IMPLEMENTAREA CONCURENTEI IN LIMBAJE DE PROGRAMARE

Ioana Leustean

INTRODUCERE IN ERLANG

http://www.erlang.org/

PARALELISM

CONCURENTA

SISTEME DISTRIBUITE

"Erlang was designed from the bottom up to program concurrent, distributed, fault-tolerant, scalable, soft, real-time systems. […]

If your problem is concurrent, if you are building a multiuser system, or if you are building a system that evolves with time, then using Erlang might save you a lot of work, since Erlang was explicitly designed for building such systems. […]

Processes interact by one method, and one method only, by exchanging messages. Processes share no data with other processes. This is the reason why we can easily distribute Erlang programs over multicores or networks. "

Joe Armstrong, Programming Erlang, Second Edition 2013

➢Bibliografie

Joe Armstrong, Robert Virding, Mike Williams, Concurrent Programming in Erlang, 1993

Joe Armstrong, Programming Erlang, Second Edition 2013

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

https://www.erlang.org/doc/

CONCURRENCY IN ERLANG

lightweight processes with asynchronous message passing

Procesele in Erlang:

- pot fi create si distruse rapid
- comunica prin mesaje, iar comunicarea este rapida
- sunt complet independente din punctul de vedere al memoriei

➢ Crearea proceselor: spawn

Functia spawn creaza un process care este executat in parallel cu procesul care l-a creat si intoarce un **Pid** (Process Identifier) , care este folosit pentru trimiterea mesajelor.

spawn/3
spawn(modul, functie, lista argumentelor)
Pid = spawn(modul, functie, lista argumentelor)

```
31> c(myconc).
{ok,myconc}
32> spawn(myconc,prelA,[5]).
A
<0.123.0>
A
A
A
A
End A
```

Pid= spawn(myconc,prelA,[5]).

```
-module(myconc).
-export([prelA/1).

prelA(X) when (X == 0) -> io:format("End A ~n");
prelA(X) when (X > 0) -> io:format("A ~n"), prelA(X-1);
prelA(_) ->      io:format("error ~n").
```

➢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

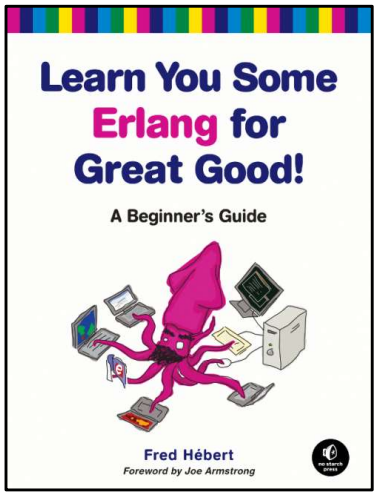
ACTOR MODEL

"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

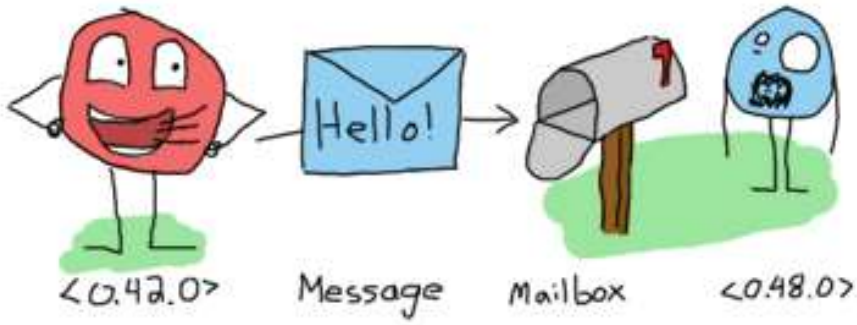
http://learnyousomeerlang.com/introduction#what-is-erlang



Varianta online

- Transmiterea mesajelor este asincrona.

