An Introduction to Elang

Joe Armstrong <erlang@gmail.com>

May 16, 2018

Contents

1 To whet your appetite

By the end of this course I hope you'll be able to *read* Erlang programs - and you'll have started to be able to *write* programs.

To start you off I'm going to show you a small program that by the end of the day you should be able to understand. I'll show it in a moment, but before this we have to talk about patterns.

1.1 Patterns

Patterns are central to Erlang. We'll meet them in many contexts.

In expressions:

```
Pattern = Value
```

In function definitions:

```
funcName(Pattern1) -> Actions1;
funcName(Pattern2) -> Actions1;
...
funcName(PatternN) -> ActionsN.
```

In `case` statements:

```
case Value of
   Pattern1 -> Actions1;
   Pattern2 -> Actions1;
   ...
   PatternN -> ActionsN
end
```

In funs (or closures):

```
F = fun(Pattern1) -> Actions1;
       (Pattern2) -> Actions1;
       ...
       (PatternN) -> ActionsN
    end,
```

In message reception:

```
receive
   Pattern1 -> Actions1;
   Pattern2 -> Actions1;
   ...
   PatternN -> ActionsN
end
```

1.2 The Multi Server

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

start() -> spawn(fun() -> loop() end).

loop() ->
    receive
        {_Pid, {email, _From, _Subject, _Text} = Email} ->
            io:format("multi_server:email:~p~n",[Email]),
            {ok, S} = file:open("inbox", [write,append]),
            io:format(S, "~p.~n", [Email]),
            file:close(S);
        {_Pid, {im, From, Text}} ->
            io:format("Msg (~s): ~s~n",[From, Text]);
        {Pid, {get, File}} ->
            io:format("multi_server:get:~s~n",[File]),
            Pid ! {self(), file:read_file(File)};
        Any ->
            io:format("multi server got:~p~n",[Any])
```

```
    end,
    loop().
```

This program is:

- An email server - (SMTP becomes an erlang message)
- An instant messgaging agent (jabber?, XMPP)
- A web server (or FTP server, or both) (HTTP, ...)

2 Getting started

- Starting the shell
- Entering commands
- Stopping the shell
- Editing commands
- Data types
- Variables
- Pattern matching

2.1 Starting the shell

- Unix-based systems: `$ erl`
- Windows: `Programs -> OTP ... -> Erlang`

This is what you see

```
$ erl
Erlang (BEAM) emulator version 5.6 [source] [smp:4] [async-threads:0]
              [hipe] [kernel-poll:false]

Eshell V5.6  (abort with ^G)
1>
```

2.2 Shell commands

- Shell is `read-eval-print` loop
- Commands end with a period (.) followed by whitespace

```
1> 3 * 7.
21
2> math:sqrt(2).
1.41421
3> "hello". "world".
"hello"
4> "world".
"world"
```

2.3 Shell commands

- Repeated prompt means command is not yet finished

```
5> 123 +
5> 3456 *
5> 789.
2726907
6>
```

2.4 Stopping the shell

- (ctrl) + C

```
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
a
```

- (ctrl) + – immediate exit
- init:stop(). – controlled exit
- erlang:halt() – uncontrolled exit

2.5 Recall and edit commands in the shell

- Arrow keys
- Emacs editor commands
  - Ctrl + B – back
  - Ctrl + F – forward
  - Ctrl + P – previous
  - Ctrl + N – next
- Tab – try to expand module or function names

2.6 Exercise

Try it out. Fire up erlang at a command prompt. Do some basic arithmetic.

```
$ erl
1> 123456789 * 987654321 * 123456789.
150534111114874476388912 41
...
```

2.7 The shell

- Major caveat: you cannot define functions in the shell
- Instead, create in separate files and compile in the shell

3 Variables

- Variables start with an upper case letter
- syntax `[A..Z][a..zA..Z0..9]` (almost)

```
1> MinutesPerHour = 60.
60
2> HoursPerDay = 24.
24
3> MinutesPerDay = MinutesPerHour * HoursPerDay.
1440
4>
```

3.1 Variables cannot vary

```
4> Count = 1.
1
5> Count = Count + 1.
** exception error: no match of right hand side value 2
```

- These are not like Java or C# variables
- More like variables in math

3.2 Back to high school

```
x + 3y = 15
x - 3y = 3

x = ?, y = ?
```

`x` and `y` mean the same thing in all equations.

3.3 Single assignment variables

- As in math, `X = X + 1` is illegal
- Enables concurrency
- No locks needed since we can't share and update data
- Makes debugging easy (because variable is only set once)
- Lives as long as it is within scope

4 Pattern matching

- `Pattern = Value` matchs Pattern against Value
- `X = 10` means "match the pattern `X` against the value `10`"

4.1 Var = value

- if `Var` is a "fresh" (new , unbound) variable, then the pattern is matched by giving `Var` the value `Value`
- if `Var` already has a value, pattern matches if its value equals `Value`, fails otherwise

