug occurred. The spawned process died before the link statement was called, so the process died but no error signal was generated. This bug took a long time to find. To fix this, spawn_link was added as an atomic operation. Even simple-looking programs can be tricky when concurrency is involved.

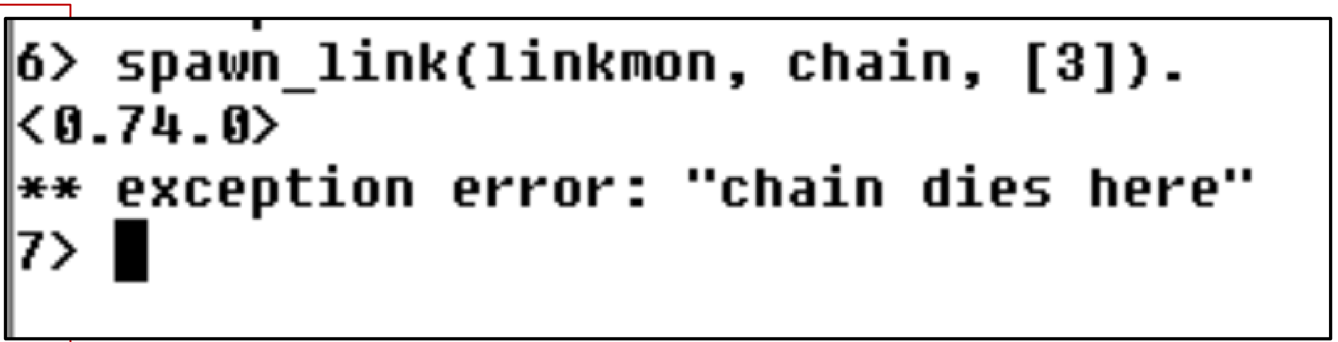
Joe Armstrong, Programming Erlang, Second Edition 2013

spawn_link (Module, Function, [arguments])

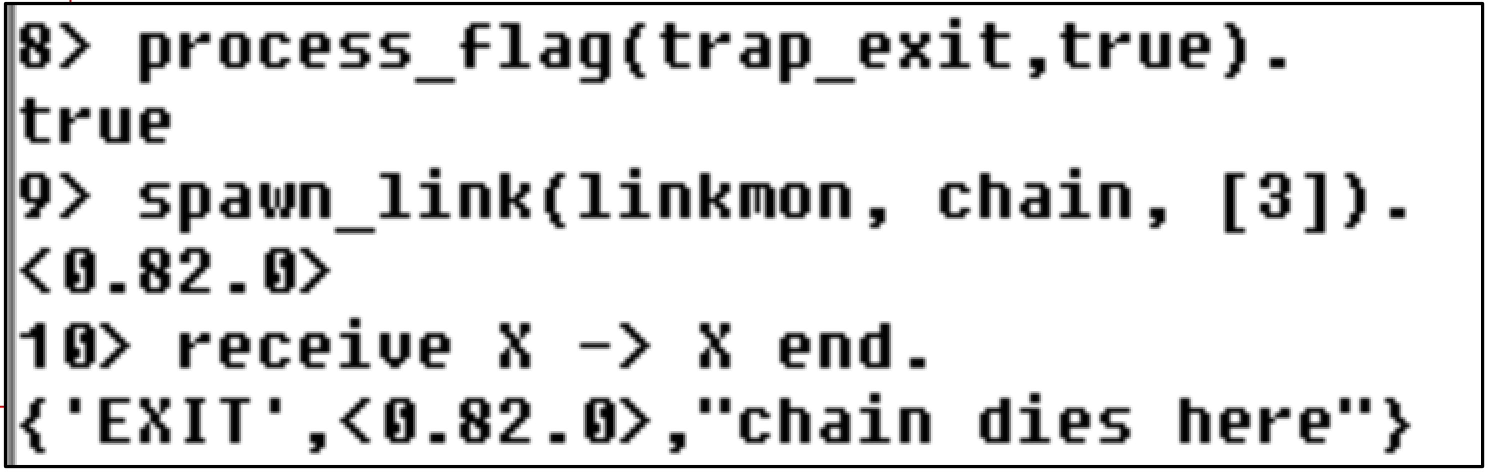
Un grup de procese care mor impreuna

```
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.
```