Datorita cozii pentru mesaje,
procesul care transmite mesajul
nu asteapta o confirmare de primire sau
prelucrarea acestuia,
mesajul intra in coada si asteapta
pana cand va fi procesat



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

"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
```

https://www.erlang.org/doc/man/erlang.html#spawn-4

➢ Cilent-Server (Exemplu simplu: doubling service)

```
3> c(myserv).
{ok,myserv}
4> Ser=spawn(myserv, server_loop, []).
<0.44.0>
5> Ser ! {self(),{double,5}}.
{<0.32.0>,{double,5}}
6> flush().
Shell got {<0.44.0>,10}
ok
7> Ser ! {self(),{double,7}}.
{<0.32.0>,{double,7}}
8> flush().
Shell got {<0.44.0>,14}
ok
9> Ser ! {self(),111}.
{<0.32.0>,111}
10> flush().
Shell got {<0.44.0>,error}
ok
```

- Procesul **Ser** este serverul si executa functia **server_loop**
- Serverul primeste mesaje de la procese client si executa o actiune (dubleaza valoarea primita)
- In acest exemplu singurul client este shell-ul
- Mesajele primite de shell, adica raspunsurile trimise de server, sunt vizualizate cu **flush()**

➢ Cilent-Server (Exemplu simplu: doubling service)

```
-module(myserv).
-export([server_loop/0]).

server_loop() ->
  receive
      {From, {double, Number}} -> From ! {self(),Number*2},
                                  server_loop() ;

       {From,_} -> From ! {self(),error},
                   server_loop()
  end.
```

```
3> c(myserv).
{ok,myserv}
4> Ser=spawn(myserv, server_loop, []).
<0.44.0>
5> Ser ! {self(),{double,5}}.
{<0.32.0>,{double,5}}
6> flush().
Shell got {<0.44.0>,10}
ok
7> Ser ! {self(),{double,7}}.
{<0.32.0>,{double,7}}
8> flush().
Shell got {<0.44.0>,14}
ok
9> Ser ! {self(),111}.
{<0.32.0>,111}
10> flush().
Shell got {<0.44.0>,error}
ok
```

➢ Cilent-Server : functie pentru pornirea server-ului

```
-module(myserv).
-export([server_loop/0]).

server_loop() ->
  receive
       {From, {double, Number}} -> From ! {self(), Number*2},
                                   server_loop() ;

        {From,_} -> From ! {self(),error},
                    server_loop()
  end.
```

```
3> c(myserv).
{ok,myserv}
4> Ser=spawn(myserv, server_loop, []).
<0.44.0>
5> Ser ! {self(),{double,5}}.
{<0.32.0>,{double,5}}
6> flush().
Shell got {<0.44.0>,10}
ok
7> Ser ! {self(),{double,7}}.
{<0.32.0>,{double,7}}
8> flush().
Shell got {<0.44.0>,14}
ok
9> Ser ! {self(),111}.
{<0.32.0>,111}
10> flush().
Shell got {<0.44.0>,error}
ok
```

```
-export([start_server/0, server_loop/0]).
start_server() ->  spawn(myserv, server_loop, []).
```

```
16> Ser=myserv:start_server().
<0.66.0>
17> Ser ! {self(), {double,45}}.
{<0.59.0>,{double,45}}
18> flush().
Shell got {<0.66.0>,90}
ok
```

➢ Cilent-Server: functia client

```
-module(myserv).
-export([start_server/0, server_loop/0,client/2]).


start_server() ->  spawn(myserv, server_loop, []).


server_loop() ->
  receive
     {From, {double, Number}} -> From ! {self(),(Number*2)},
                                 server_loop() ;

     {From,_} -> From ! {self(),error},
                 server_loop()
  end.
```

```
client(Pid, Request) ->
      Pid ! {self(), Request},
      receive
            {Pid, Response} -> Response
      end.
```

functia **client intoarce raspunsul primit de la server**

apelarea functiei client

```
3> c(myserv).
{ok,myserv}
4> Server = myserv:start_server().
<0.43.0>
5> myserv:client(Server,{double,15675}).
31350
6> myserv:client(Server,nothing).
error
7> myserv:client(Server, {double, 887}).
1774
```

- Client-Server

  client_loop creaza mai multe procese client si intoarce lista rezultatelor