```
1> X = 10.
10
2> X = 10.  % matches
10
3> X = 2*4 + 2.  % matches
10
4> X = 11.
** exception error: no match of right hand side value 11
```

5 Data types

Simple data types:

- Integers
  - `1 45 123123 2#101012 8#0723 16#face`
- Floats
  - `12.34e-07 3.14159`
- Atoms
  - `true false hello_world 'funny atom #$%!!'`
  - atoms are like enumerated types in C
  - (`monday`, `tuesday`, `wednesday`, ... are atoms – these can be used to represent days of the week)

5.1 Complex types - Tuples

- Ordered list of values `{food, egg}`
- can be nested `{person,{firstname,"joe"},{lastname,"armstrong"}}`

5.2 Creating and unpacking a tuple

- Constructed using `{ ... }` notation
- Unpacked by pattern matching

```
4> Person = {person, "Joe", "Armstrong"}.
{person, "Joe", "Armstrong"}
5> {Type, First, Last} = Person.
{person, "Joe", "Armstrong"}
6> Type.
person
7> Last.
"Armstrong"
```

5.3 Repeated occurrence of variables

If a variable occurs more than once in a pattern, it must have the same value.

```
1> {X, Y, X} = {1, a, 1}.
{1,a,1}
2> X.
1
3> Y.
a
4> {A, B, A} = {1, a, 2}.
** exception error: no match of right hand side value {1,a,2}
```

5.4 Anonymous variables

_ is a "don't care" variable. Used as a placeholder in pattern matches.

```
1> {X, _, X} = {1, a, 1}.
{1,a,1}
2> X.
```

5.5 Complex types - Lists

- Sequence of values (can be different types)
- `[]` is the empty list
- `[X1,X2,...,Xn]` is a list with elements `X1`, `X2`, ..., `Xn`
- `[...]` in a pattern deconstructs the list

```
1> X = [1, 2, a, b].
[1,2,a,b]
2> [A, B, C, D] = X.
[1,2,a,b]
3> A.
1
4> D.
b
```

5.6 Lists as recursive structures

- First element of a list is the *head*, what remains is a list called the *tail*. (Lisp programmers, think CAR and CDR)
- `[ ]` is the empty list
- `[ H | T ]` is the list with head `H` and tail `T` (which must be a list)
- Can be used to create and to pattern match

```
1> X = [1,2,3,4].
[1,2,3,4]
2> Y = [abc|X].
[abc,1,2,3,4]
3> [X1,_,X2|X4] = Y
[abc,1,2,3,4]
4> X3.
2
5> X4.
[3,4]
```

5.7 Strings

- `$a` is short for `97` (The ASCII code for the character `a`)
- `"abc"` is shorthand for `[$a, $b, $c]`
- The shell prints lists of integers as strings (if it can)
- `++` concatenates lists

```
1> Name = "Armstrong".
"Armstrong"
2> "Joe " ++ Name.
"Joe Armstrong"
3> [ 99, 97, 110].
"can"
```

5.8 Quiz: If strings are lists...

```
1> A = "Hi,".
"Hi,"
2> B = "there".
"there"
3> length([A, B]).
?
4> length(A ++ B).
?
5> length([A | B]).
?
```

5.9 Complex types - Maps

```
1> Person = #{first => "Joe", last => "Armstrong"}.
#{first => "Joe",last => "Armstrong"}
2> maps:get(first, Person).
"Joe"
3> maps:get(age, Person).
** exception error: {badkey,age}
     in function  maps:get/2
```

```
        called as maps:get(age,#{first => "Joe",last => "Armstrong"})

4> maps:find(age,Person).
error
5> maps:find(first,Person).
{ok,"Joe"}
```

You can pattern match on maps:

```
birthday(#{age := N} = Person) ->
    Person#{arg => N+1}.
```

5.10 Exercise

- Try to predict the result of each of the following *before* typing it into the shell. Use the command `f().` after each command to *forget* any bindings. `b()` prints all bindings.

```
X = true.
{X,abc} = {123,abc}.
{X,Y,Z} = {222,def,"cat"}.
{X,Y} = {333,ghi,"cat"}
{X,Y,X} = {{abc,12},42,{abc,12}}.
{X,Y,X} = {{abc,12},42,true}.
[H|T] = [1,2,3,4,5].
[H|T] = "cat".
[A,B,C|T] = [a,b,c,d,e,f].
```

5.11 Shell commands

```
> help().
 b()          -- display all variable bindings
 e(N)         -- repeat the expression in query <N>
 f()          -- forget all variable bindings
 f(X)         -- forget the binding of variable X
 h()          -- history
 history(N)   -- set how many previous commands to keep
 results(N)   -- set how many previous command results to keep
 v(N)         -- use the value of query <N>
```

... many more commands ...

6 Compiling and testing

- Step 1 - Write a unit test
- Step 2 - Write some code
- Step 3 - Iterate 1 and 2

(In most of the exercises I'll give you the unit tests.)

6.1 Write the unit test

```
-module(math3a).
-compile(export_all).

test() ->
    10 = sum([1,2,3,4]).
```

But...