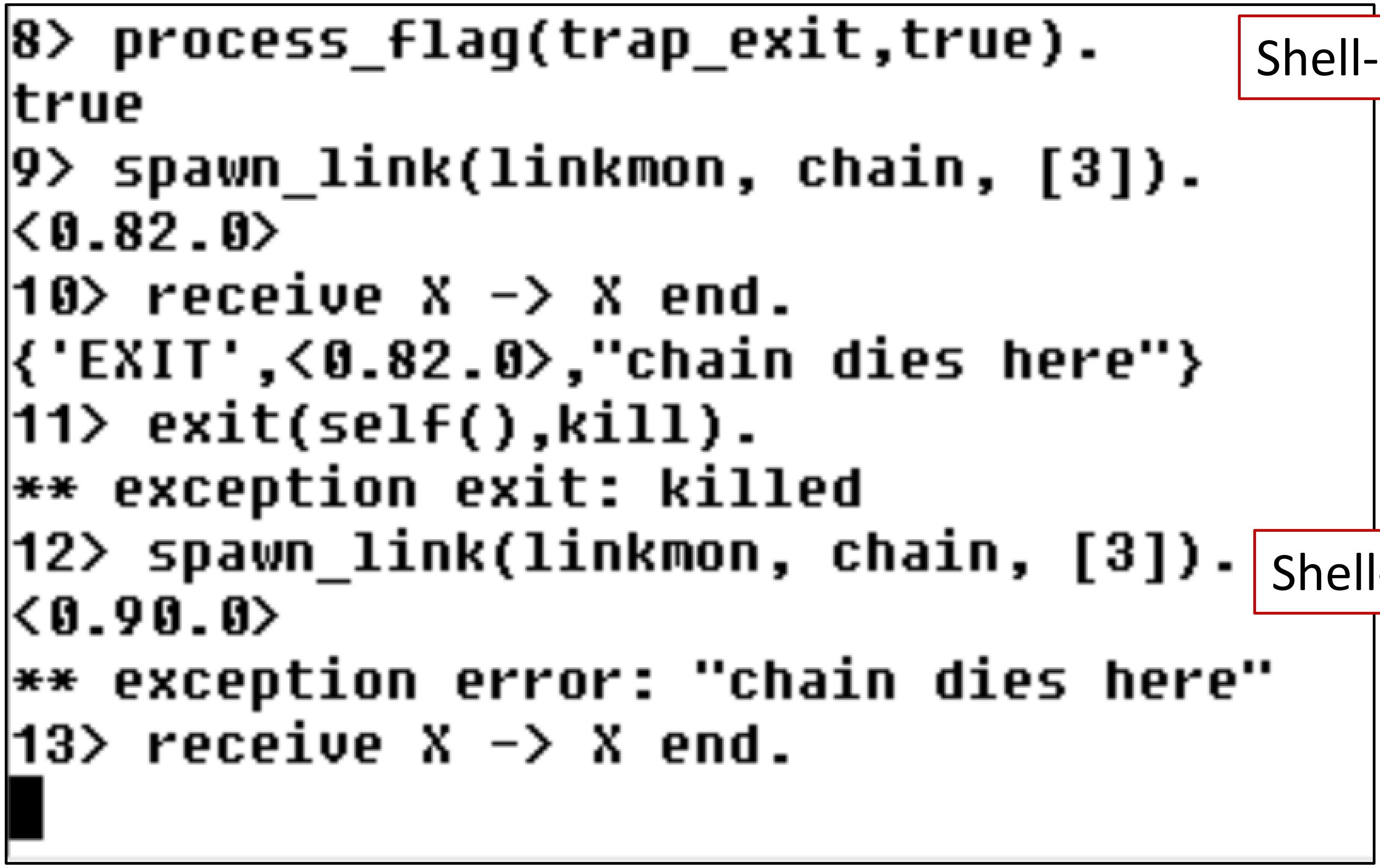


```
6> spawn_link(linkmon, chain, [3]).
<0.74.0>
** exception error: "chain dies here"
7> 
```



```
8> process_flag(trap_exit,true).
true
9> spawn_link(linkmon, chain, [3]).
<0.82.0>
10> receive X -> X end.
{'EXIT',<0.82.0>,"chain dies here"}
```