```
client_loop(Pid,0,L)  -> Pid! {self(),"Good Bye"},
                         L;

client_loop(Pid, X, L)  ->  R= client(Pid,{double,X}),
                            client_loop(Pid, X-1, L++[R]).
```

```
31> c(myserv2).
{ok,myserv2}
32> Ser = myserv2:start_server().
<0.113.0>
33> myserv2:client_loop(Ser,10,[]).
[20,18,16,14,12,10,8,6,4,2]
34> flush().
Shell got {<0.113.0>,"Good Bye"}
ok
```

- Client-Server

  client_loop creaza mai multi clienti si intoarce lista rezultatelor

```
client_loop(Pid,0,L)  -> Pid! {self(),"Good Bye"},
                         L;

client_loop(Pid, X, L)  ->  R= client(Pid,{double,X}),
                            client_loop(Pid, X-1, L++[R]).
```

Functiile client sunt executate **secvential!**

➢Client-Server

procesele client se executa in **paralel** si se intoarce lista rezultatelor

```
worker(Parent, Pid, Number) ->   spawn( fun() ->
                                        Result = client (Pid,{double,Number}),
                                        Parent ! {self(),Result}
                                        end ).
calls (Pid,N) ->  Parent = self(),
                  Pids = [worker(Parent,Pid, X)  || X <- lists:seq(1,N)],
                        [ wait_one(P)  ||  P <- Pids ].

wait_one (Pid) ->
                receive
                    {Pid,Response} -> Response
                end.
```

% **Pid** este id-ul procesului server
% **Parent** este id-ul procesului care creaza clientii
% **worker** creaza un proces client si intoarce id-ul acestuia

- Client-Server

```
start_server() ->  spawn(myserv, server_loop, []).
start_seq_clients(Pid, N) -> client_loop(Pid,N,[]).
start_par_clients(Pid, N) -> calls(Pid,N).
```

```
62> c(myserv2).
{ok,myserv2}
63> Server = myserv2:start_server().
<0.15705.27>
64> myserv2:start_par_clients(Server, 100000).
[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,
 44,46,48,50,52,54,56,58|...]
65> myserv2:start_seq_clients(Server, 100000).
```

➢Client-Server

unui proces i se poate asocia un nume (atom) folosind register

myserv3.erl

```
start_server() ->  register(serv, spawn(fun() ->server_loop() end)).

server_loop() ->
  receive
    {From, {double, Number}} -> From ! {serv,(Number*2)},
                                server_loop() ;
      {From, "Good Bye"} -> From! {serv,"Good Bye"},
                            server_loop();

    {From,_} -> From ! {serv,error},
                server_loop()
  end.
```

procesul server are numele **serv**

- Registered process

register (Name, Pid) asociaza numele **Name** procesului cu id-ul **Pid**
Name este atom si este "eliberat" cand procesul se termina

whereis(Name) intoarce id-ul procesului inregistrat cu **Name**
(sau **undefined** daca nu exista)

Processes — Erlang System Documentation v27.2

```
start_server() ->  register(serv, spawn(fun() ->server_loop() end)).
```

➢ Client-Server

```
start_par_clients(N) -> calls(N).
worker(Parent,  Number) ->
   spawn( fun() ->
                Result = client ({double,Number}),
                 Parent ! {self(),Result}
              end ).

calls (N) ->
      Parent = self(),
      Pids = [worker(Parent,X)|| X <- lists:seq(1,N)],
      [waitone(P)|| P <- Pids].



waitone (Pid) ->
              receive
                 {Pid,Response} -> Response
               end.
```

```
client(Request) ->
        serv ! {self(), Request},
        receive
              {serv, Response} -> Response
        end.
```

```
1> cd ("D:/DIR/ER/myer").
D:/DIR/ER/myer
ok
2> c(myserv3).
{ok,myserv3}
3> myserv3:start_server().
true
4> myserv3:start_par_clients(50).
[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,
 44,46,48,50,52,54,56,58|...]
5>
```

Distributed Erlang: programele ruleaza in noduri diferite

