Erlang

Control dels errors

Característiques

- **Concurrència:**
  - Utilitza processos lleugers amb uns requisits de memòria que poden variar de forma dinàmica.
  - Els processos no tenen memòria compartida i es comuniquen per pas de missatges asíncrons.
- **Distribució:**
  - Està dissenyat per executar-se en un entorn distribuït.
  - Una màquina virtual Erlang rep el nom de node Erlang.
  - Un sistema distribuït d'Erlang és una xarxa de nodes Erlang.
  - Un node Erlang pot crear processos paral·lels que s'executen en altres nodes, que potser utilitzen altres sistemes operatius.
  - Els processos que resideixen en els diferents nodes es comuniquen mitjançant el pas de missatges.

Característiques

- **Robustesa:**
  - Té diverses primitives de detecció d'errors que es poden utilitzar per estructurar sistemes tolerants a fallades.
  - Per exemple, els processos poden monitoritzar l'estat i les activitats d'altres processos, encara que els processos s'executen en altres nodes.
  - Els processos en un sistema distribuït es poden configurar per commutar, en cas d'error, o amb altres nodes en cas de fallades, i així ser recuperats amb facilitat.
- **Soft-real-time:**
  - Admet programació "suau" de sistemes de temps real, que requereixen temps de resposta de l'ordre de mil·lisegons.

Característiques

- **Actualització de codi en calent:**
  - Molts sistemes no es poden aturar per fer el manteniment.
  - Erlang té codi que permet canviar parts de l'aplicació en un sistema en funcionament.
  - El codi antic es pot eliminat i substituir pel nou codi.
  - Durant la transició, el codi antic i el nou codi poden coexistir.
  - Per tant, és possible instal·lar pegats i actualitzacions en un sistema en funcionament sense alterar el seu funcionament.
- **Codi de càrrega incremental:**
  - Els usuaris poden controlar en detall com es carrega el codi.
  - En els sistemes encastats, tot el codi es carrega generalment a l'arrencada.
  - En els sistemes de desenvolupament, el codi es carrega quan és necessari, fins i tot quan el sistema està en funcionament.
  - Si l'anàlisi revela errors, només s'ha de substituir el codi erroni.
- **Interfícies externes**:
  - Els processos d'Erlang es comuniquen amb eines externes amb el mateix mecanisme de pas de missatges que s'utilitza entre els processos Erlang.
  - Aquest mecanisme s'utilitza per a la comunicació amb el sistema operatiu i la interacció amb altres programes.

OTP: *Open Telecom Platform*

- OTP és una extensa col·lecció de llibreries que s'han estandarditzat dins el món Erlang. Molts projectes que usen "Erlang" realment fan servir "Open Source Erlang/OTP", és a dir, l'entorn Erlang de lliure distribució i les llibreries.
- OTP és un codi obert, i es distribueix juntament amb Open Source Erlang. Algunes llibreries incloses en OTP són:

    - **mnesia** és un Sistema de gestió de Bases de Dades distribuït,
    - **mnemosyne** és una interfície per a realitzar consultes a una Bases de Dades distribuïda.
    - **odbc** proporciona una interfície per accedir a bases de dades mitjançant connexió ODBC.
    - **snmp** és uns agent de Simple Network Management Protocol Extensible
    - **eva** és una adaptació a Event and Alarm Handling.
    - **orber** i altres són implementacions per Erlang de diferents parts de CORBA.
    - **asn1** inclou mòduls amb suport de ASN.1 en temps de compilació i d'execució.
    - **crypto** i **ssl** proporciona funcions per computar resums de missatges, xifrat i desxifrat. Utilitza parts d'OpenSSL.
    - **gs** (Graphics System) és una llibreria de rutines per escriure interfícies d'usuari gràfiques. Els programes que fan servir GS funcionen en qualsevol plataforma Erlang sense importar-hi el sistema del sistema.
    - **inets** implementa un servidor HTTP/1.1 i un client FTP.
    - **jinterface** és un paquet per comunicació amb Java.
    - **megaco** és una infraestructura per construir aplicacions basades en el protocol Megaco/H.248.
    - **debugger** eina gràfica per depuració.
    - **tv** permet examinar gràficament taules MTS i Mnesia.

Procés de tractament d'errors

- **El disseny de sistemes:**
  - **Distribuïts**
  - **Tolerants a fallades**
  - **Escalables**

La reputació d'erlang per gestionar aspectes sistemes tolerants a fallades i d'alta disponibilitat té els seus fonaments en les estructures simples però poderoses construïdes sobre el model de concurrència del llenguatge.

Permeten

- **Fer processos per controlar el comportament d'altres processos**
- **Recuperar-se d'errors d'execució**

Donen a l'Erlang un avantatge competitiu sobre d'altres llenguatges de programació, ja que faciliten el desenvolupament de la complexa arquitectura que ofereix la tolerància a fallades i l'aïllament dels errors i garanteix un funcionament sense aturades

Processos enllaçats i senyals de sortida

- Enllaç (link)
  - és un tipus específic de relació que es pot crear entre dos processos.
  - Quan s'estableix aquesta relació i un dels processos mor (per una caiguda, un error o sortida inesperada), el procés enllaçat també mor.
  - Es tracta de detectar tan aviat com es pugui la fallada per aturar els errors: si en un procés es produeix un error, es mort, però els processos que en depenen no, llavors tots aquests processos dependents han de tractar amb una dependència desapareguda.
  - Deixar-los morir i després reiniciar tot el grup és una alternativa acceptable.
- Els enllaços ens permeten fer-ho

Processos enllaçats i senyals de sortida

Per establir un enllaç entre dos processos, Erlang té la funció primitiva:

- Link/1

L'argument és el Pid de procés que es vol enllaçar

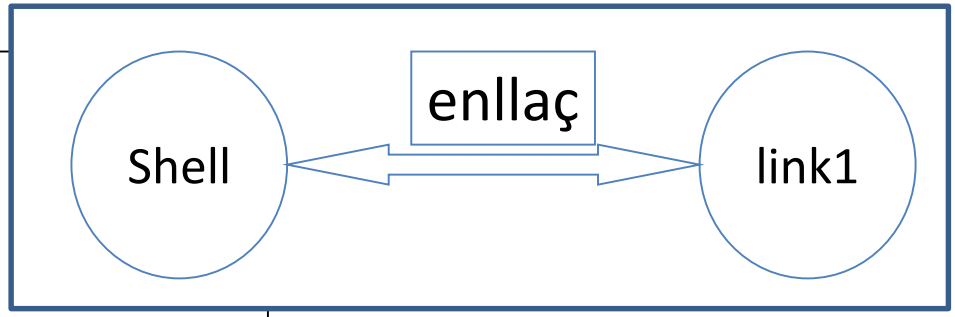
- Quan és crida la funció es crea un enllaç entre el procés actual i el procés identificat pel Pid.
- Un enllaç es pot desfer, amb la funció unlink/1.
- Quan un dels processos enllaçats mor, s'envia un tipus especial de missatge, amb informació relativa al fet que ha succeït.
- Aquest 'missatge' no s'envia si un procés mor per causes naturals (és a dir: que acaba l'execució de les seves funcions)

Processos enllaçats i senyals de sortida

Exemple:

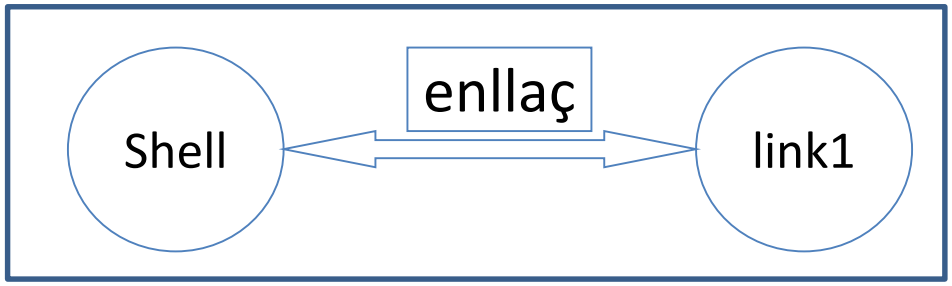
```
-module(link1).
-export([proces/0]).

proces() ->
        timer:sleep(5000),
        exit("s'ha acabat el temps").
```

```
1> c(link1).
{ok,link1}
2> spawn(fun link1:proces/0).
<0.129.0>
3> link(spawn(fun link1:proces/0)).
true
** exception error: "s'ha acabat el temps"
```



Processos enllaçats i senyals de sortida

- La funció "link/1“ crea un vincle bidireccional entre el procés que el crida i el procés indicat pel pid. En els diagrames de processos d’Erlang, es mostren els processos vinculats entre si per una línia.



