

# IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

Ioana Leustean

INTRODUCERE IN  
ERLANG



<http://www.erlang.org/>

## ➤ Bibliografie

Joe Armstrong, Programming Erlang, Second Edition 2013

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

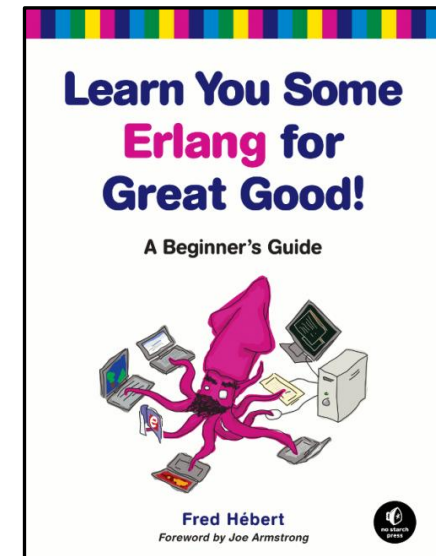
## ACTOR MODEL

"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>



[Varianta online](#)

## ➤ Modelul Actori

- Introdus de Carl Hewitt in 1973
- Actorii sunt o notiune abstracta (corespunzatoare proceselor)
- Actorii au memorie proprie, NU au memorie partajata
- Actorii comunica prin mesaje
- Un actor este capabil sa:
  - trimita mesaje actorilor pe care ii cunoaste
  - creeze noi actori
  - raspunda mesajelor pe care le primeste
- Mesajele contin un destinatar si un continut
- Trimiterea mesajelor este asincrona

➤ Concurenta in Erlang este implementata folosind urmatoarele primitive:

```
Pid = spawn (fun)
```

```
Pid = spawn (module, fct, args)
```

```
Pid ! Message
```

```
receive ... end
```

```
receive ... after ... end
```

## ➤ receive ... after ... end

```
receive  
Pattern1 when Guard1 -> Expr1;  
Pattern2 when Guard2 -> Expr2;  
Pattern3 -> Expr3  
...  
after T ->  
    ExpressionT  
end
```

- La intrarea in **receive**, daca exista un **after**, se porneste un timer.
- Mesajele din coada sunt investigate in ordinea sosirii; daca un mesaj se potriveste cu un pattern atunci expresia corespunzatoare este prelucrata.
- Mesajele care nu se potrivesc cu nici un pattern sunt puse intr-o coada separate (*save queue*).
- Daca nu mai sunt mesaje in coada procesul se suspenda si asteapta venirea unui nou mesaj; la venirea acestuia, numai el este prelucrat, nu si mesajele din *save queue*.
- Cand un mesaj se potriveste cu un pattern, mesajele din *save queue* sunt puse la loc in coada si timerul se sterge.
- Daca timpul T s-a scurs fara ca un mesaj sa se potriveasca unui pattern, atunci ExpressionT se executa, iar mesajele din *save queue* sunt puse inapoi in coada.

- Modelul client-server cu transmiterea starii
- Functia **start(stare)** porneste serverul (proces inregistrat)
  - Comenzile client: **store, take, show, terminate**

```
start(FoodList) -> register(fridge, spawn(fun()-> fridgef(FoodList) end)).  
fridgef(FoodList) ->
```

```
    receive  
    % comanda store  
    % comanda take  
    ....  
    end.
```

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

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

```
show() ->  
    fridge ! {self(), show},  
    receive  
        {fridge, Msg} -> Msg  
    end.
```

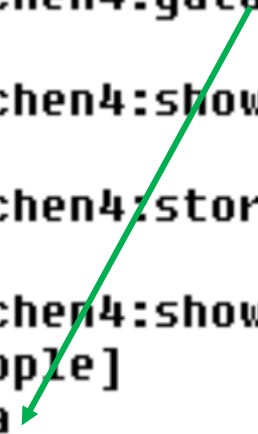
```
terminate() ->  
    fridge ! {self(), terminate},  
    receive  
        {fridge, Msg} -> Msg  
    end.
```