```
PS C:\Users\igleu> cd C:\Users\igleu\Documents\DIR\ICLP\ICLP2024\c11-12-2024-Erlang\ER\myer
PS C:\Users\igleu\Documents\DIR\ICLP\ICLP2024\c11-12-2024-Erlang\ER\myer> erl -sname ioserver
Eshell V11.0  (abort with ^G)
(ioserver@LAPTOP-KCKGLLT6)1> myserv3:start_server().
true
```

Command Prompt - erl -snan

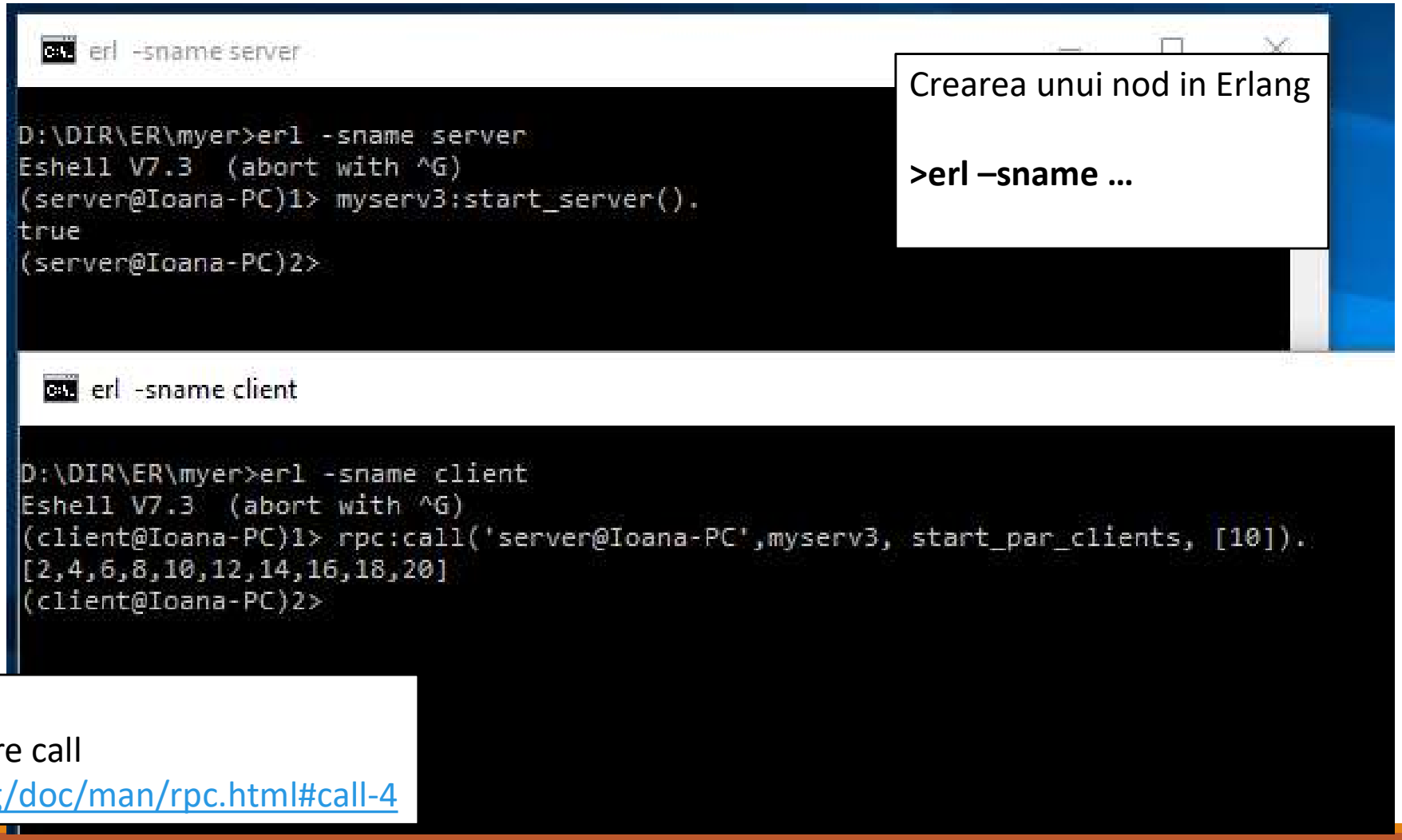
```
Microsoft Windows [Version 10.0.22631.3593]
(c) Microsoft Corporation. All rights reserved.