Si fem:
Pid = spawn (Pro)).
link(Pid)

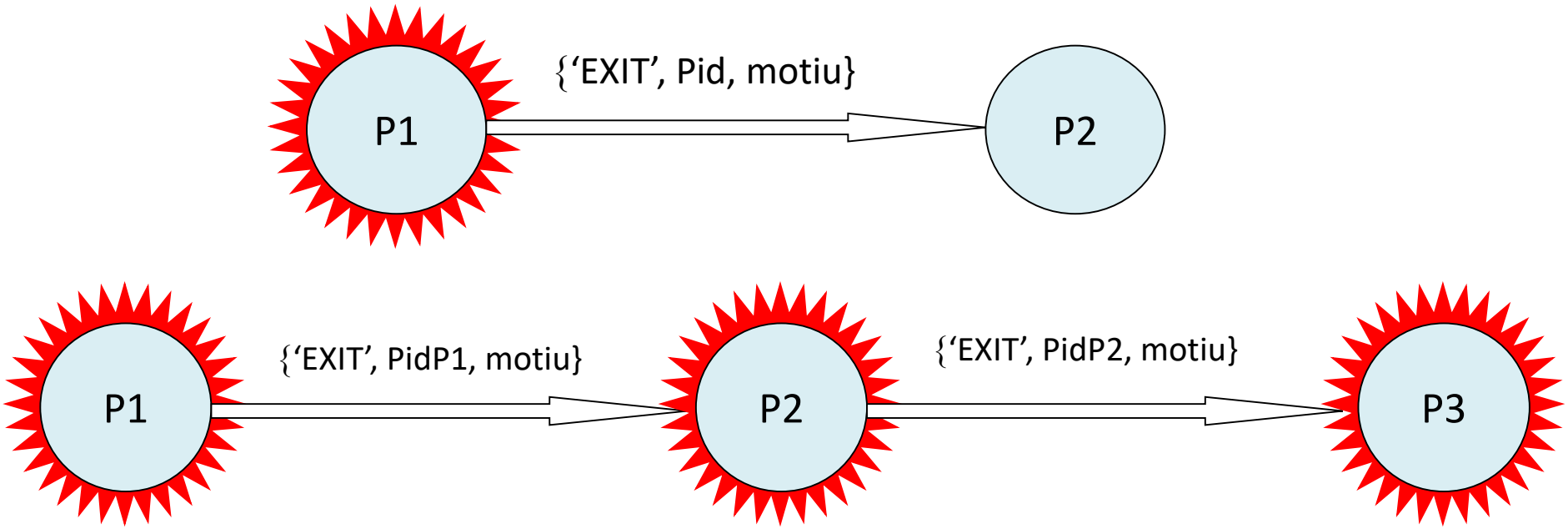
Si el procés *Pro* peta abans de poder crear l’enllaç pot provocar un comportament inesperat

- Funció spawn_link/1-4.
  - spawn_link(Fun) -> Pid
  - spawn_link(Node, Fun) -> Pid
  - spawn_link(Module, Fun, Args) -> Pid
  - spawn_link(Node, Module, Fun, Args) -> Pid

Funció atòmica -> Una sola operació

Processos enllaçats i senyals de sortida

- Els enllaços són bidireccionals, tant es si un procés P1 està vinculat al procés P2 o P2 a P1, el resultat serà el mateix.
- Si un procés enllaçat finalitza de forma anormal, s'envia un missatge a tots els processos que estan vinculats amb el procés que falla.
- El procés de recepció del senyal de sortida, propaga el senyal de sortida als processos vinculats (aquesta col·lecció és coneix amb el nom de conjunt d'enllaços).



Processos enllaçats i senyals de sortida

Exemple 1:

```
-module(add_one).
-export([start/0, request/1, loop/0]).

start() ->
  register(add_one, spawn_link(add_one, loop, [])).

request(Int) ->
  add_one ! {request, self(), Int},
  receive
      {result, Result} ->  Result
      after 1000       ->  timeout
  end.

loop() ->
  receive
      {request, Pid, Msg} ->
          Pid ! {result, Msg + 1}
  end,
  loop().
```

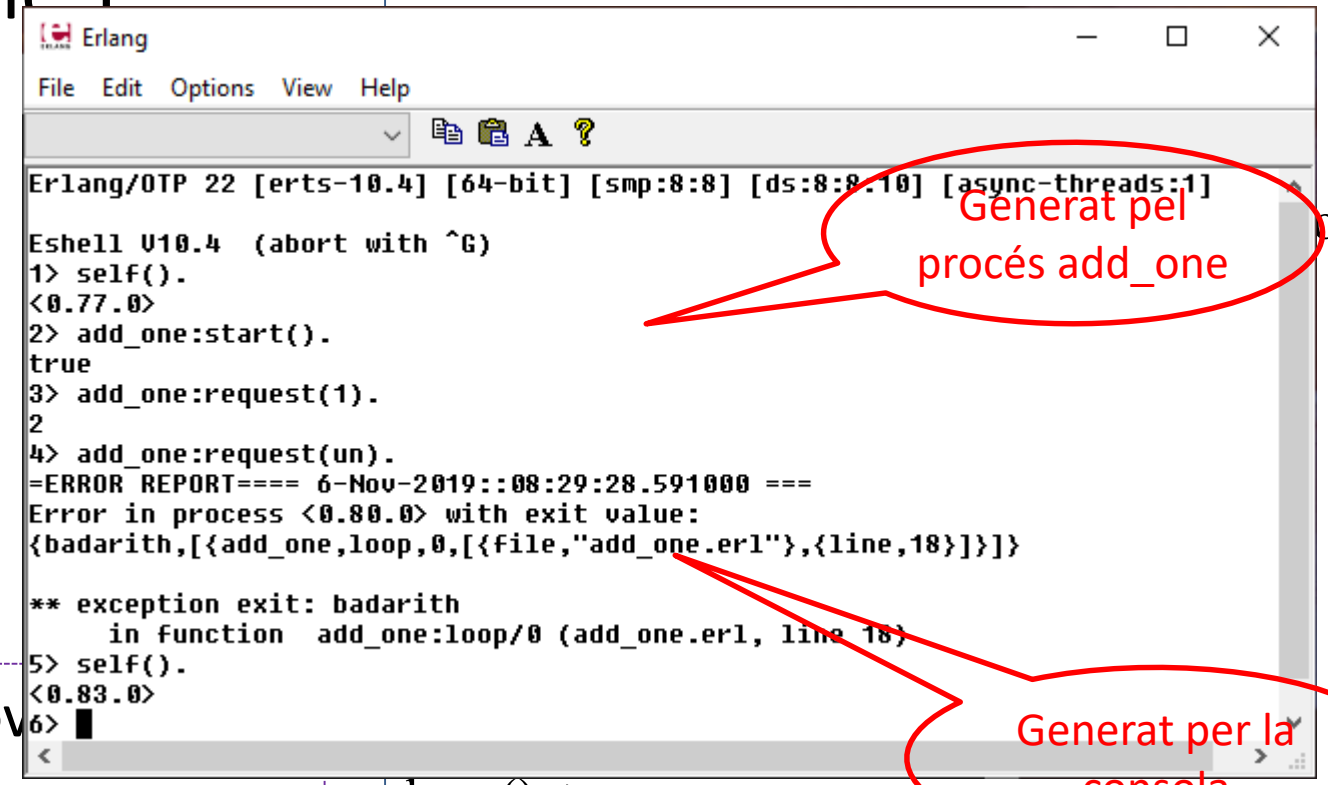
Joc de proves:

```
1> self().
2> add_one:start().
3> add_one:request(1).
4> add_one:request(un).
5> self().
```

Processos enllaçats i senyals de sortida

Exemple 1:

```
-module(add_one).
```




```
loop() ->
  receive
      {request, Pid, Msg} ->
          Pid ! {result, Msg + 1}
  end,
  loop().
```

Joc de prov

```
1> self().
2> add_one:start().
3> add_one:request(1).
4> add_one:request(un).
5> self().
```

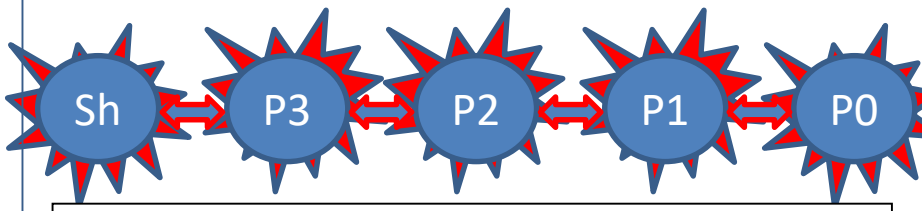
Processos enllaçats i senyals de sortida

Exemple 2:

```
-module(link2).
-export([cadena/1]).

cadena(0) ->
    receive
        _ -> ok
    after 2000 ->
        exit("Aquí mor la cadena“)
    end;

cadena(N) ->
    Pid = spawn(fun() -> cadena(N-1) end),
    link(Pid),
    receive
        _ -> ok
end.
```

Joc de proves:

```
1>c(link2).
{ok,link2}
2> link(spawn(link2, cadena, [3])).
true
** exception error: " Aquí mor la cadena "
```



[shell] == [3] == [2] == [1] == [0]
[shell] == [3] == [2] == [1] == *mor*
[shell] == [3] == [2] == *mor*
[shell] == [3] == *mor*
[shell] == *mor*
mor, missatge d’error mostrat
[shell] <-- reiniciat