➤ receive ... after ... end

```
fridgef(FoodList) ->
receive
    {From, {store, Food}} -> From ! {fridge, ok},
                                fridgef([Food | FoodList]);
    {From, {take, Food}} ->
        case lists:member(Food, FoodList) of
            true -> From ! {fridge, {ok, Food}},
                                fridgef(lists:delete(Food, FoodList));
            false -> From ! {fridge, not_found},
                                fridgef(FoodList)
        end;
    {From, show} -> From ! {fridge, FoodList},
                                fridgef(FoodList);
    {From, terminate} -> From ! {fridge, done}
after 30000 -> timeout
end,
receive
    gata -> io:format("Sunt gata~n")
end.
```

**gata() -> fridge ! gata**

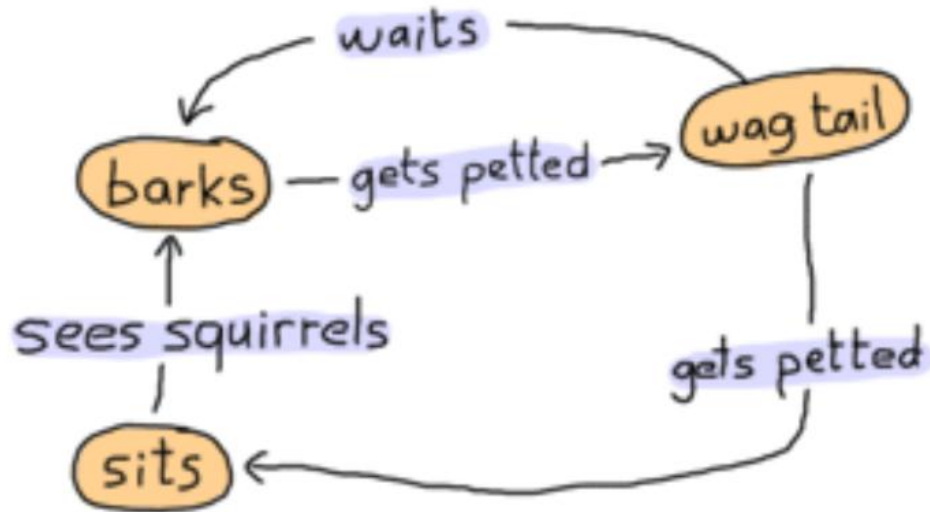
```
1> c(mykitchen4).
{ok, mykitchen4}
2> mykitchen4:start([apple]).
true
3> mykitchen4:gata().
gata
4> mykitchen4:show().
[apple]
5> mykitchen4:store(water).
ok
6> mykitchen4:show().
[water, apple]
Sunt_gata
```



- functia gata() intoarce imediat, shell-ul nu ramane blocat
- se pot trimite mesaje serverului
- mesajul **gata** este prelucrat dupa ce au trecut 30 sec fara o comanda prelucrata de primul receive



## ➤ Implementarea unui automat finit (Finite-State Machine)



Starile = {barks, sits, wag\_tail}

Actiunile = {gets\_petted, see\_squirrels, waits}

dog as a state-machine

<http://learnyoussomeerlang.com/finite-state-machines#what-are-they>

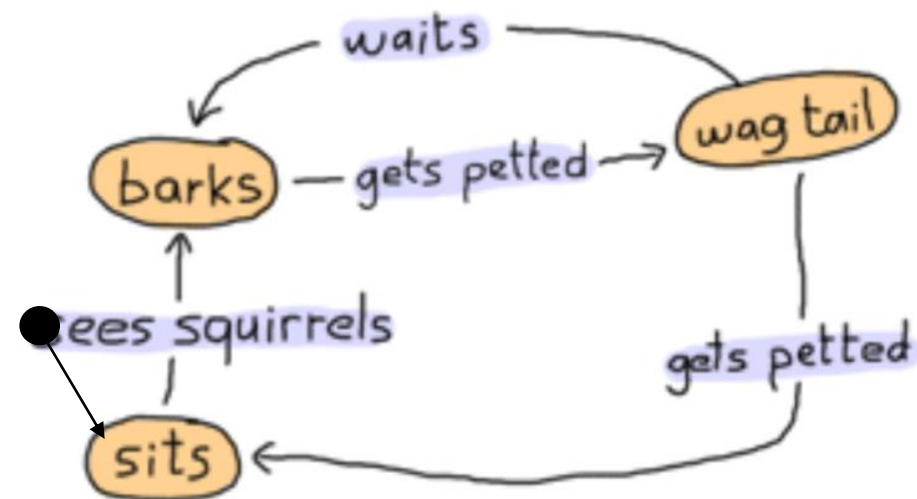
## ➤ Implementarea unui automat finit

```
-module(dog_fsm).  
-export([start/0, squirrel/1, pet/1]).
```

```
start() ->  
    spawn(fun() -> bark() end).    % starea initiala
```

```
%actiunea see_squirrels  
squirrel(Pid) -> Pid ! squirrel.
```