Fred Hébert, Learn You Some Erlang For Great Good, 2013



```
8> process_flag(trap_exit,true).
true
9> spawn_link(linkmon, chain, [3]).
<0.82.0>
10> receive X -> X end.
{'EXIT',<0.82.0>,"chain dies here"}
11> exit(self(),kill).
** exception exit: killed
12> spawn_link(linkmon, chain, [3]).
<0.90.0>
** exception error: "chain dies here"
13> receive X -> X end.
```

Shell-ul este process sistem

Shell-ul nu este process sistem

➢ Exemplu server-client

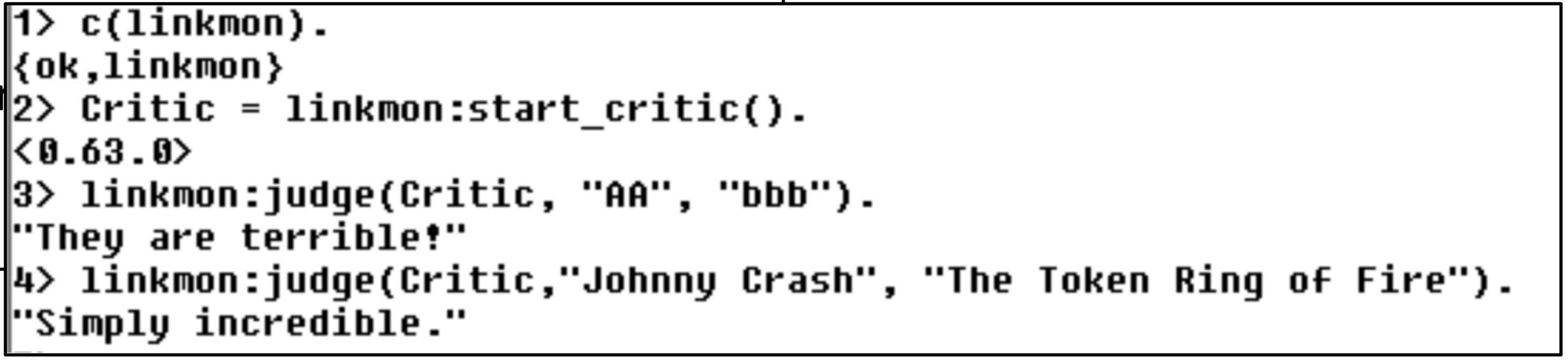
server (critic)

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

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 terr
end,
critic().
```

client (judge)

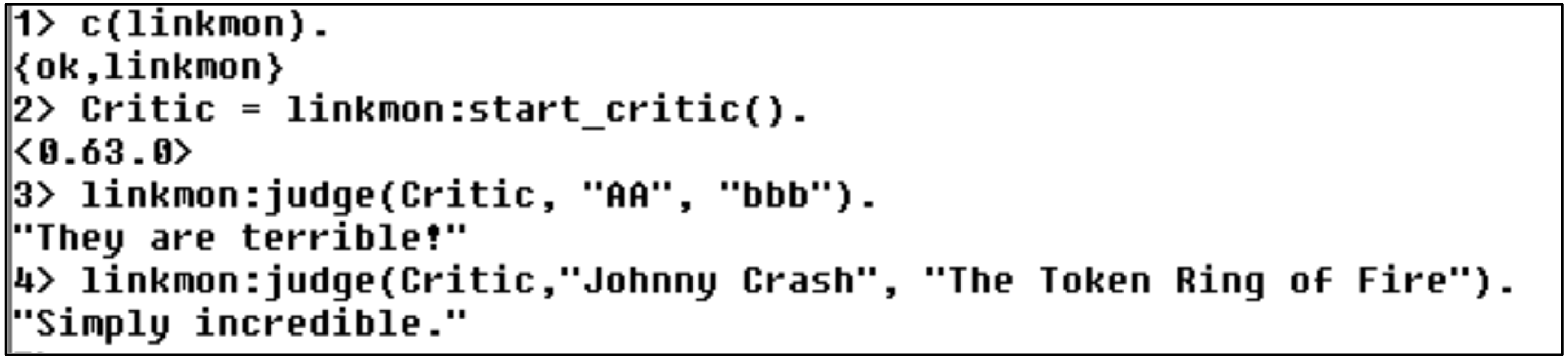
```
judge(Pid, Band, Album) ->
Pid ! {self(), {Band, Album}},
receive
    {Pid, Criticism} -> Criticism
after 2000 ->
timeout
end.
```



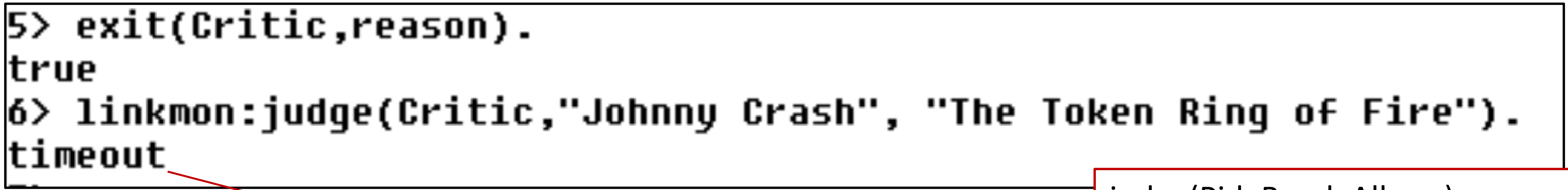
```
1> c(linkmon).
{ok,linkmon}
2> Critic = linkmon:start_critic().
<0.63.0>
3> linkmon:judge(Critic, "AA", "bbb").
"They are terrible!"
4> linkmon:judge(Critic,"Johnny Crash", "The Token Ring of Fire").
"Simply incredible."
```

linkmon.erl

http://learnyousomeerlang.com/errors-and-processes