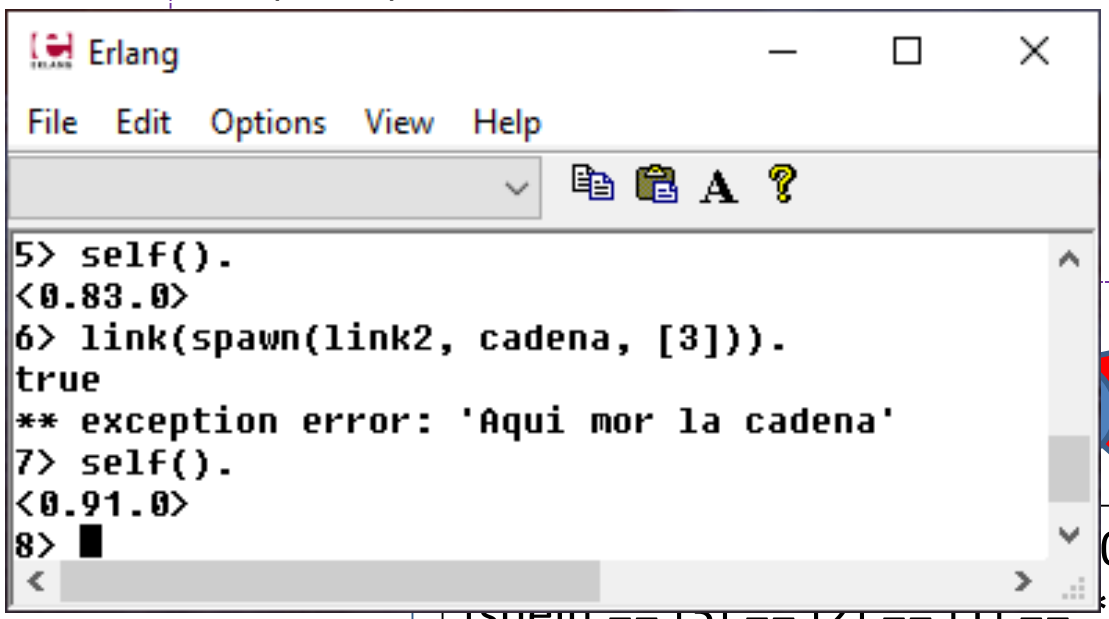
Processos enllaçats i senyals de sortida

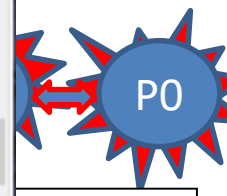
Exemple 2:

```
-module(link2).
-export([cadena/1]).

cadena(0) ->
    receive
        _ -> ok
    after 2000 ->
        exit("Aquí mor la
    end;

cadena(N) ->
    Pid = spawn(fun() -> cadena(N-1) end),
    link(Pid),
    receive
        _ -> ok
end.
```

Joc de proves:

1>c(link2).

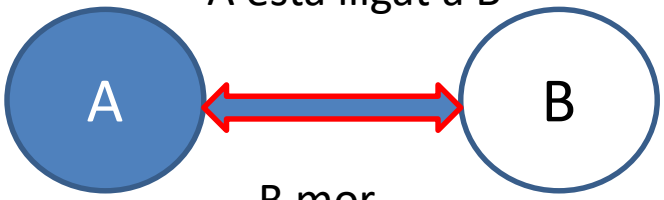




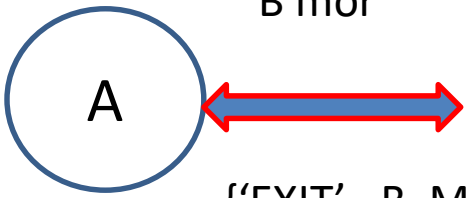
0]
mor
[shell] == [3] == [2] == *mor*
[shell] == [3] == *mor*
[shell] == *mor*
mor, missatge d'error mostrat
[shell] <-- reiniciat

Processos enllaçats i senyals de sortida: Trampes

A està lligat a B

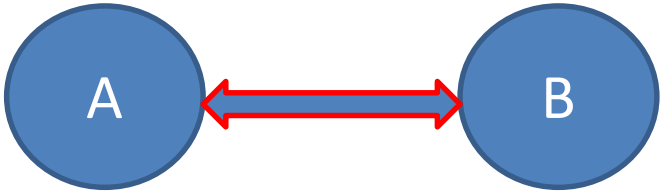


B mor



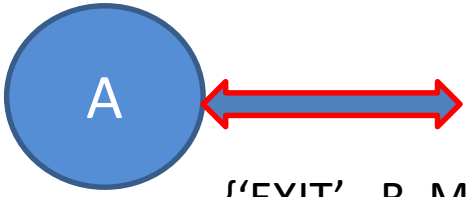
{'EXIT' , B, Motiu}
S'envia un senyal a A
A es mor

Podem variar aquest comportament segons les nostres necessitats.
Per això, el procés A atrapa el senyal de sortida i el processa decidint que és el que vol fer en aquest moment.



A atrapa les sortides de B
A està lligat a B

B mor



{'EXIT' , B, Motiu}
S'envia un senyal a A
A No es mor

Processos enllaçats i senyals de sortida: Trampes

Però, com sap el procés B a qui ha d'avisar?

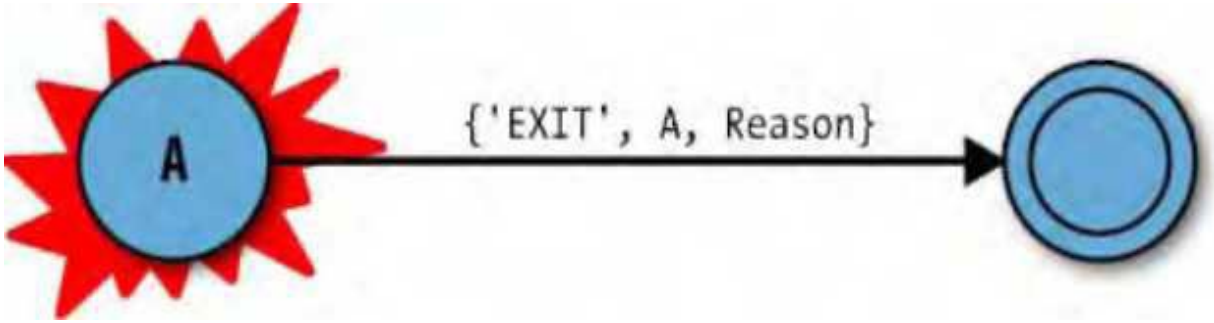
- El procés B té un conjunt de processos enllaçats. Quan un procés es vincula a un altre procés registra el seu PID en un conjunt de processos enllaçats.
- Quan el procés enllaçat acaba, envia el senyal de sortida a tots els processos enllaçats.

Trampes activades	**Senyal de sortida**	**Comportament**
Si	kill	El procés mor i transmet el senyal 'kill' al conjunt de processos enllaçats.
Si	Motiu	Es rep un missatge *{'EXIT', Pid, Motiu}*.
No	normal	Continua, no fa res. Ignora el senyal.
No	kill	El procés mor i transmet el senyal 'kill' al conjunt de processos enllaçats. Moren tots
No	Motiu	El procés mor i transmet el senyal 'Motiu' al conjunt de processos enllaçats. Moren tots

Processos enllaçats i senyals de sortida: Trampes

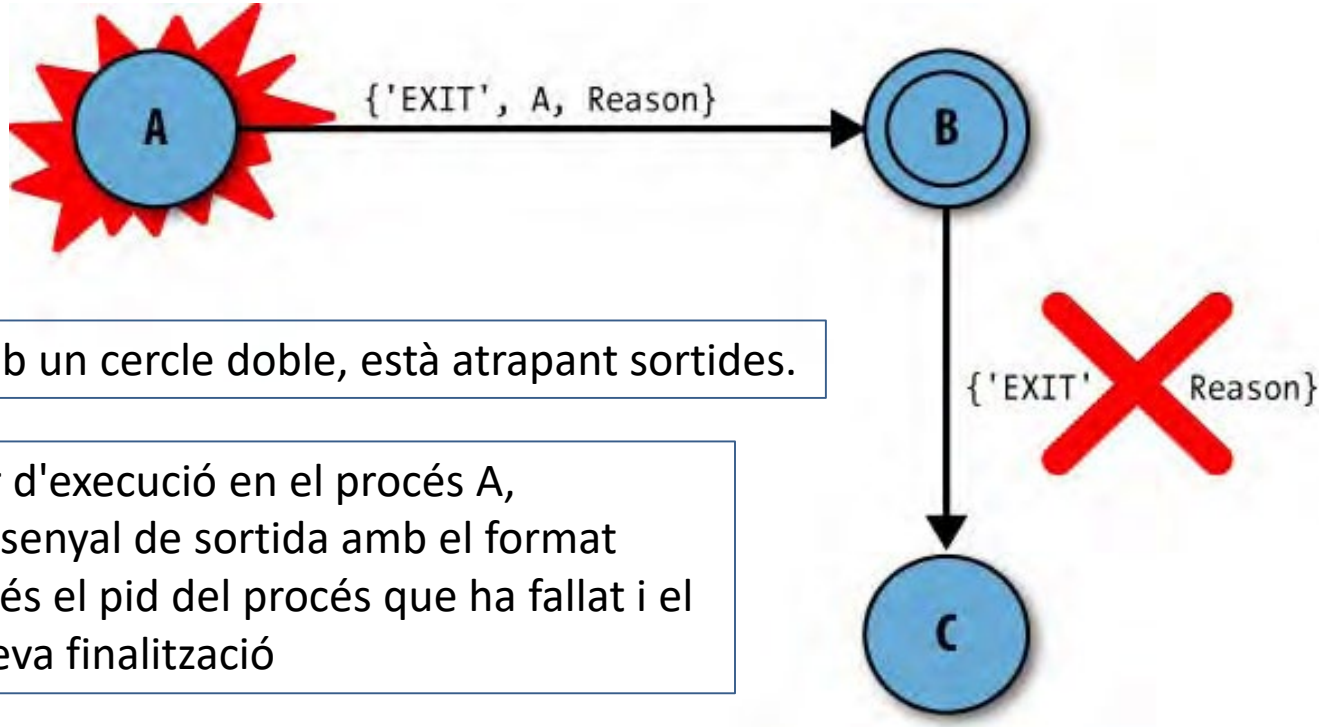
process_flag (trap_exit, true)