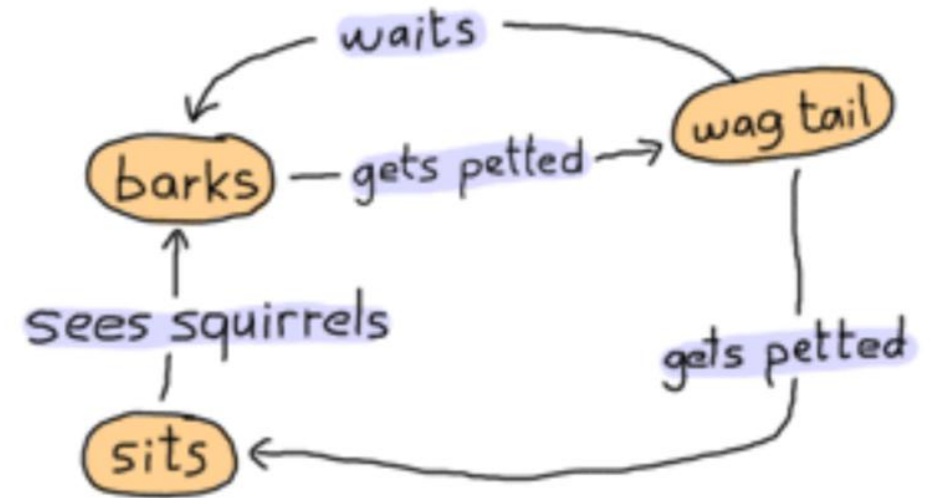
```
%actiunea gets_pettet  
pet(Pid) -> Pid ! pet.
```



actiunile sunt implementate prin mesaje si  
sunt vizibile in exterior

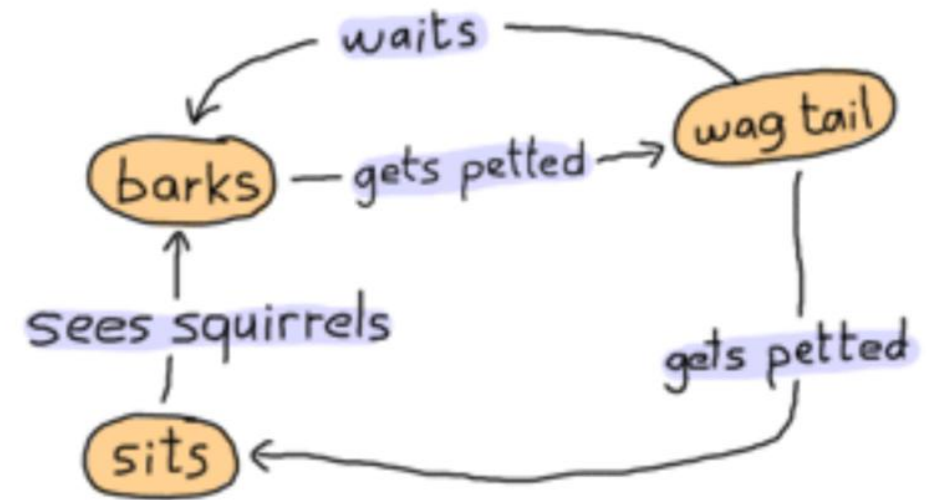
## ➤ Finite-StateMachine: implementarea starilor

```
bark() ->  
  io:format("Dog says: BARK! BARK!~n"),  
  receive  
    pet -> wag_tail();  
  
    _ -> io:format("Dog is confused~n"),  
         bark()  
  
  after 2000 -> bark()  
end.
```