C:\Users\igleu>erl -sname ioclient
Eshell V11.0  (abort with ^G)
(ioclient@LAPTOP-KCKGLLT6)1> {serv,'ioserver@LAPTOP-KCKGLLT6'}!{self(),{double,5}}.
{<0.84.0>,{double,5}}
(ioclient@LAPTOP-KCKGLLT6)2> flush().
Shell got {serv,10}
ok
(ioclient@LAPTOP-KCKGLLT6)3>
```

Crearea unui nod in Erlang

>erl –sname …

Numele e atom!

Distributed Erlang: programele ruleaza in noduri diferite



rpc:call
remote procedure call
http://erlang.org/doc/man/rpc.html#call-4

Distributed Erlang: programele ruleaza in noduri diferite

```
C:\Users\igleu>cd C:\Users\igleu\Documents\DIR\ICLP\ICLP2024\c11-12-2024-Erlang\ER\myer

C:\Users\igleu\Documents\DIR\ICLP\ICLP2024\c11-12-2024-Erlang\ER\myer>erl -sname ioserv
Eshell V11.0  (abort with ^G)
(ioserv@LAPTOP-KCKGLLT6)1>
```

Windows PowerShell

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvem

PS C:\Users\igleu> erl -sname new
Eshell V11.0  (abort with ^G)
(new@LAPTOP-KCKGLLT6)1>  spawn('ioserv@LAPTOP-KCKGLLT6', myserv3, start_server, []).
<8381.89.0>
(new@LAPTOP-KCKGLLT6)2>
```

Windows PowerShell

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\igleu> erl -sname ioclient
Eshell V11.0  (abort with ^G)
(ioclient@LAPTOP-KCKGLLT6)1> {serv,'ioserv@LAPTOP-KCKGLLT6'}!{self(),{double,5}}.
{<0.84.0>,{double,5}}
(ioclient@LAPTOP-KCKGLLT6)2> flush().
Shell got {serv,10}
```

cu spawn/4 serverul este pornit din alt nod
https://www.erlang.org/doc/apps/erts/erlang#spawn/4

➢ ?MODULE

macro (definit de sistem) care intoarce numele modulului curent

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

myrec() ->
  receive
   {do_A, X} ->  prelA(X);
   {do_B, X} ->  prelB(X);
              _ -> io:format("Nothing to do ~n")
  end.
```

```
3> Pid = myconc1:start().
<0.112.0>
4> Pid ! {do_A,2}.
A
{do_A,2}
A
End A
```

➢ Cilent-Server (simple) template

```
-module(servtemplate1).
-compile(export_all).  %exporta toate functiile

start_server() ->  spawn(?MODULE, server_loop, []).

 client(Pid, Request) ->
        Pid ! {self(), Request},
        receive
              {Pid, Response} -> Response
         end.

server_loop() ->
   receive
        .....
        {From, Request} -> From ! {self(),Response} ,
                                   server_loop()

        end.
```

```
-module(servtemplate2).
-compile(export_all).

start_server() ->
  register(serv,spawn(?MODULE, server_loop, [])).

 client(Request) ->
        serv ! {self(), Request},
        receive
              {serv, Response} -> Response
         end.

server_loop() ->
   receive
        .....
        {From, Request} -> From ! {serv, Response} ,
                                   server_loop()

        end.
```

➢ Schimb de mesaje cu transmiterea starii (message passing with data storage)

- Procesul (serverul) este un frigider care accepta doua tipuri de comenzi
  - depoziteaza alimente,
  - scoate alimente .