```
1> c(linkmon).
{ok,linkmon}
2> Critic = linkmon:start_critic().
<0.63.0>
3> linkmon:judge(Critic, "AA", "bbb").
"They are terrible!"
4> linkmon:judge(Critic,"Johnny Crash", "The Token Ring of Fire").
"Simply incredible."
```



```
5> exit(Critic,reason).
true
6> linkmon:judge(Critic,"Johnny Crash", "The Token Ring of Fire").
timeout
```

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

linkmon.erl
http://learnyousomeerlang.com/errors-and-processes

➢ Proces sistem supervisor (restarter)

restarter/supervizor

server

```
critic() ->
 .....
```

client

```
judge1(Band, Album) ->
critic ! {self(), {Band, Album}},
Pid = whereis(critic),
receive
{Pid, Criticism} -> Criticism
after 2000 ->
timeout
end.
```

```
start_critic1() ->
spawn(?MODULE, restarter, []).

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
                                ok;
    {'EXIT', Pid, _} ->
                           restarter()
end.
```

http://learnyousomeerlang.com/errors-and-processes

Serverul este repornit

server

```
critic() ->
  .....
```

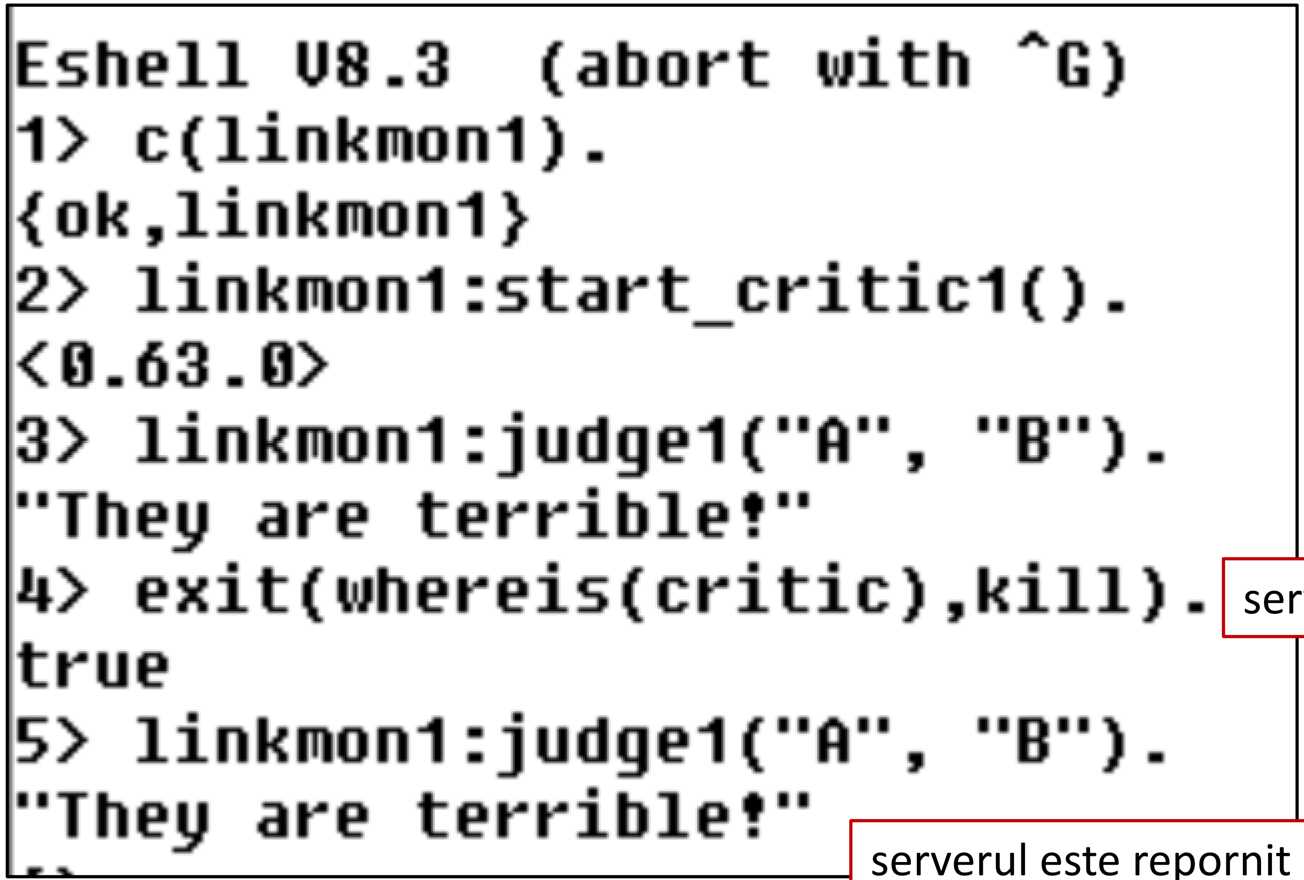
client

```
judge1(Band, Album) ->
......
```

supervizor

```
start_critic1() ->
spawn(?MODULE, restarter, []).

restarter() ->
process_flag(trap_exit, true),
.....
end.
```



```
Eshell V8.3  (abort with ^G)
1> c(linkmon1).
{ok,linkmon1}
2> linkmon1:start_critic1().
<0.63.0>
3> linkmon1:judge1("A", "B").
"They are terrible!"
4> exit(whereis(critic),kill).
true
5> linkmon1:judge1("A", "B").
"They are terrible!"
```

serverul moare

serverul este repornit de supervizor

server

```
critic() ->
  .....
```

client

```
judge1(Band, Album) ->
critic ! {self(), {Band, Album}},
```

data race!

```
Pid = whereis(critic),
receive
{Pid, Criticism} -> Criticism
after 2000 ->
timeout
end.
```

supervizor

```
start_critic1() ->
spawn(?MODULE, restarter, []).

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.
```

http://learnyousomeerlang.com/errors-and-processes