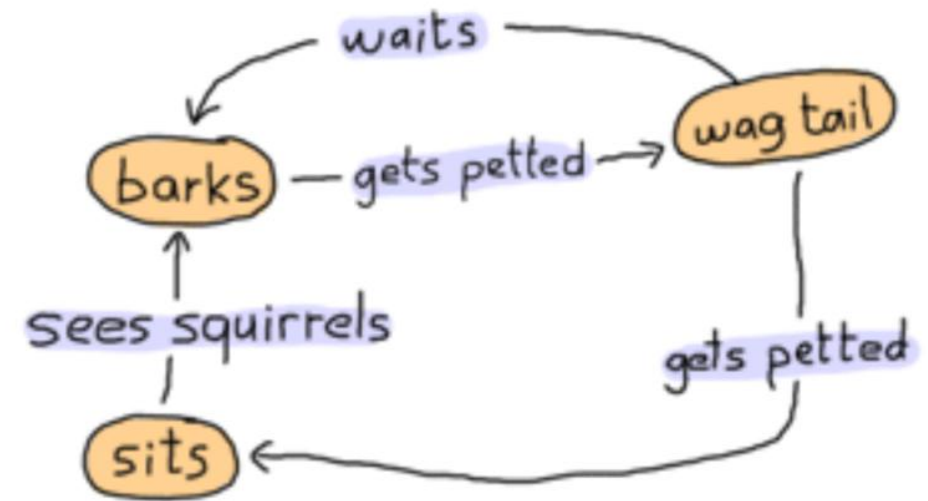
## ➤ Automat finit: definirea starilor

```
wag_tail() ->  
  io:format("Dog wags its tail~n"),  
  receive  
    pet -> sit();  
  
    _ -> io:format("Dog is confused~n"),  
         wag_tail()  
  
  after 30000 ->  
    bark()    % actiunea waits  
  
end.
```



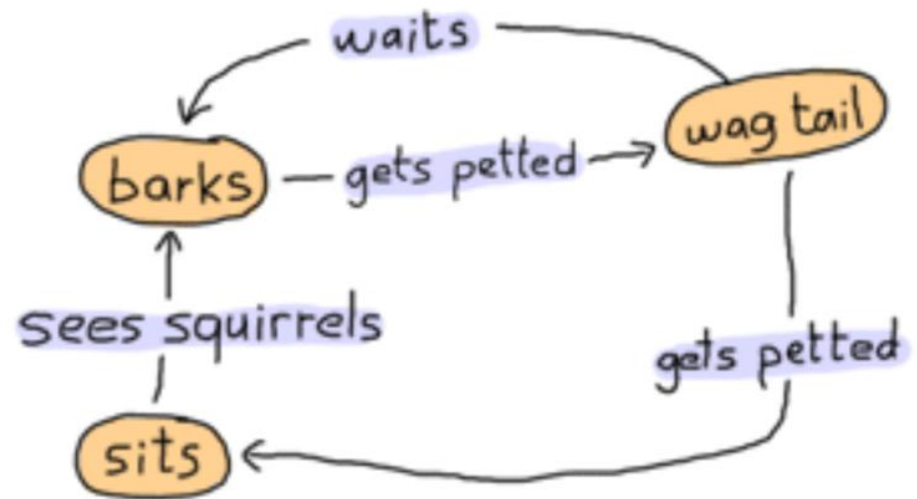
## ➤ Automat finit: definirea starilor

```
sit() ->  
  io:format("Dog is sitting. Gooooood boy!~n"),  
  receive  
    squirrel -> bark();  
    _ -> io:format("Dog is confused~n"),  
         sit()  
  end.
```