- Acelasi aliment poate fi depozitat de mai multe ori si poate fi scos de cate ori a fost depozitat.

- La fiecare moment trebuie sa stim ce alimente se gasesc in frigider (starea procesului).

- Starea procesului se transmite prin parametrii functiilor.

kitchen.erl
http://learnyousomeerlang.com/

- Schimb de mesaje cu transmiterea starii

```
fridgef(FoodList) ->
                receive
                % comanda store
                % comanda take
                ....
                end.
```

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

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

kitchen.erl

http://learnyousomeerlang.com/

➢ Mesaje cu transmiterea starii

kitchen.erl
http://learnyousomeerlang.com/

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

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

```
6> c(kitchen).
{ok,kitchen}
7> Fridge = kitchen:start([milk, cheese, ham]).
<0.99.0>
8> kitchen:store(Fridge, juice).
ok
9> kitchen:take(Fridge, milk).
{ok,milk}
10> kitchen:take(Fridge, juice).
{ok,juice}
11> kitchen:take(Fridge, juice).
not_found
```

- Varianta: registered process, comenzile **show** (pentru a vizualiza starea) si **terminate**

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

- Varianta: registered process, comenzile **show** (pentru a vizualiza starea) si **terminate**

```
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}
    end.
```

mykitchen.erl

```
2> c(mykitchen).
{ok,mykitchen}
3> mykitchen:start([milk, apple]).
true
4> mykitchen:take(milk).
{ok,milk}
5> mykitchen:store(orange).
ok
6> mykitchen:show().
[orange,apple]
7> mykitchen:terminate().
done
```

➢ receive … after … end

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

- procesul asteapta pana cand primeste un mesaj care se potriveste cu un pattern sau pana cand expira timpul .
- timpul T este exprimat in milisecunde
- procesul va astepta maxim T milisecunde sa primeasca un mesaj
- daca nici un mesaj care se potriveste cu un patern nu este primit in timpul T, procesul executa ExpressionT

➢receive … after … end

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

- nu exista sabloane, deci niciun mesaj nu va fi acceptat;
- procesul va fi blocat T milisecunde

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

- orice mesaj se potriveste cu patternul, deci apelul recursiv va goli coada de mesaje, dupa care procesul va continua
- instructiunea `after 0 -> …`

  verifica coada de mesaje si apoi continua;
  daca aceasta clauza lipseste, procesul se va bloca cand coada de mesaje se goleste

➢ 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.

mykitchen3.erl

```
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)
 end,
 io:format("al doilea receive~n"),
 receive
      {Pid, terminate} -> Pid ! {fridge, done}
  end.
```

```
1> c(mykitchen3).
{ok,mykitchen3}
2> mykitchen3:start([]).
true
3> mykitchen3:store(apple).
ok
4> mykitchen3:terminate().
```

procesul este **blocat pentru ca nu poate iesi din primul receive**

➢ 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)

 after 30000 ->   timeout
 end,
  io:format("al doilea receive~n"),
  receive
      {From,terminate} -> From ! {fridge, done}
   end.
```

```
1> c(mykitchen3).
{ok,mykitchen3}
2> mykitchen3:start([]).
true
3> mykitchen3:terminate().
al doilea receive
done
```

dupa 30 sec

- in apelul **terminate()**, procesul shell asteapta mesaj de la fridge pentru **receive** din **terminate()**

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

- apelul functiei **terminate()** se incheie numai dupa ce trec cele 30 sec si poate fi prelucrat mesajul "terminate" in al doilea **receive** din **fridgef()**

- 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 trec 30 sec fara o comanda prelucrata de primul receive

- Selective receives

```
5> self()! hi, self() ! low.
low
6> flush().
Shell got hi
Shell got low
ok
```

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

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

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

Varianta a functiei flush() care ordoneaza mesajele dupa prioritati