- Per atrapar els senyals de sortida s'ha d'activar el flag -> trap_exit
- Amb la funció process_flag (trap_exit, true)
- La crida es fa generalment en la funció d'inicialització, permetent que els senyals de sortida es converteixen en missatges amb el format {'EXIT', Pid, raó}.
- Si un procés està atrapant sortides, aquests missatges es guarda a la bústia de procés exactament de la mateixa manera que altres missatges.
- Es poden recuperar aquests missatges mitjançant la recepció, i posant sobre ells la coincidència de patrons com qualsevol altre missatge.



Processos enllaçats i senyals de sortida: Trampes

process_flag (trap_exit, true)



El Procés B, marcat amb un cercle doble, està atrapant sortides.

Si es produeix un error d'execució en el procés A, finalitzarà i enviarà un senyal de sortida amb el format {'EXIT', A, Motiu} on A és el pid del procés que ha fallat i el Motiu és la raó de la seva finalització

L'àtom 'EXIT' s'utilitza per etiquetar la tupla i facilitar la comparació de patrons a la recepció.

Aquest missatge s'emmagatzema a la bústia del procés B sense afectar a C.

Llevat que B explícitament informi a C que A ha acabat, C mai ho sabrà.

Processos enllaçats i senyals de sortida: Trampes

Exemple 1

```
1> process_flag(trap_exit, true).
false
2> add_one:start().
true
3> add_one:request(one).

=ERROR REPORT==== 13-Nov-2012::15:37:43 ===
Error in process <0.198.0> with exit value:
{badarith,[{add_one,loop,0,[{file,"add_one.erl"},{line,17}]}]}

timeout
4> flush().
Shell got {'EXIT',<0.198.0>,{badarith,[{add_one,loop,0,
[{file,"add_one.erl"},{line,17}]}]}}
ok
```

Processos enllaçats i senyals de sortida: Trampes

Exempl

```
1> process_
false
2> add_one
true
3> add_one

=ERROR REI
Error in pro
{badarith,[{

timeout
4> flush().
Shell got {'E
[{file,"add_
ok
```

Erlang

File Edit Options View Help

```
** exception error: 'Aqui mor la cadena'
7> self().
<0.91.0>
8> process_flag(trap_exit, true).
false
9> add_one:start().
true
10> add_one:request(un).
=ERROR REPORT==== 6-Nov-2019::08:48:17.484000 ===
Error in process <0.94.0> with exit value:
{badarith,[{add_one,loop,0,[{file,"add_one.erl"},{line,18}]}]}

timeout
11> flush().
Shell got {'EXIT',<0.94.0>,
                  {badarith,[{add_one,loop,0,
                                      [{file,"add_one.erl"},{line,18}]}]}}
ok
12> self().
<0.91.0>
13>
```

Exemple 2

```
-module(add_two).
-export([start/0, request/1, loop/0]).

start() ->
      process_flag(trap_exit, true),
      Pid = spawn_link(add_two, loop, []),
      register(add_two, Pid),
      {ok, Pid}.

request(Int) ->
       add_two ! {request, self(), Int},
       receive
            {result, Result}          -> Result;
            {'EXIT', _Pid, Reason} -> {error, Reason}
            after 1000                -> timeout
       end.

loop() ->
      receive
            {request, Pid, Msg} ->
               Pid ! {result, Msg + 2}
      end,
      loop().
```

Joc de proves:

```
1> self().
2> add_two:start().
3> add_two:request(26).
4> add_two:request(six).
5> self().
```

Exe

Joc

1> se
2> ad
3> ad
4> ad
5> se

```
-module(add_two).
-export([start/0, request/1, loop/0]).
```

Erlang

File Edit Options View Help

```
12> self().
<0.91.0>
13> add_two:start().
{ok,<0.99.0>}
14> add_two:request(26).
28
15> add_two:request(one).
=ERROR REPORT==== 6-Nov-2019::08:50:42.069000 ===
Error in process <0.99.0> with exit value:
{badarith,[{add_two,loop,0,[{file,"add_two.erl"},{line,21}]}]}

{error11111,{badarith,[{add_two,loop,0,
                        [{file,"add_two.erl"},{line,21}]}]}}
16> self().
<0.91.0>
17>
```

```
loop() ->
    receive
        {request, Pid, Msg} ->
            Pid ! {result, Msg + 2}
    end,
    loop().
```

Processos enllaçats i senyals de sortida: Resum

- **No em fa res que acabi un procés**: es crea un procés amb spawn. Així si el procés creat falla el procés actual continua.
  - Pid = spawn (fun() -> ... end)
- **Vull morir si el meu procés fill mor**: sempre que el procés creat no acabi amb un senyal de sortida normal el procés que el crea morirà.
  - Pid = spawn_link (fun() -> ... end)
- **Vull gestionar els errors si el procés que he creat falla**: en aquest cas, no només enllacem el procés que creem sinó que atrapem els senyals de sortida.

```
process_flag(trap_exit, true),
Pid = spawn_link(fun() -> ... end),
  ...
loop(...).

loop(Estat) ->
  receive
      {'EXIT', Pid, Motiu} ->  %% tractem l'error capturat
            loop(Estat1);
      ...
  end
```

Monitors

- Ja hem vist com realitzar un control d'errors en sistemes concurrents. -> link i trap_exit.
- Els processos enllaçats donen mecanisme pel qual, dos processos A i B quedaven enllaçats de tal manera que, si el procés A mor envia un senyal ‘exit’ al procés B i viceversa.
- Amb aquest mecanisme s’evita que hi hagi processos moribunds en el nostre sistema.
- Potser no ens interessa un control tan ferri com el que ens proporciona el sistema de processos enllaçats.
- Potser, només necessitem saber si un procés és viu o no, per a poder-lo utilitzar, o no.
- Potser, no representa cap problema seguir treballant sense el procés que s’ha mort.
- En aquests casos, el que es necessita és: un monitor.
- Un monitor és un tipus d'enllaç asimètric, mentre que els enllaços (link) són de caràcter simètric.
  - Si el procés M monitoritza al procés A, el procés M ho sabrà tot sobre el procés A, però no a l'inrevés.

Monitors

- Un exemple que il·lustri clarament aquest comportament és el d'un servidor de suport.
  - Imagineu que disposem d'un servidor central i un altre de suport.
  - El servidor central un dia decideix caure.
  - No estaria bé que els clients poguessin monitoritzar el servidor?
    - Perquè en el cas de caigudes del servidor central passar automàticament al servidor de suport.
  - Un comportament molt desitjable en molts casos.

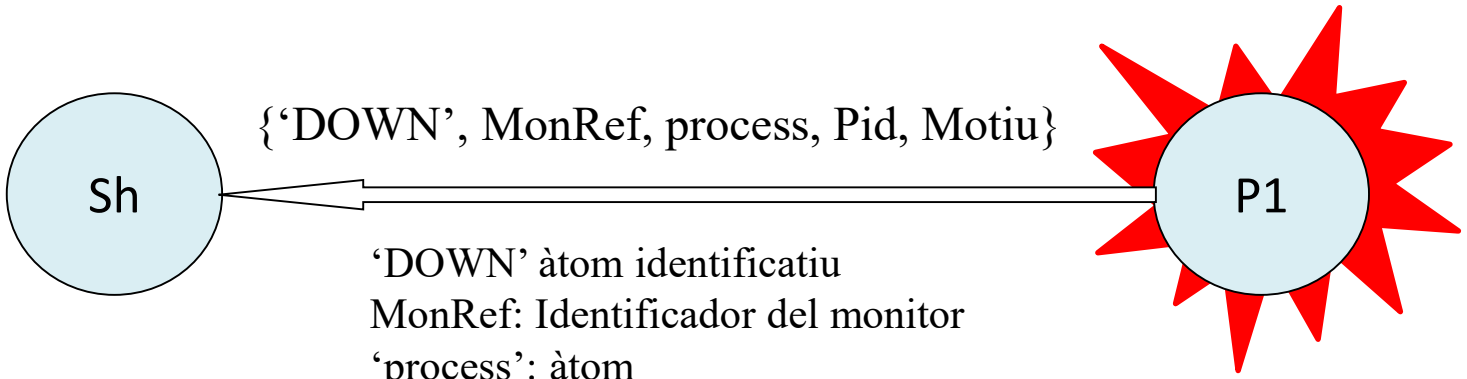
- Els monitors són un tipus especial d'enllaç amb dues diferències:
  - Són unidireccionals
  - Poden ser apilats
  - Envien missatges i no senyals com el link
- Els Monitors són la solució quan un procés vol saber el que està passant amb un segon procés, però que cap dels dos és vital per l'altre.
- Les referències son apilables. És genial per escriure biblioteques que necessiten saber que està succeint amb altres processos.