```
1> c(math3a).
./math3a.erl:5: function sum/1 undefined
error
```

6.2 Correct the code and run

```
-module(math3b).
-compile(export_all).

test() ->
    10 = sum([1,2,3,4]).

sum([H|T]) -> H + sum(T).
```

```
1> c(math3b).
{ok,math3b}
2> math3b:test().
```

```
** exception error: no function clause matching math3b:sum([ ])
     in function  math3b:sum/1
     in call from math3b:test/0
```

6.3 Try again

```
-module(math3c).
-compile(export_all).

test() ->
    10 = sum([1,2,3,4]),
    -5 = sum([-6,1]),
    0 = sum([]),
    horray.

sum([H|T]) -> H + sum(T);
sum([]) -> 0.
```

```
7> c(math3c).
{ok,math3c}
8> math3c:test().
horray
```

6.4 Anatomy of a function

```
%% File name is amod.erl
-module(amod).
%% these can be called from outside the module
-export([foo/1, bar/2]).
%% import this function from lists
-import(lists, [reverse/1]).

foo(L) ->
    L1 = lists:flatten(L),   %% call 'flatten' which must be exported from lists
    reverse(L1).             %% same as lists:reverse(L1)

bar(A, B) -> A + B.
```

6.5 Useful Trick when developing

`-compile(export_all)` exports all functions from this module.

6.6 Running your program

- Compile it `$ erlc mycode`
- Run it `$ erl -noshell -pa /path/to/code -s Mod Func Args ...`

6.7 Or use escript

```
#!/opt/local/bin/escript

main(List) ->
    io:format("Args are ~p~n", [List]).
```

Then, from the command line

```
% chmod +x myprog
% ./myprog 34 67
Args are ["34","67"]
```

6.8 Code paths etc.

- Erlang autoloads compiled modules using the current code search paths
- For libraries, can set the path in `./.erlang` or `${HOME}/.erlang`
- both are read by `erl` when it starts
- Can contain any `erl` commands

```
code:add_patha("/Users/joe/Desktop/work/2007/jaerlang").
code:add_patha("/Users/joe/Desktop/work/2007/jaerlang/socket_dist").
```

6.9 Exercise: During an idle moment...

Try adding `io:format("Hello from:~p~n",[file:get_cwd()]).` to either `/.erlang` or `${HOME}/.erlang`.

7 Getting help

Manual pages are not installed by default (but MacPorts does load them for you)

- Unix – `wget http://www.erlang.org/download/otp_doc_man_R11B-5.tar.gz`
  - Unpack so that the `man` directory is unpacked at the root of the Erlang distribution (usually `/usr/local/lib/erlang`).
  - View using `erl -man Modulename`
- Windows – the HTML rendered documentation is included in the distribution

8 Sequential erlang

8.1 Review: Types

Sequential Erlang programs manipulate instances of data types

- `integers` – `2176537165` `1` `16#face`
- `floats` – `1.24234`
- `atoms` – `monday`, `tuesday`
- `binaries` – `<<"3868GAJSJB">>` memory buffers

With two complex data types

- `lists` – `[X1, X2, ...]`
- `tuples` – `{X1, X2, ...}`

8.2 Functions

Remember:

```
func(Pattern1) -> Actions1;
func(Pattern2) -> Actions2;
...
func(PatternN) -> ActionsN.
```

Get the punctuation right - SEMI-COLON ... DOT.

```
temp_convert({f, F}) -> {c, 5*(F-32)/9};
temp_convert({c, C}) -> {f, 32 + 9*C/5}.

lookup(Key, [{Key,Val}|_]) -> {yes, Val};
lookup(Key, [_|T]) -> lookup(Key, T);
lookup(Key, []) -> no.
```

8.3 Built-In functions (BIFs)

- We can't do everthing with pattern matching For example, type conversion BIFS, convert an atom to a list, convert a tuple to a list etc.
- `atom_to_list(Atom)`, `tuple_to_list(Tuple)`, `term_to_binary(term)`, `binary_to_term(Bin)`
- `erl -man erlang` describes all BIFS (book pages 451–460 contains a summary)

8.4 BIFs For type conversion

```
gen_tcp:send(Socket, term_to_binary(Term)), ...

receive
   {tcp_data, Socket, Bin} ->
        Term = binary_to_term(Bin)
   ...
end.
```

8.5 BIFS for efficiency

```
encode(Term) ->
   B = term_to_binary(Term),
   Md5 = erlang:md5(B),
   <<B/binary,Md5/binary>>.

decode(<<Md5:16/binary,B/binary>>) ->
   Md5 = erlang:md5(B), %% throws an error if wrong value
   binary_to_term(B).
```

8.6 BIFS

- Quirky syntax
- Sometimes `tuple_to_list`, sometimes `erlang:now()`
- In theory the `erlang:` BIFS are implement in Erlang (not true :-)
- Behave as if they were defined in the module `erlang`

8.7 apply

You can build a call to a function and apply that function at run-time using `apply(Mod, Func, [Arg1, Arg2, ...])`.