```
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().
```

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

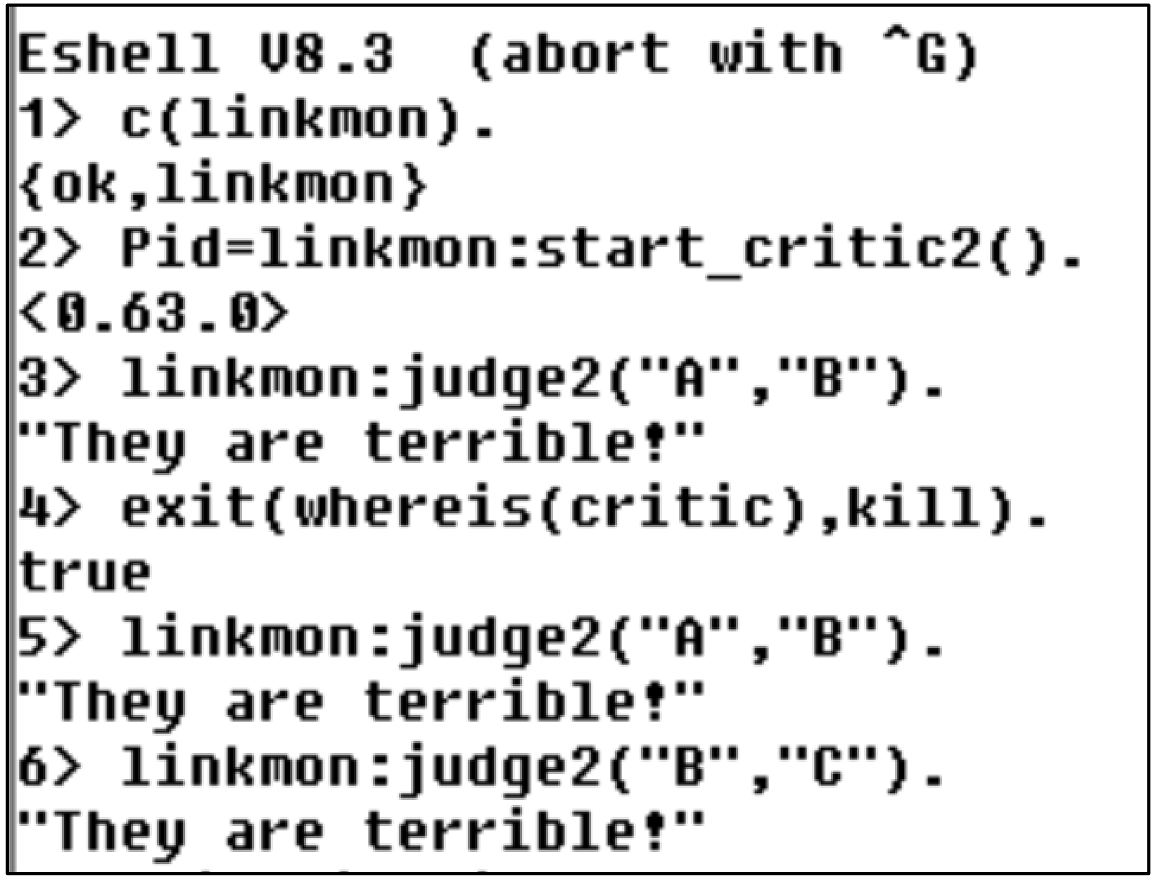
Data type: reference