Monitors

- Funció BIF: monitor/2
- El primer paràmetre és l'àtom 'process' i el segon és l'identificador del procés que es vol monitoritzar.
  - MonRef = monitor(process, Pid). Retorna l'identificador del monitor
  - Es pot desfer el control amb: demonitor(MonRef).

Exemple: monitor(process, spawn(fun() -> timer:sleep(5000) end)).

Si P1 cau:

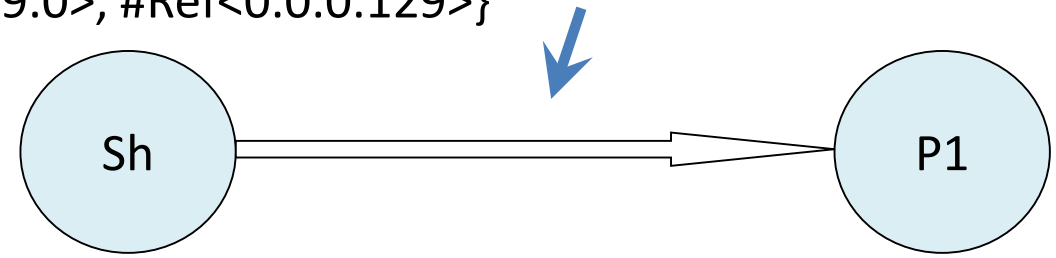


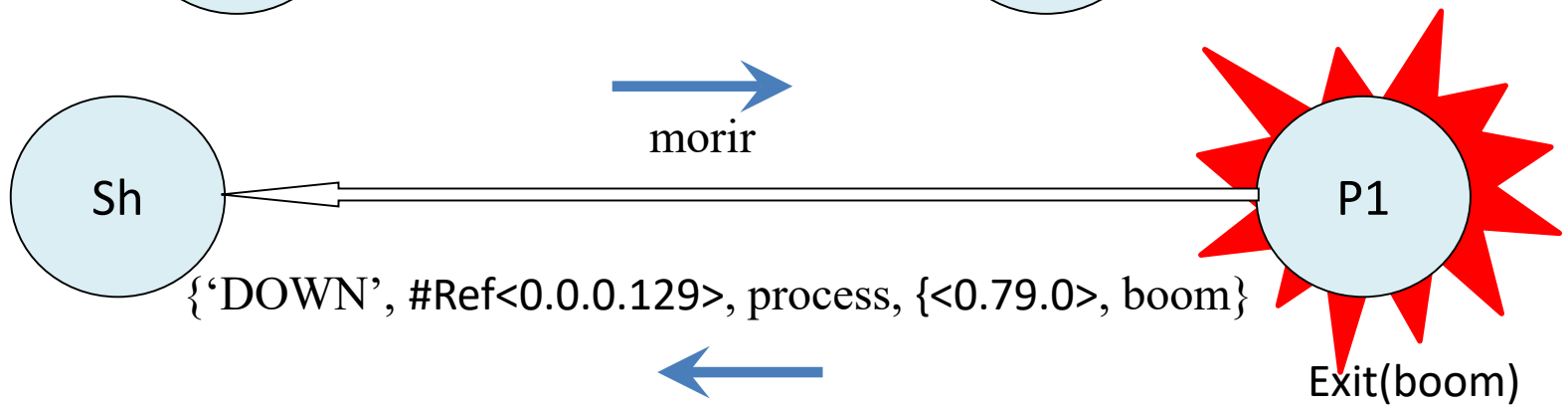
'DOWN' àtom identificatiu
MonRef: Identificador del monitor
'process': àtom
Pid: Identificador del procés que ha caigut
Motiu: Justificació de la caiguda

Monitors

- La funció atòmica: spawn_monitor/1 o 3
  - spawn_monitor(Fun) -> {Pid, RefMon}
  - spawn_monitor(Module, Function, Args) -> {Pid, RefMon}

Exemple: {Pid, Ref} = spawn_monitor(fun() -> receive _ -> exit(boom) end end).
{<0.79.0>, #Ref<0.0.0.129>}





>Pid!morir
Morir

>flush()
Shell got {'DOWN',#Ref<0.0.0.147>,process,<0.87.0>,boom}
ok

Monitors: Exemple (Pacient)

```
-module(pacient).
-export([start/0,reanimar/0,mort/0, mort_natural/0, asfixia/0, aturada_cardiaca/0, missatge/1]).
 start ()              -> register(?MODULE, spawn(fun proces/0)).
 reanimar ()           -> start().
 proces()              ->
     receive
          mort ->
               io:format ("El pacient s'ha mort. Hora de la mort ~p ~n", [erlang:localtime()]),
               exit(normal);
          {alarma, Motiu} ->
               io:format ("Monitoritzant alarma ~p~n", [Motiu]),
               exit(Motiu);
          UnAltre ->
               io:format("Rebut <~p> ~n", [UnAltre]),
               proces()
     end.

mort()                -> ?MODULE!mort.
mort_natural()        -> ?MODULE!{alarma, mort_natural}.
asfixia()             -> ?MODULE!{alarma, asfixia}.
aturada_cardiaca()    -> ?MODULE!{alarma, aturada_cardiaca}.
missatge(M)           -> ?MODULE!M.
```

Monitors: Exemple (monitor)

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

start(Pid) ->
    spawn(?MODULE, loop, [Pid]).

loop(Pid) ->
    io:format("Monitoritzan al pacient ~p. Constants estables.~n", [Pid]),
    monitor(process,Pid),
    receive
        {'DOWN', Ref, process, Pid,  normal} ->
            io:format("~p diu que  ~p ha mort per causes naturals~n",[Ref,Pid]);
        {'DOWN', Ref, process, Pid,  Motiu} ->
            io:format("~p diu que  ~p ha mort per ~p~n",[Ref,Pid,Motiu]);
        UnAltre ->
            io:format("Monitor rep <~p>~n", [UnAltre])
    end.
```

Monitors: Exemple (reanimador)

```
-module(reanimador).
-compile(.....).

start() ->
    pacient:start(),
    spawn(?MODULE, loop, [whereis(pacient)]).

loop(Pid) ->
    io:format("Monitoritzant al pacient ~p. Constants estables.~n", [Pid]),
    monitor(process,Pid),
    receive
        {'DOWN', Ref, process, Pid,  normal} ->
            io:format("~p diu que  ~p ha mort~n",[Ref,Pid]);
        {'DOWN', Ref, process, Pid,  Motiu} ->
            io:format("~p: Alarma. El pacient ~p està patint un/a ~p~n",[Ref,Pid,Motiu]),
            io:format("Reanimant al pacient. Torna ... torna ... viu~n"),
            pacient:reanimar(),
            loop(whereis(pacient))
    end.
```

Missatge 'normal'

Missatge amb un motiu

Monitors

- monitor_node(Node, true|false)
- Serveix per supervisar l'estat del node. S'activa amb un 'true' i es desactiva amb un 'false'.
- Al igual que els monitors, les crides repetides per fer el monitoratge d'un node, s'acumulen.
- Si el node falla o no existeix, es dona el missatge del tipus {nodedown, Node}. Si un procés a fet dues crides per monitoritzar el node, quan el node finalitza s'envien dos missatges del tipus 'nodedown'. Si no hi ha la connexió al node, es farà l'intent de crear-ne una, si falla, donarà un missatge del tipus 'nodedown'.

Funció: try ... catch

- Es tracta d'una sentencia *case.* La seva sintaxis es:

```
try funciónOExpresión of
    Patró1 [when Guarda1] -> Expressió1;
    ...
    PatrónN [when GuardaN] -> ExpressioN
catch
    TipusExcepcio1: ExcepcióPatró1 [when ExcepcióGuarda1] -> ExcepcióExpressio1;
    ...
    TipusExcepciónN: ExcepcióPatrónN [when ExcepcióGuardaN] -> ExcepcióExpressioN
after
    AfterExpresions
end
```

Es poden ometre parts de la sentència:

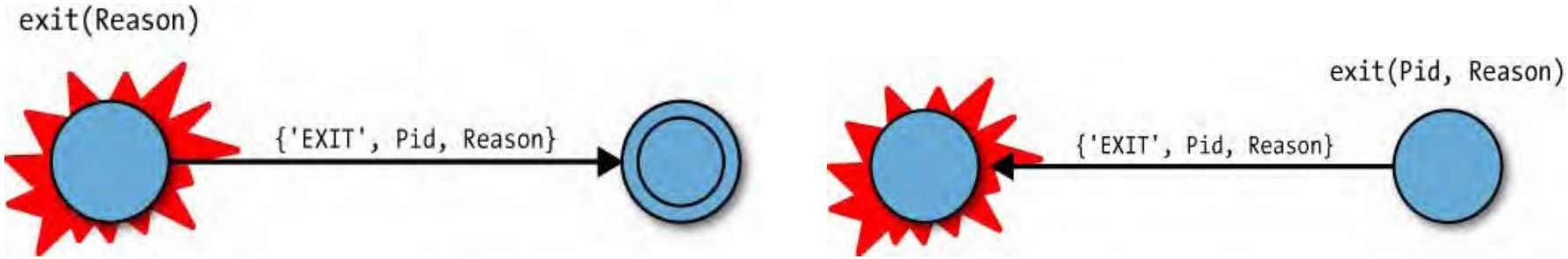
```
try funcióOExpresió of
catch
    TipusExcepcio1: ExcepcióPatró1 [when ExcepcióGuarda1] -> ExcepcióExpressio1;
    ...
    TipusExcepciónN: ExcepcióPatrónN [when ExcepcióGuardaN] -> ExcepcióExpressioN
end
```

Generació d'errors

- Es pot generar un error o enviar una excepció amb les següents funcions:
  - **exit (Motiu):** aquesta funció s'utilitza quan es vol acabar el procés actual. Si l'excepció no és capturada, el missatge {'EXIT', Pid, Motiu} es difondria a tots els processos que estan enllaçats al procés actual.
  - **throw (Motiu):** s'utilitza per enviar una excepció. L'usuari d'una funció que envia una excepció té dues possibilitats: ignorar-la o, tractar amb un try ... Catch.
  - **error(Motiu):** s'utilitza per errors greus. I no s'espera cap gestió per part de l'usuari de la funció que l'utilitza.

Funció exit

- La funció exit(Motiu) finalitza el procés que el crida i el motiu de la finalització és l'argument que es passa a la funció. El procés de finalització genera un senyal de sortida que s'envia a tots els processos als quals està vinculat.
- Si el motiu es diferent de l'àtom 'normal', acaba de forma anormal -error d'execució-
- Si exit/1 es crida dins d'una construcció del tipus try ... Catch es pot capturar, ja que és dins d'una mateixa captura.



- Si es vol enviar un senyal de sortida a un procés particular, es pot cridar la funció de sortida exit (Pid, Reason)

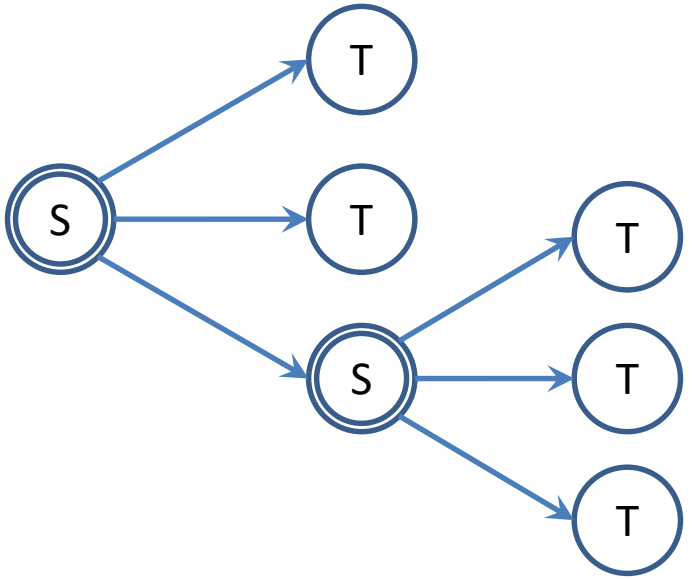
Funció: try ... catch: Exemple

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

provocar_error(1) ->  a;
provocar_error(2) -> throw(a);
provocar_error(3) -> exit(a);
provocar_error(4) -> {'EXIT', a};
provocar_error(5) -> error(a).

prova() -> [capturar(I) || I <- [1,2,3,4,5]].

capturar(N) ->
    try provocar_error(N) of
        Val -> {N, noCapturat, Val}
    catch
        throw:X -> {N, capturat, throw, X};
        exit:X -> {N, capturat, exit, X};
        error:X -> {N, capturat, error, X}
    end.
```

```
1> c(trycatch).
{ok,trycatch}
2> trycatch:prova().
[{1,noCapturat,a},
 {2,capturat,throw,a},
 {3,capturat,exit,a},
 {4,noCapturat,{'EXIT',a}},
 {5,capturat,error,a}]
```

Sistemes robustos

- En Erlang, es poden construir sistemes robusts per capes.
- Utilitzant processos, es crea un arbre en el que els fulls consisteixen la capa d'aplicació que controla les tasques operatives mentre que els nodes interiors supervisen els fulls i els altres nodes per sota d'ells
- Els processos de qualsevol nivell poden atrapar errors que es produeixen en un nivell per sota seu.
- Un supervisor és un procés que supervisa el fills (nodes de l'arbre).
- Un treballador és un procés (full) que realitza tasques operatives.
- Els processos fills, son: els supervisors i els treballadors que pertanyen a un supervisor particular.

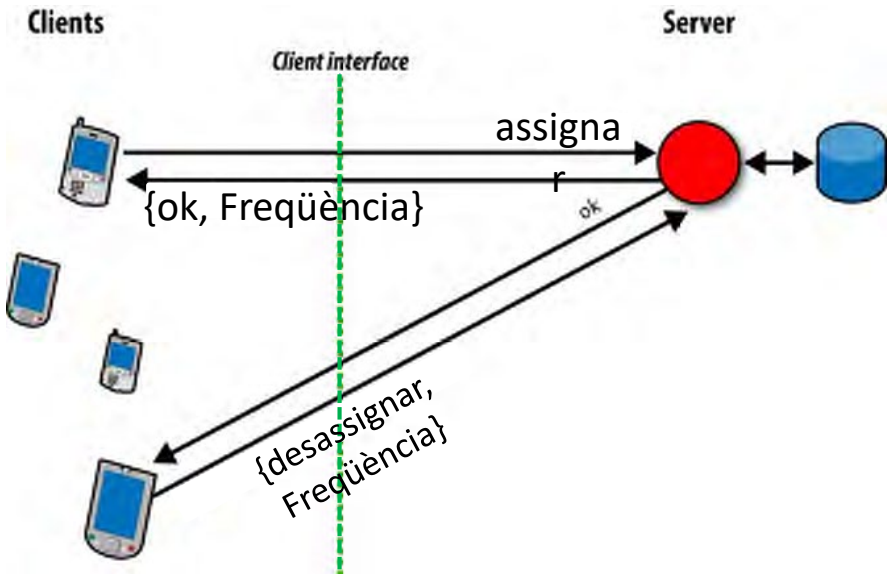


S -> Supervisor
T -> Treballador

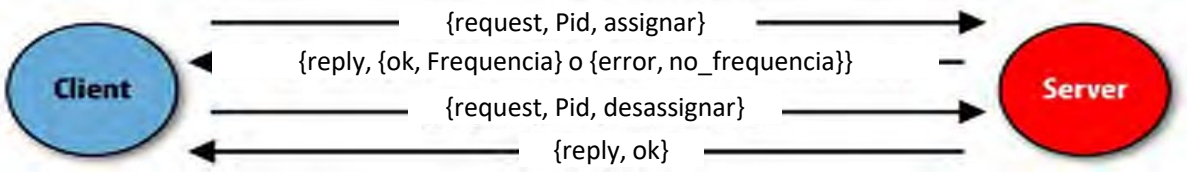
Mòdul supervisor

Exemple: Models Client / Servidor. Descripció del problema

- Es vol gestionar l'assignació de freqüències de ràdio per uns telèfons mòbils connectats a la xarxa. El telèfon demana que se li assigni una freqüència cada vegada que fa una trucada, i l'allibera quan la trucada ha acabat.
- Els client seran els diferents telèfons que poden intentar establir una trucada
- El servidor serà el que assigna les freqüències de ràdio
- Quan un telèfon mòbil vol establir una connexió amb un altre abonat, crida a la funció del client per demanar una freqüència *freqüència:assignar()*. Aquesta funció genera un missatge síncron (assignar) que s'envia al servidor.
- El servidor controla i respon amb un missatge que conté una freqüència disponible {ok, Freqüència} o un error si totes les freqüències estan utilitzades {error, no_freqüències }.



- Quan el client finalitza la trucada telefònica ha d'alliberar la connexió, s'ha de cancel·lar l'assignació de la freqüència perquè altres clients la pugin utilitzar. I ho fa cridant a la funció de client freqüència:desassignar (Freqüència). La crida genera un missatge {desassignar, Freqüència} que s'envia al servidor.
- El servidor pot assignar la freqüència disponible a altres clients i respon amb l'àtom de correcte (ok).