```
1> M = list_to_atom("erlang").
erlang
2> F = list_to_atom("tuple_to_list").
tuple_to_list
3> apply(M, F, [{a,b,c}]).
[a,b,c]
4> M:F({a,b,c}).
[a,b,c]
```

8.8 Funs

Funs are "anonymous functions" (also called lambda-expressions, closures).

```
1 > Double = fun(X) -> 2*X end.
#Fun<erl_eval.4.4564646>
2 > Double(2).
4
```

Funs can be arguments to functions.

```
3 > lists:map(Double, [1,2,3,4]).
[2,4,6,8]
```

8.9 Let's add a for loop to Erlang

```
-module(hofs).
-export([for/3]).

for(Max, Max, F) -> [ F(Max) ];
for(I, Max, F)   -> [ F(I) | for(I+1,Max,F) ].
```

```
1> c(hofs).
{ok,hofs}
2> hofs:for(1,10,fun(I) -> 2*I end).
[2,4,6,8,10,12,14,16,18,20]
```

8.10 Funs can return funs

```
1> MakeAdder = fun(Inc) -> fun(X) -> X + Inc end end.
#Fun<erl_eval.6.72228031>
2> Add5 = MakeAdder(5).
#Fun<erl_eval.6.72228031>
3> Add5(10).
15
```

8.11 map

- apply the same function to every element of a list. The result is a list the same length as the input list
- `map(F, [X1,X2,...Xn]) -> [F(X1), F(X2), ..., F(Xn)]`

- *Incredibly useful*

```
1 > lists:map(fun(X) -> X * X end, [1,2,3]).
[1,4,9]
```

8.12 Writing our own map

```
my_double([ H | T ]) -> [ 2*H | my_double(T) ];
my_double([]) -> [].
```

8.13 Writing our own map

```
my_double([ H | T ]) -> [ 2*H | my_double(T) ];
my_double([]) -> [].
```

Generalize:

```
my_map(Fun, [ H | T ]) -> [ Fun(H) | my_map(Fun, T) ];
my_map(Fun, [])  -> [].

my_double(L) -> my_map(fun(X) -> 2*X end, L).
```

8.14 Accumulators

Let's write `sum` using an accumulator:

```
sum(L) -> sum(L, 0).

sum([H|T], Sum) ->
    Sum1 = Sum + H,
    sum(T, Sum1);
sum([], Sum) ->
    Sum.
```

Note we have two different functions, `sum/1` and `sum2/2`. This is a common pattern–the additional parameters are analogous to function-local variables in other languages.

8.15 List comprehensions

```
[ Constructor || Pattern <- List, Predicate, ...]
```

- Come from set theory–the set of all `X` such that `Y`, eg `{ x : x is even }`
- In Erlang, use `||`

```
2> L = lists:seq(1,20).
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
3> [ X || X <- L, (X rem 2) =:= 0].
[2,4,6,8,10,12,14,16,18,20]
```

  * Allows very compact code
  `map(F, L) -> [F(X) || X <- L].`

8.16 Quicksort

```
-module(qsort).
-export([sort/1]).

sort([Pivot|T]) ->
    sort([ X || X <- T, X < Pivot ])
++ [Pivot] ++
    sort([ X || X <- T, X >= Pivot ]);
sort([]) ->
    [].

1> c(qsort).
{ok,qsort}
2> qsort:sort([1,a,3,4,b,c,2]).
[1,2,3,4,a,b,c]
```

8.17 Multiple return values

- Single return value `X = some_func(....)`
- To get multiple return values use a tuple
  `{X, Y, Z} = another_func(....)`

8.18 Accumulators

- Pass extra argument(s) into the function

```
-module(accum).
-export([evens_and_odds/1]).
-import(lists, [reverse/1]).

evens_and_odds(L) -> evens_and_odds(L,[], []).

evens_and_odds([H|T], E, O) when H rem 2 =:= 0 -> evens_and_odds(T, [H|E], O);
evens_and_odds([H|T], E, O) ->  evens_and_odds(T, E, [H|O]);
evens_and_odds([], E, O)    ->  {E, O}.
```

```
1> accum:evens_and_odds([1,2,3,4,5,6,7]).
{[6,4,2],[7,5,3,1]}
```

8.19 Accumulators

- User `reverse` to fix order at end