## ➤ Implementarea unui automat finit

```
1> c(dog_fsm).  
{ok,dog_fsm}  
2> Pid=dog_fsm:start().  
Dog says: BARK! BARK!  
<0.63.0>  
Dog says: BARK! BARK!  
Dog says: BARK! BARK!  
Dog says: BARK! BARK!  
3> dog_fsm:pet(Pid).  
Dog wags its tail  
pet  
4> dog_fsm:pet(Pid).  
Dog is sitting. Gooooood boy!  
pet  
5> dog_fsm:squirrel(Pid).  
Dog says: BARK! BARK!  
squirrel  
Dog says: BARK! BARK!  
Dog says: BARK! BARK!  
Dog says: BARK! BARK!  
Dog says: BARK! BARK!
```



<http://learnyousomeerlang.com/finite-state-machines#what-are-they>

## ➤ Tratarea erorilor

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

## ➤ OTP

OTP stands for Open Telecom Platform, although it's not that much about telecom anymore (it's more about software that has the property of telecom applications, but yeah.) If half of Erlang's greatness comes from its concurrency and distribution and the other half comes from its error handling capabilities, then the OTP framework is the third half of it.

<http://learnyousomeerlang.com/what-is-otp#its-the-open-telecom-platform>

## ➤ 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."

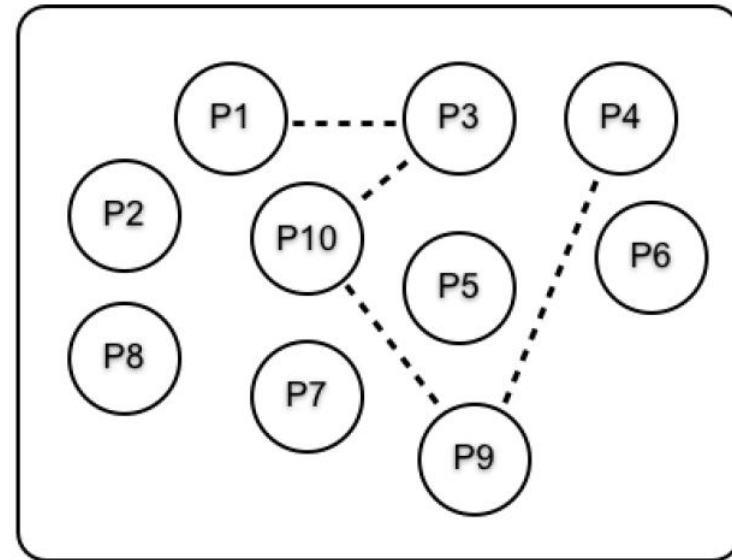
Joe Armstrong, Programming Erlang, Second Edition 2013

## ➤ Erori in programarea concurenta

- procese 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 process se termina, el emite un *exit reason*
  - daca un proces se termina normal, *reason* este atomul **normal**
  - daca un process se termina cu eroare (la runtime), *reason* este {Reason, Stack}
  - un proces se poate termina singur prin apelul functiei *exit*

- cand un proces se termina (normal sau cu eroarea) trimite automat un semnal tuturor proceselor de care este legat

## ➤ Mesaje si semnale de eroare

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

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.

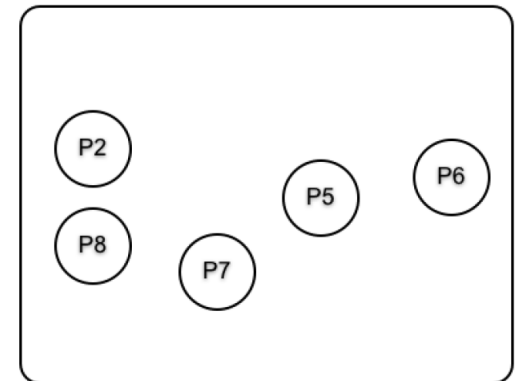
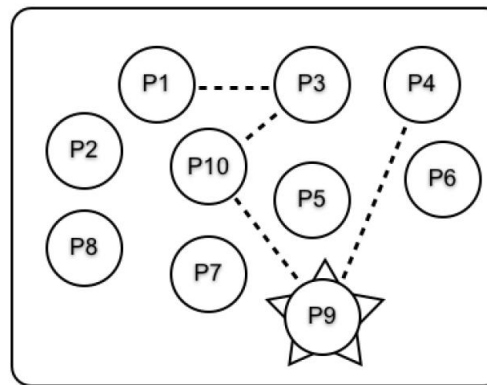
## ➤ Procese normale si procese sistem