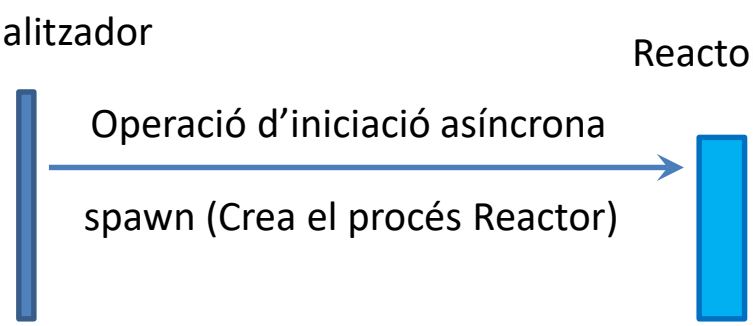
Exemple: Models Client / Servidor: Solució

- La primera part del codi del mòdul d'assignació de les freqüències, corresponent al servidor, pot ser:

```
-module(frequencia).
-export([iniciar/0, stop/0, assignar/0, desassignar/1]).
-export([init/0]).

% Funcions d'inicialització per crear i inicialitzar el servidor.
iniciar() ->
    register(frequencia, spawn(frequencia, init, [])).

init() ->
    Frequencies = {get_frequencies(), []},
    loop(Frequencies).

% Codificat
get_frequencies() -> [10,11,12,13,14,15].
```

Funcions del servidor: iniciar/0

La funció *iniciar/0* genera un procés init/0 que inicia l'execució del mòdul de freqüència.
spawn retorna el pid del procés que es passa com a segon paràmetre a la funció (¡BIF) *register*. El primer paràmetre és l'àtom *frequencia*, que és l'alies amb el que estem registrant el procés.
Aquesta assignació segueix el conveni de registrar un procés amb el mateix nom que el mòdul en què es defineix.
El nou procés (*init*) crea una tupla amb la llista de les freqüències disponibles (funció: *get_frequencies/0)*, i una llista de les freqüència assignades ([]). La tupla, serà el que anomenem les dades o cicle d'estat, s'afegeix a la variable *Frequencies,* que s'usa com a paràmetre de la funció de recepció avaluació (reactor), que en aquest exemple s'ha anomenat loop/1.



Exemple: Models Client / Servidor: Solució

```
% El bucle principal
loop(Frequencies) ->
    receive
        {request, Pid, assignar} ->
            {NovesFrequencies, Reply} = assignar(Frequencies, Pid),
            reply(Pid, Reply),
            loop(NovesFrequencies);
        {request, Pid , {desassignar, Freq}} ->
            NovesFrequencies = desassignar(Frequencies, Freq),
            reply(Pid, ok),
            loop(NovesFrequencies);
        {request, Pid, stop} ->
            reply(Pid, ok)
    end.

reply(Pid, Reply) ->
    Pid ! {reply, Reply}.
```

- Funcions del servidor: loop/1

Missatges:

{request, Pid, Missatge}

patró

Identificador del client

Acció que s'ha de realitzar

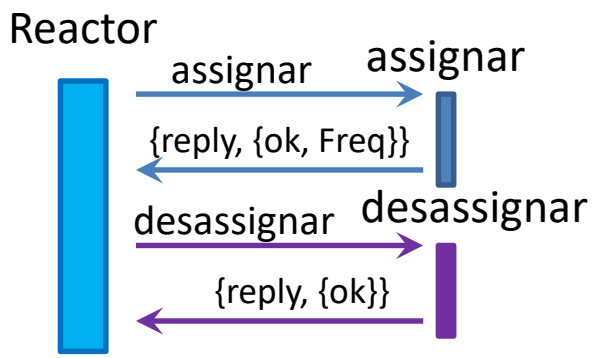
Acceptarà rebre tres tipus de peticions:

- assignar,
- desassignar i
- stop.

El *Missatge* és el patró lligat a l'expressió i s'utilitza per determinar quin clàusula s'ha d'executar.

Aquestes funcions retornen un respostes al client, que en aquest cas consisteix en la freqüència assignada o un ok.

El *pid* del client s'ha enviat com a part de la petició, i s'utilitza per identificar el procés que l'ha cridat i retornar-li la resposta a *reply/2*.

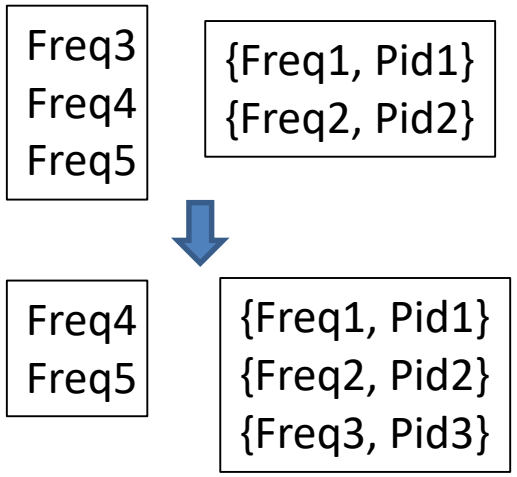


Exemple: Models Client / Servidor: Solució

- Funcions del servidor

```
% Funcions internes per assignar i alliberar freqüències.

assignar({[], Assignat}, _Pid) ->
{{[], Assignat}, {error, no_frequencia}};

assignar({[Freq|Lliure], Assignat}, Pid) ->
{{Lliure, [{Freq, Pid}|Assignat]}, {ok, Freq}}.

desassignar({Lliure, Assignat}, Freq) ->
NouAssignat=lists:keydelete(Freq, 1, Assignat),
      {[Freq|Lliure],  NouAssignat}.
```

{[Freq|Lliure], Assignat}



assignar/2

- Si no hi ha freqüències disponibles, *assignar/2* coincideix amb el patró de la primera clàusula, ja que el primer element de la tupla que conté la llista de freqüències disponibles està buida. Retorna la tupla {error, no_frequencia} juntament amb de les dades del bucle sense canvis.
- Si com a mínim hi ha una freqüència disponible coincidirà amb la segona clàusula. La freqüència s'elimina de la llista de les freqüències disponibles, i es passa a la llista de les freqüències assignades juntament amb el pid del client i.

desassignar/2

- Eliminà de la llista de freqüències assignades la freqüència *Freq* utilitzant la funció de la llibreria *lists:keydelete/3* i l'afegeix a la llista de freqüències disponibles.

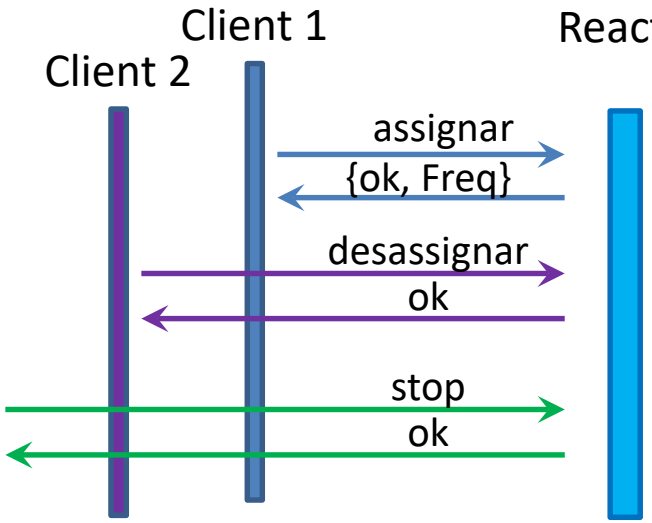
Retorna una llista de tuples que s'ha eliminat la tupla que conté l'element *key* de la posició de la tupla *N*

Exemple: Models Client / Servidor: Solució

```
%  Les funcions del client
stop()      -> call(stop).
assignar() -> call(assignar).
desassignar(Freq) -> call({desassignar, Freq}).

% Amaguem el pas de missatges i el protocol de missatge en una interfície funcional.
call(Missatge) ->
    frequencia ! {request, self(), Missatge},
    receive
        {reply, Reply} -> Reply
    end.
```

- Funcions del client

Al tractar-se d'una transferència síncrona, quan s'ha enviat el missatge, el client es queda esperant la resposta
La resposta del servidor {reply, Reply} és l'àtom 'reply', i la variable *Reply* retorna el valor de les funcions de client.



La crida a la funció *call/1*, passa el missatge que s'ha d'enviar al servidor com un argument. Aquesta funció encapsula el protocol de missatges entre el servidor i els seus clients, enviant un missatge amb el format {request, Pid, Missatge}. L'àtom *request* és una etiqueta a la tupla, Pid és l'identificador del procés que el crida (retorn de la funció BIF self()), i *Missatge* és el missatge que es vol enviar.

Exemple: Models Client / Servidor; Solució

- Un client vol iniciar una trucada i crida a la funció *frequencia:assignar()*.
- Aquesta funció enviarà un missatge amb el format {request, Pid, assignar} al servidor de freqüències. Aquest missatge activarà la funció del servidor *assignar(Frequencies, Pid)*, on *Frequencies* és la tupla de les freqüències assignades i disponibles. La funció *assignar* comprovarà si hi ha freqüències disponibles:
  - Si hi ha freqüències disponibles, retorna la variable "NovesFrequencies"actualitzada, on la freqüència assignada s'ha mogut de la llista de freqüències disponible a la llista de freqüències assignades juntament amb el pid del client. La resposta (freqüència assignada) s'envia al client de la forma *{ok, Frequencia assinada}*.
  - Si no hi ha freqüències disponibles, les dades del bucle "NovesFrequencies"es torna sense canvis i s'envia el missatge *{error, no_frequencia }* com a resposta al client.
- La resposta al client s'envia amb la funció *reply(Pid, Missatge)*, que dóna format al missatge intern client / servidor i l'envia de tornada al client.
- La crida a la funció *bucle/1* es fa de forma recursiva, i es passen les noves dades del bucle com paràmetres.
- La funció *desassignar* funciona d'una manera similar. La funció client envia un missatge amb el format {request, Pid, desassignar} que es correspon amb el segon paràgraf de la declaració rebre del servidor. Fa una crida a la funció *desassignar(Frequencies, Freq)* i elimina la freqüència de la llista d'assignats a la de disponibles, i torna les dades actualitzades al bucle. L'àtom correcte (ok) s'envia de tornada al client, i es crida la funció *bucle/1* recursivament amb les dades actualitzades.
- Si es rep la petició d'aturada 'stop', es retorna 'ok' al procés que l'ha cridat i el servidor finalitza l'execució, ja que no hi ha cap més crida recursiva.