```
-module(accum2).
-export([evens_and_odds/1]).
-import(lists, [reverse/1]).

evens_and_odds(L) -> evens_and_odds(L,[], []).

evens_and_odds([H|T], E, O) when H rem 2 =:= 0 -> evens_and_odds(T, [H|E], O);
evens_and_odds([H|T], E, O) ->  evens_and_odds(T, E, [H|O]);
evens_and_odds([], E, O)    ->  {reverse(E), reverse(O)}.
```

```
2> accum:evens_and_odds([1,2,3,4,5,6,7]).
{[2,4,6],[1,3,5,7]}
```

8.20 Guards

Extend function pattern matching to include additional predicates and comparisons

```
func(A, B)  when  ...guard...  ->
```

For example:

```
func1(A, B) when is_integer(A) ->
    ...
```

8.21 Single Guard conditions

- Type-match predicates (`is_integer(X)`, `is_tuple(X)`, ...)
- Built-in guard functions (`length(X)`, `hd(X)`, ...)
- comparisons, boolean, and arithmetic expressions

```
func1(A, B) when is_integer(A) andalso A + 1 > B ->
    ...
```

8.22 Full Guard

- A comma separated list of single guard conditions
- Matches only if all match

```
func1(A, B) when is_integer(A), is_tuple(B) ->
    ...
```

8.23 Guard Sequence

- A semicolon separated list of full guards
- matches if any match

```
func1(A, B) when is_integer(A), is_tuple(B); is_float(A), is_tuple(B) ->
    ...
```

Or

```
classify(Day) when A =:= saturday; A =:= sunday ->
    weekend;
classify(Day) ->
    weekday.
```

8.24 case and if

```
goto_work(Day) ->
    case classify(Day) of
        weekday -> true;
        weekend -> false
    end.
```

```
goto_work(Day) ->
   if
      Day =:= sunday -> false;
      Day =:= saturday -> false;
      true             -> true
   end.
```

Remember:

```
case Value of
    Pattern1 -> Actions1;
    Pattern2 -> Actions2;
    ...
    PatternN -> ActionsN
end
```

Get punctuation right SEMI-COLONS.

8.25 Booleans

- `true` and `false` atoms (not built-in type)
- By convention only, but a strongly recommended convention.

8.26 Many functions work with booleans

- `lists:filter(Fun, L) -> L'`

  This takes a list of values `[X1,X2,...,Xn]` and produces a new list `[Xi,Xj,...]` of all the vlaues in `L` for which `Fun(X)` is `true`.

  `lists:partition(Fun, L) -> {T, F}`

  Splits the elements `X` in `L` into two sub-lists `T` and `F` depending on whether `Fun(X)` is `true` or `false`.

These functions will not work on things like `on` and `off`.

8.27 Tame large tuples with records

`{person,"joe","armstrong"}` is fine when there are just a few elements but when we have more than (say) half a dozen elements things get messy.

```
-record(person, {name,firstname,lastname,age,sex,weight,height}).

X = #person{name="jane", age=32, sex=female} %% creates a new record

birthday(#person{age=N} = X) ->
    X#person{age = N+1}
```

Records both construct and pattern match.

8.28 Record defaults

```
-record(window, {width=100, ht=100, x=10, y=10, color=red}).

X = #window{width=200}, %% other arguments get default values
```

8.29 Not mentioned

- macros
- parse transforms

8.30 Exercises:

Write a program to number the elements in a list. Here's the test case:

```
test() ->
  [{1,a},{2,b},{3,c}] = number_list([a,b,c]),
```

Write functions `one(List)`, `two(List)`, and `three(List)` that each returns two lists containing the even and odd elements in `List`. Use three different techniques. The test case is:

```
test() ->
    {[2,4],[1,3,5]} = one([1,2,3,4,5]),
    {[2,4],[1,3,5]} = two([1,2,3,4,5]),
    {[2,4],[1,3,5]} = three([1,2,3,4,5]),
    hooray.
```

9 Exceptions

- Type errors occur when calling a BIF with the wrong type
- Pattern matching errors
- Explicit errors caused by program executing `throw`, `exit`, `error`
- exceptions can be *converted* into values with `catch` and `try ... catch ... after ... end`

9.1 Convert exceptions into values

```
1> X = atom_to_list(123).
** exited: {badarg,[{erlang,atom_to_list,[1234]},
                    {erl_eval,do_apply,5},
                    {erl_eval,expr,5},
                    {shell,exprs,6},
                    {shell,eval_loop,3}]} **
2> X.
** 1: variable 'X' is unbound **