- exista doua tipuri de procese: procese normale si procese sistem
- 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

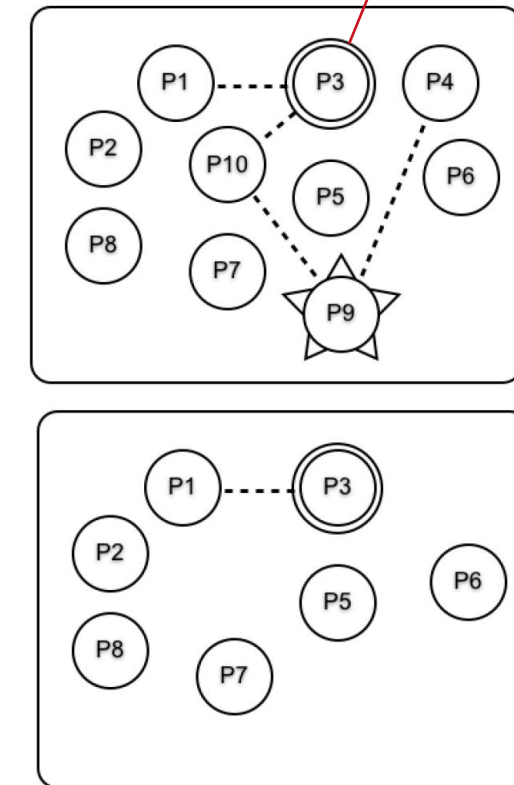
## ➤ Primirea semnalelor de eroare

- 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.

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



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

Joe Armstrong, Programming Erlang,  
Second Edition 2013

## ➤ link() si spawn\_link()

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

procesul s-a terminat  
inainte de a face legatura

### 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.

```
2> Pid = spawn(linkmon, myproc, []).
<0.85.0>
3> link(Pid).
* 1: syntax error before: ')'
3> link(Pid).
* exception error: no such process or port
  in function link/1
    called as link(<0.85.0>)
4> f().
ok
5> Pid = spawn(linkmon, myproc, []).
<0.91.0>
6> link(Pid).
true
7> 7> 7> ** exception error: reason
7> █
```

Joe Armstrong, Programming Erlang, Second Edition 2013

## ➤ link() si spawn\_link()

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

```
Eshell V11.0 (abort with ^G)
1> c(linkmon).
linkmon.erl:4: Warning: export_all f
{ok,linkmon}
2> Pid=spawn(linkmon, myproc, []).
<0.85.0>
3> link(Pid).
true
4> 4> 4> ** exception error: reason
4> spawn_link(linkmon, myproc, []).
<0.89.0>
5> 5> 5> ** exception error: reason
5> █
```

### 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

```

Erlang/OTP 23 [erts-11.0] [64-bit] [smp:16:16]

Eshell V11.0 (abort with ^G)
1> c(linkmon).
linkmon.erl:4: Warning: export_all flag enabled

{ok,linkmon}
2> self().
<0.79.0>
3> spawn_link(linkmon,chain, [3]).
<0.87.0>
** exception error: "chain dies here"
4> self().
<0.92.0>
5> process_flag(trap_exit,true).
false
6> process_flag(trap_exit,true).
true
7> spawn_link(linkmon,chain, [3]).
<0.96.0>
8> flush().
Shell got {'EXIT',<0.96.0>,"chain dies here"}
ok
9> self().
<0.92.0>

```

procesul evaluator initial

exceptie

proces evaluator nou

- pentru efectuarea comenzilor, shell-ul foloseste un proces evaluator
- la aparitia unei exceptii, procesul evaluator curent se termina, iar shell-ul creaza un nou process evaluator

[Erlang -- shell](http://www.erlang.org/docs)

## ➤ 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 terrible!"};
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>



<http://www.erlang.org/docs>

```
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://learnyoussomeerlang.com/errors-and-processes>



<http://www.erlang.org/docs>

## ➤ 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.
```

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

```
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.
```

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](http://erlang.org/doc/reference_manual/data_types.html#id67845)

linkmon.erl

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



<http://www.erlang.org/docs>

```
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