A reference is a term that is unique in an Erlang runtime system, created by calling **make_ref/0**.
http://erlang.org/doc/reference_manual/data_types.html#id67845

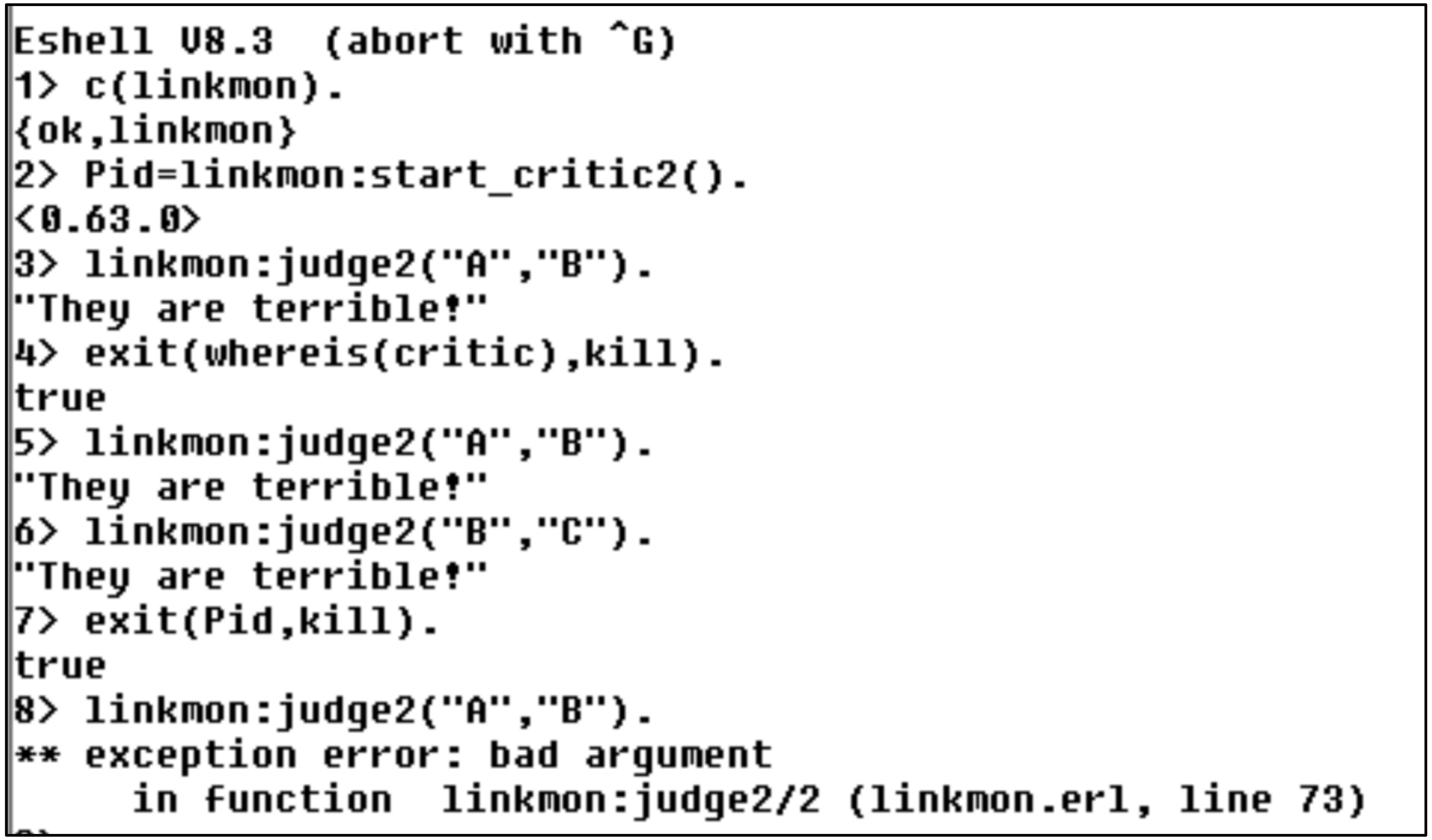
linkmon.erl
http://learnyousomeerlang.com/errors-and-processes

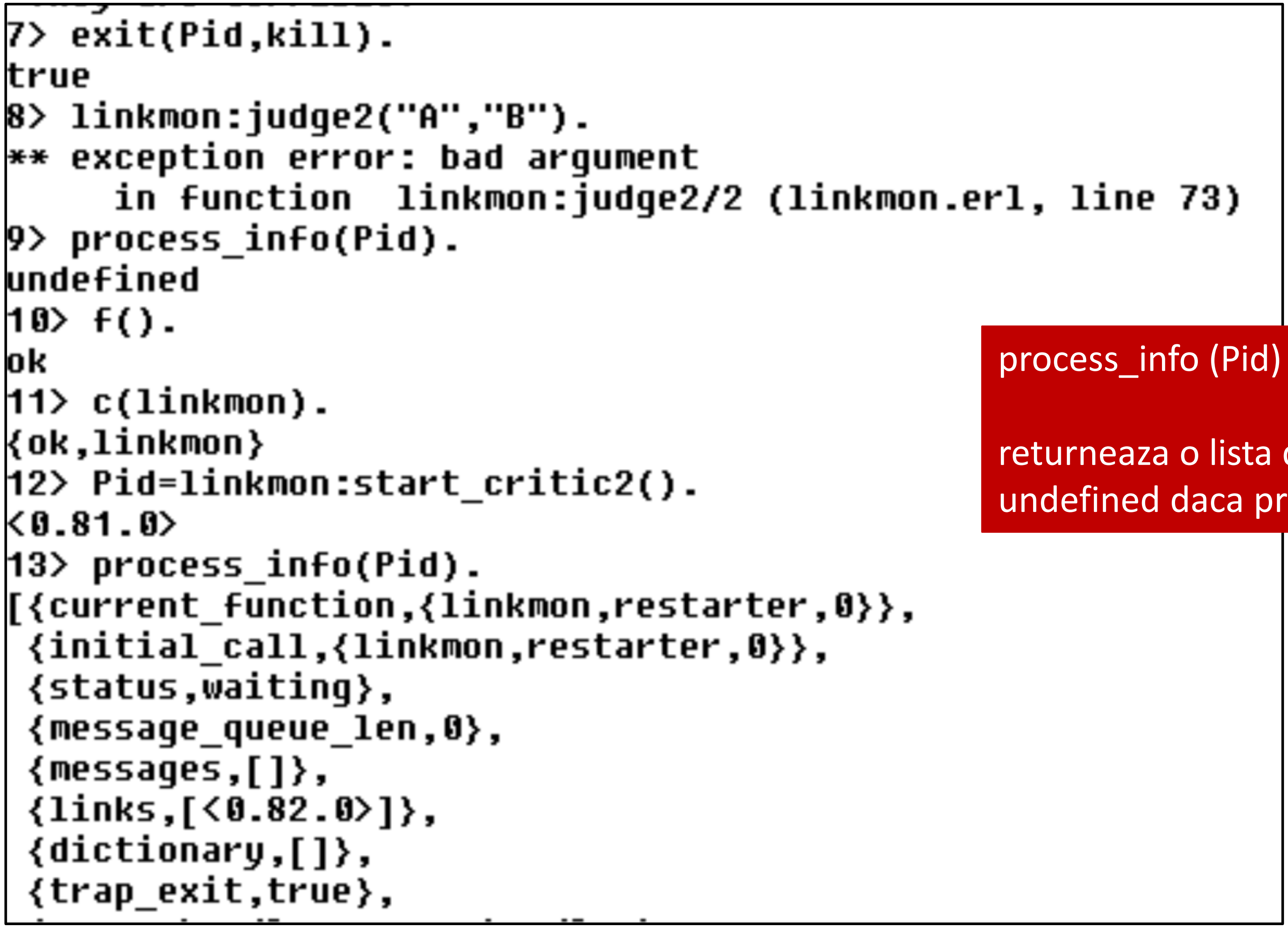
```
start_critic2() ->
spawn(?MODULE, restarter, []).