3> X = (catch atom_to_list(123)).
{'EXIT',{badarg,
  [{erlang,atom_to_list,[1234]},
   {erl_eval,do_apply,5},
   {erl_eval,expr,5},
   {erl_eval,expr,5},
   {shell,exprs,6},
   {shell,eval_loop,3}]}}
4> X.
{'EXIT',{badarg,
  [{erlang,atom_to_list,[1234]},
   {erl_eval,do_apply,5},
   {erl_eval,expr,5},
   {erl_eval,expr,5},
   {shell,exprs,6},
   {shell,eval_loop,3}]}}
```

9.2 Protecting a function call

`catch F(X)` returns `F(X)` if no exception is generated while evaluating the function

```
some_function(X, Y) ->
    case (catch something_that_might_crash(X, Y)) of
        {'EXIT', Why} ->
            %% .. handle the error ...;
        NormalReturn ->
            %% regular processing
    end.
```

Evaluating `some_function(X, Y)` will never raise an exception (assuming that we correctly handle the error in the `EXIT` branch of the `case` statement)

9.3 Protecting a sequence of expressions

```
start(Arg1, Arg2) ->
    try
       begin
           ok = start_server(),
           L1 = dothis(Arg1),
           ...
       end
    catch
       exit: Pattern ->
          ...
    after
        ....
    end
```

9.4 Use exceptions for "impossible" situations

I once found this code:

```
opcode(load) -> 1;
opcode(store) -> 2;
```

```
opcode(X) ->    %% invalid: what should I do now?
    io:format("bad opcode:~p~n",[X]).
```

What does this return for invalid opcodes?

9.5 Solutions

do something

```
opcode(load) -> 1;
opcode(store) -> 2;
opcode(X) -> exit({ebadOpCode, X}).
```

Solution two: *do nothing, be lazy* (best)

```
opcode(load) -> 1;
opcode(store) -> 2.
```

Program will generate an exception when called with an invalid argument *precisely as in solution two, the only difference is the name of the exception.*

9.6 exit, throw, or error

- use `exit` if this is so serious that the process should die
- use `thow` if you intend to catch the error in the program
- use `error` if you want to behave like a "normal" library error

9.7 Defensive programming is painful...

```
case file:open("config") of
    {ok, T} ->
        case T of
           {port,P,host,H} ->
              case open_socket(Host, Port) of
                  {ok, Pid} ->
                     ...
                  {error, Why} ->
```

```
                    ...
                end
            ...  ->
                ...
        end
    ...
```

9.8 So don't program defensively

- Use **ONE BIG CATCH AT THE TOP OF YOUR PROGRAM**
- write *everything else* as if it is going to work.
- let your code crash early

```
start(...) ->
    try
        begin
            {ok, T} = file:consult("config"),
            {port,Port,host,Host} = T,
            {ok, Pid} = open_socket(Host, Port)
        end;
    catch
      What:Why ->
          io:format("oooo err: ~p ~p~n",[What,Why])
    end
```

(We'll talk about inter-process errors shortly...)

9.9 Exercise:

The BIF `list_to_integer("1234")` will return `1234`, but exits with a bad argument exception if given a bad string. Write a function `list_to_int(String)` that returns `{ok, Int)` or `{error, eBadInt}`.

The unit test is:

```
test() ->
    {ok, 123} = list_to_int("123"),
    {error, eBadInt} = list_to_int("abc"),
    hooray.
```

9.10 Debugging

- There is a debugger–very few people use it
- Errors are pretty easy to find. This is a consequence of single assignment
- Compiler errors are "obvious"
- `lib_misc:dump(File, Term)` for large data structures
- `io:format("...~p....~n", [Var1, ...])`
- `?NYI` macro
- `ifdef(DEBUGGING)`

9.11 lib_misc:dump(File, Term)

```
1 > V = epp:parse_file("ex1_bit_syntax.erl","",""),true.
true
2> lib_misc:dump("debug", V).
Dumping term to debug
ok
```

`V` is stored in a file called `debug`

```
{ok,
 [{attribute,1,file,{"ex1_bit_syntax.erl",1}},
  {attribute,1,module,ex1_bit_syntax},
  {attribute,2,compile,export_all},
  {function,
   31,
   test,
   0,
   [{clause,
     31,
     [],
     [],
     [{match,
```

```
        32,
        {var,32,'Code'},
    ...
```

9.12 MACRO: Not Yet Implemented

```
-define(NYI(X),(begin
                    io:format("*** NYI ~p ~p ~p~n",[?MODULE, ?LINE, X]),
                    exit(nyi)
                end)).
```

Use like this:

```
    ...
    ?NYI({glurk, X, Y}).
    ...
```

9.13 MACRO: DEBUGGING

```
-define(DEBUGGING, true).
-ifdef(DEBUGGING).
-define(DEBUG(X), X).
-else.
-define(DEBUG(X), void).
-endif.