```
2> c(sel).
{ok,sel}
3> self()! {5, low1}, self() ! {9,low2}, self() ! {15, high1}, self()!{11,high2}
.
{11,high2}
4> sel:important().
[high1,high2,low1,low2]
```

More On Multiprocessing | Learn You Some Erlang for Great Good!

- 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.”

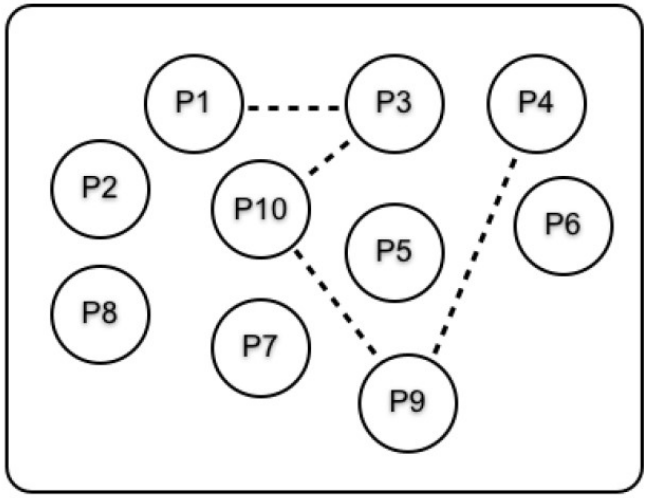
Joe Armstrong, Programming Erlang, Second Edition 2013

➢ 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 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.

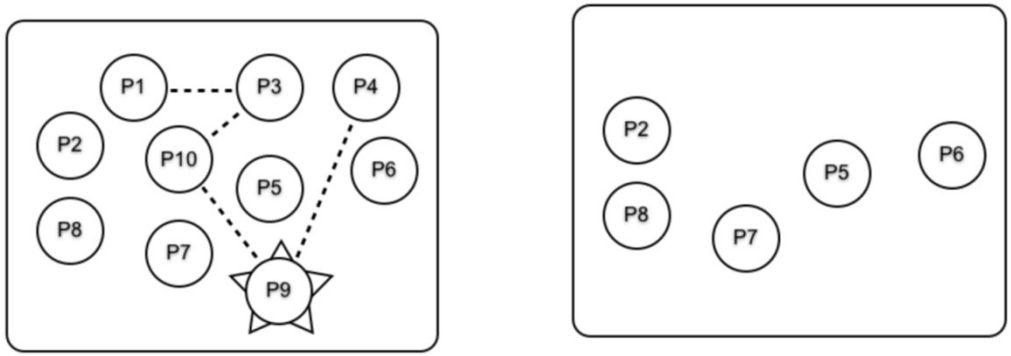
➢Procese normale si procese sistem

- exista doua tipuri de procese: procese normale si procese 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

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

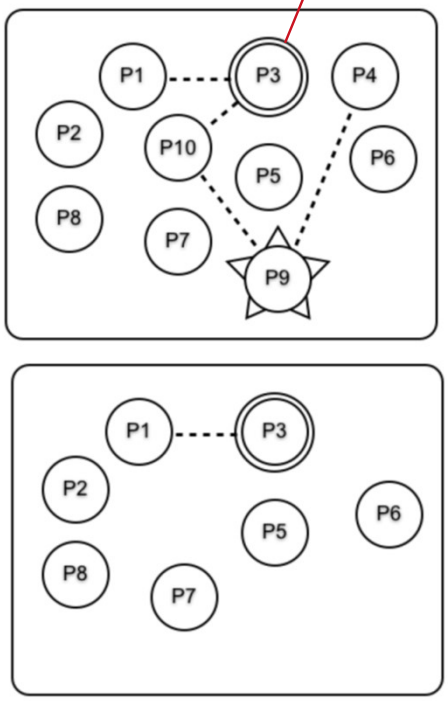
  {'EXIT', Pid, Why}

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 system pimeste 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

```
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>
```

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

➢ 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

- 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

server

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

client

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

supervizor

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

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

Serverul este repornit

```
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