Exemple: Models Client / Servidor: Solució

- simulació

```
1> frequencia:iniciar().
true
2> frequencia:assignar().
{ok,10}
3> frequencia:assignar().
{ok,11}
4> frequencia:assignar().
{ok,12}
5> frequencia:assignar().
{ok,13}
6> frequencia:desassignar(11).
ok
7> frequencia:assignar().
{ok,11}
8> frequencia:stop().
ok
```

Sistemes robustos: Exemple -> Clients monitoritzats

- Exemple del client/servidor dels números de telèfon.
  - El servidor no és fiable! Si el client es bloqueja abans d'enviar el missatge per alliberar la freqüència, el servidor no pot cancel·lar l'assignació de la freqüència i permetre que altres clients pugin utilitzar-la.
- Es pot reescriure el servidor, de manera que sigui fiable mitjançant el control dels clients.
  - Quan s'assigna una freqüència a un client, el servidor s'enllaça amb el client.
  - Si un client finalitza abans de cancel·lar l'assignació d'una freqüència, el servidor rep un senyal de sortida i allibera la freqüència automàticament.
  - Si el client no acaba, i cancel·la l'assignació de la freqüència utilitzant la funció de client, el servidor elimina l'enllaç.

Sistemes robustos: Exemple -> Clients monitoritzats

```
%% Funcions d'inicialització del servidor
iniciar() -> register(frequencia, spawn(frequencia, init, [])).

init() ->
      process_flag(trap_exit, true),
      Frequencies = {get_frequencies(), []},
      loop(Frequencies).

get_frequencies() -> [10,11,12,13,14,15].

loop(Frequencies) ->
      receive
            {request, Pid, assignar} ->
                  {NovesFrequencies, Reply} = assignar(Frequencies, Pid),
                  reply(Pid, Reply),
                  loop(NovesFrequencies);
            {request, Pid , {desassignar, Freq}} ->
                   NovesFrequencies = desassignar(Frequencies, Freq),
                  reply(Pid, ok),
                  loop(NovesFrequencies);
            {'EXIT', Pid, _Reason} ->
                  NovesFrequencies= exited(Frequencies, Pid),
                  loop(NovesFrequencies);
            {request, Pid, stop} -> reply(Pid, ok)
      end.
```

```
-module(frequencia).
-export([iniciar/0,stop/0, assignar/0, desassignar/1]).
-export([init/0]).
```

Activem la bandera o flag 'trap_exit' per que converteixi els missatges d'error generats per la sortida d'un procés a missatges que es puguin rebre i processar.

S'afegeix la clàusula de recepció de missatges de {'EXIT', Pid, _Reason} per detectar la sortida del client i poder alliberar la freqüència que tenia assignada.

Sistemes robustos: Exemple -> Clients monitoritzats

```
assignar({[], Assignat}, _Pid) ->
      {{[], Allocated}, {error, no_frequencies}};
assignar({[Freq|Frequencies], Assignat}, Pid) ->
      link(Pid),
      {{Frequencies,[{Freq,Pid}| Assignat]},{ok,Freq}}.

desassignar({Lliure, Assignat}, Freq) ->
      {value,{Freq,Pid}} = lists:keysearch(Freq,1, Assignat),
      unlink(Pid),
      NouAssignat =lists:keydelete(Freq,1, Assignat),
      [Freq| Lliure], NouAssignat}.

exited({Lliure, Assignat}, Pid) ->
      case lists:keysearch(Pid,2, Assignat) of
            {value,{Freq,Pid}} ->
                  NouAssignat = lists:keydelete(Freq,1, Assignat),
                  {[Freq| Lliure], NouAssignat};
            false ->
                  {Lliure, Assignat}
      end.
```

Freqüències:
És una tupla de llistes
{Llista de freqüències lliures,
Llista de freqüències assignades}

Quan s'assigna una nova freqüència a un client, s'enllaça amb el servidor, per que pugui avisar-lo quan caigui.

Quan s'allibera una freqüència per part d'un client, es desenllaça del servidor.

Sistemes robustos: Exemple -> Clients monitoritzats

```
stop()              -> call(stop).
assignar()          -> call(assignar).
desassignar(Freq)   -> call({desassignar, Freq}).

%% Amaguem el pas de missatges i el protocol.
call(Message) ->
    frequencia ! {request, self(), Message},
    receive
        {reply, Reply} -> Reply
    end.

reply(Pid, Message) ->
    Pid ! {reply, Message}.
```

```
1> frequencia:iniciar().
true
2> frequencia:assignar().
{ok,10}
3> frequencia:assignar().
{ok,11}
4> exit(self(),kill).
Llista freq assignades
{[12,13,14,15],[{11,<0.188.0>},{10,<0.188.0>}]}
** exception exit: killed
4> frequencia:assignar().
{ok,11}
5> frequencia:stop().
ok
```

Sistemes robustos: Exemple -> Supervisor

- Els supervisors són processos que només han d'iniciar els fills i monitoritzar-los.
- Com s'aplica a la pràctica?
  - Els fills poden iniciar-se en la fase d'inici del supervisor, o dinàmicament, quan el supervisor està funcionant.
  - Els supervisors atrapen les sortides i els enllaços dels seus fills quan cauen.
  - Si un procés fill finalitza, el supervisor ha de rebre el senyal de sortida.
  - El supervisor pot utilitzar el Pid del fill del senyal de sortida per identificar el procés i reiniciar-lo.
- Els supervisors han de gestionar les finalitzacions dels processos i els reinicis de manera uniforme, i han de prendre les decisions sobre les accions que cal fer. Aquestes accions poden incloure:
  - no fer res,
  - reiniciar el procés,
  - reiniciar el subarbre complet,
  - o acabar,
- Tots els supervisors s'han de comportar d'una manera similar, amb independència del que fa el sistema. Juntament amb els clients/servidors, màquines d'estats finits i controladors d'esdeveniments, es considera que és un patró de disseny de procés:
  - La part genèrica del supervisor inicia als fills, els supervisa, i els reinicia en cas d'una finalització.
  - La part específica del supervisor consisteix en els fills, incloent quan i com s'inicia i es reinicia.

Sistemes robustos: Exemple -> Supervisor

```
-module(my_supervisor).
-export([start_link/2, stop/1]).
-export([init/1]).

start_link(Name, ChildSpecList) ->
    register(Name, spawn_link(my_supervisor, init, [ChildSpecList])),
    ok.

init(ChildSpecList) ->
    process_flag(trap_exit, true),
    loop(start_children(ChildSpecList)).

start_children([]) -> [];
start_children([{M, F, A} | ChildSpecList]) ->
    case (catch apply(M,F,A)) of
        {ok, Pid} ->
            [{Pid, {M,F,A}}|start_children(ChildSpecList)];
        _ ->
            start_children(ChildSpecList)
    end.
```

```
list_children(Name) ->
    Name!list.
```

Sistemes robustos: Exemple -> Supervisor

```
restart_child(Pid, ChildList) ->
    {value, {Pid, {M,F,A}}} = lists:keysearch(Pid, 1, ChildList),
    {ok, NewPid} = apply(M,F,A),
    [{NewPid, {M,F,A}}|lists:keydelete(Pid,1,ChildList)].

loop(ChildList) ->
    receive
        {add, [{M,F,A}]} ->
            {ok, NewPid} = apply(M,F,A),
            loop([{NewPid, {M,F,A}}|ChildList]);
        list ->
            io:format('Fills .=~n'),
            llist_children(ChildList),
            loop(ChildList);
        {'EXIT', Pid, _Reason} ->
            NewChildList = restart_child(Pid, ChildList),
            loop(NewChildList);
        {stop, From}  ->
            From ! {reply, terminate(ChildList)}
    end.
```

Sistemes robustos: Exemple -> Supervisor

```
1> c(my_supervisor).
{ok,my_supervisor}
2> my_supervisor:start_link(super, [{add_three, start, []}]).
ok
3> my_supervisor:add_children(super, [{add_two, start, []}]).
{add,[{add_two,start,[]}]}
4> whereis(add_three).
<0.426.0>
5> whereis(add_two).
<0.428.0>
6> exit(whereis(add_two), kill).
true
7> my_supervisor:list_children(super).
Fills .=
list
{<0.432.0>,{add_two,start,[]}}
{<0.426.0>,{add_three,start,[]}}
8> my_supervisor:stop(super).
ok
```

```
stop(Name) ->
    Name ! {stop, self()},
    receive
        {reply, Reply} ->
            Reply
    end.
```

```
terminate([{Pid, _} | ChildList]) ->
    exit(Pid, kill),
    terminate(ChildList);
terminate(_ChildList) ->  ok.

llist_children([]) -> ok;
llist_children([Child|ChildList]) ->
    io:format('~p~n',[Child]),
    llist_children(ChildList).
```