...
    ?DEBUG(io:format("Found:~p~n",[Files])),
...
```

10 Concurrency

- The world *is* concurrent
- Programming concurrent activities in a sequential language is *artificially difficult*
- Modern processors are concurrent

10.1 Models of concurrency

- Shared memory (locks, mutexes, coroutines, processes, threads, deadlock, livelock, failure, thread-safe)
- Message passing

10.2 Processes or threads?

- Threads *share* resources (essentially an efficiency hack – beware of premature optimizations)
- Processes do not share things. Processes are the basis of *security* in an operating system
- (Caution) the world "process" is thought of as a synonym for "slow, big, difficult to manage" Erlang processes are *very lightweight*. (Much lighter than a conventional thread)

10.3 Concurrency in other languages

- Can only create very small number of processes (a few thousand at most)
- Heavy weight
- Only supports message passing and not the kind of error handling semantics that Erlang has

10.4 Basic ideas

- If you know the name of a process, you can send it a message
- `Pid ! Message`
- If `Pid` is hidden you cannot send a message to the processes (security)
- Message arrives in a mailbox (like email)
- You never know if a message arrives (if you want to be sure send a message back)

- You can *link* to a process. If a process dies and you linked to it, you will be sent an error signal.
- The error signal is like a message (more later)

10.5 Everything is a process

In the Erlang world we model all non-Erlang things as processes.

We interact with all objects by sending them messages. The behavior of all objects in the external world is inferred from analysing the messages they send to us.

This is the *purest* form of OO (encapsulation, isolation, polymorphic).

10.6 Processes

- `spawn` – create a new process
- `send` – send a message to a process
- `receive` – receive a message

10.7 Spawn

Three syntaxes:

- `spawn(fun foo/0)` – runs `foo/0` in a new process
- `spawn(fun() -> ... end)` – spawns an inline fun
- `spawn(Mod, Func, [Arg1, Arg2, ..., Argn])` performs `apply(Mod, Fun, [Arg1, Arg2, ..., Argn])` in a new process.

In `Pid1` evaluate `Pid2 = spawn(fun() -> ...end)`

10.8 Send

`Pid ! Msg` sends the message `Msg` to the to the *Mailbox* of a process named `Pid`. `Pid` is the return value of of a previously evaluated `spawn` expression.

10.9 Receive

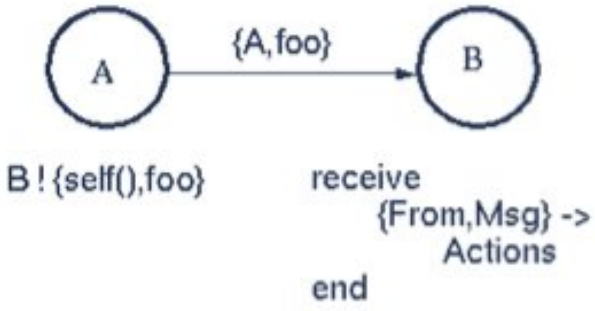
Receive suspends the process until a message arrrives in the mailbox that matches one of the patterns in a `receive` statement. (Rather like a fancy select statement)

Syntax:

```
receive
    Pattern1 -> Actions1;
    Pattern2 -> Actions2;
    ...
end
```

10.10 Including self() in a message

- `self()` is the pid of the current process
- including `self()` in a message allows the recipient to respond

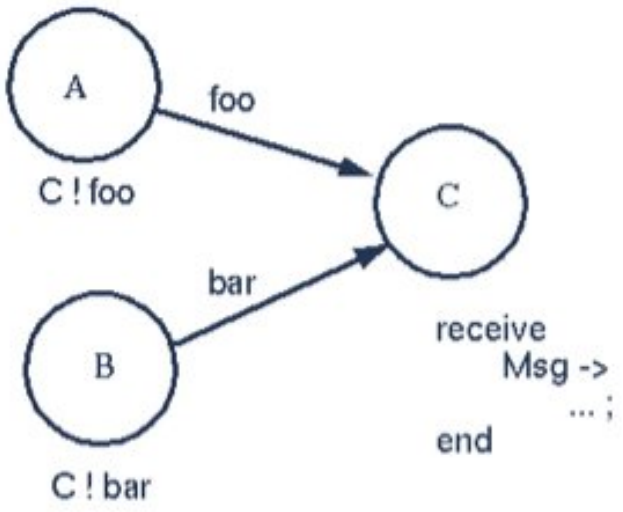


10.11 Spawn send and receive example

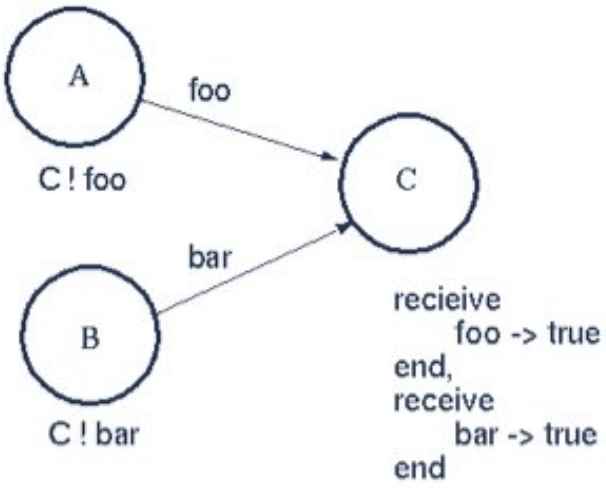
```
-module(counter1).
```