restarter() ->
process_flag(trap_exit, true),
Pid = spawn_link(?MODULE, critic2, []),
register(critic, Pid),
receive
{'EXIT', Pid, normal} -> % not a crash
ok;
{'EXIT', Pid, shutdown} -> % manual termination, not a crash
ok;
{'EXIT', Pid, _} ->
restarter()
end.
```



```
Eshell V8.3  (abort with ^G)
1> c(linkmon).
{ok,linkmon}
2> Pid=linkmon:start_critic2().
<0.63.0>
3> linkmon:judge2("A","B").
"They are terrible!"
4> exit(whereis(critic),kill).
true
5> linkmon:judge2("A","B").
"They are terrible!"
6> linkmon:judge2("B","C").
"They are terrible!"
```


```
Eshell V8.3  (abort with ^G)
1> c(linkmon).
{ok,linkmon}
2> Pid=linkmon:start_critic2().
<0.63.0>
3> linkmon:judge2("A","B").
"They are terrible!"
4> exit(whereis(critic),kill).
true
5> linkmon:judge2("A","B").
"They are terrible!"
6> linkmon:judge2("B","C").
"They are terrible!"
7> exit(Pid,kill).
true
8> linkmon:judge2("A","B").
** exception error: bad argument
     in function  linkmon:judge2/2 (linkmon.erl, line 73)
```



```
7> exit(Pid,kill).
true
8> linkmon:judge2("A","B").
** exception error: bad argument
     in function  linkmon:judge2/2 (linkmon.erl, line 73)
9> process_info(Pid).
undefined
10> f().
ok
11> c(linkmon).
{ok,linkmon}
12> Pid=linkmon:start_critic2().
<0.81.0>
13> process_info(Pid).
[{current_function,{linkmon,restarter,0}},
 {initial_call,{linkmon,restarter,0}},
 {status,waiting},
 {message_queue_len,0},
 {messages,[]},
 {links,[<0.82.0>]},
 {dictionary,[]},
 {trap_exit,true},
```

process_info (Pid)

returneaza o lista de informatii sau undefined daca procesul nu exista