```
-export([start/0]).

start() ->
    Pid = spawn(fun() -> counter(1) end),
    tick(Pid),
    tick(Pid).

tick(Pid) -> Pid ! bump.

counter(N) ->
    io:format("Counter is now ~w~n", [ N ]),
    receive
        bump ->
            io:format("Got bumped~n"),
            counter(N+1)
    end.

1> c(counter1).
{ok,counter1}
2> counter1:start().
Counter is now 1
bump
Got bumped
Counter is now 2
Got bumped
Counter is now 3
```

10.12 Selected or ordered receives

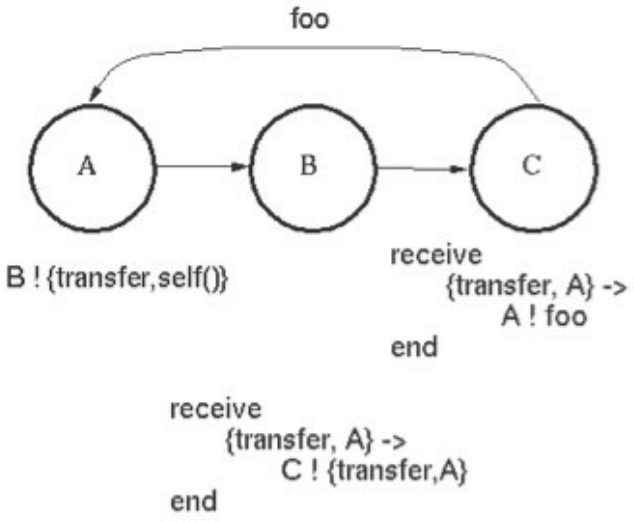


Receive message from A or B whichever comes first



Receive message from A then B

10.13 Sending Pids in messages



10.14 Counter processes

How do we get the value of the counter from the process?

```
-module(counter2).
-export([start/0, tick/1, read/1]).

start()   -> spawn(fun() -> counter(1) end).
tick(Pid) -> Pid ! bump.

read(Pid) ->
    Pid ! {self(), read},
    receive
        N ->  N
    end.

counter(N) ->
    receive
```

```
        bump         -> counter(N+1);
        {From, read} -> From ! N, counter(N)
    end.
```

```
1> c(counter2).
{ok,counter2}
2> P = counter2:start().
<0.38.0>
3> counter2:tick(P).
bump
4> counter2:tick(P).
bump
5> counter2:read(P).
3
```

Is anything wrong with this code?

10.15 Another process might reply

"Better" version – include a return value to match on that makes sure the expected process answers.

```
-module(counter3).
-export([start/0, tick/1, read/1]).

start()   -> spawn(fun() -> counter(1) end).
tick(Pid) -> Pid ! bump.

read(Pid) ->
    Pid ! {self(), read},
    receive
        {Pid, N} ->  N
    end.

counter(N) ->
    receive
        bump         -> counter(N+1);
```

```
        {From, read} -> From ! {self(), N}, counter(N)
    end.
```

10.16 Exercise

`multi_server`

Write `multi_server.erl`. `Pid = multi_server:start()` starts a multi server. It responds to three messages:

- `{Pid, {email, Who, Subject, Text}}` - a request to store email in your inbox
- `{Pid, {im, Who, Text}}` - an instant message - write message to terminal.
- `{Pid, {get, File}}` - a request to fetch a file. The server should reply by sending the file contents (as a message) back to `Pid`.

  In all of these `Pid` is the process identifier of the process sending a request to the server.

Hints:

- `{ok, Bin} = file:read_file(File)` reads `File` into a binary.
- `{ok, Stream} = file:open("inbox", [write,append])` opens a file for write append
- `io:format(Stream, "~p.~n",[Term])` writes a term to a file
- `file:close(Stream)` closes a file

10.17 Data abstraction (1)

Did we need to "invent" the functions `tick` and `read`?

```
2> c(counter4).
{ok,counter4}
3> Pid = spawn(fun() -> counter4:counter(0) end).
<0.42.0>
4> Pid ! bump.
```

```
bump
5> Pid ! bump.
bump
6> counter4:rpc(Pid, read).
2
```

```
-module(counter4).
-export([counter/1, rpc/2]).

rpc(Pid, What) ->
    Pid ! {self(), What},
    receive
        {Pid, N} ->  N
    end.

counter(N) ->
    receive
        bump         -> counter(N+1);
        {From, read} -> From ! {self(), N}, counter(N)
    end.
```

10.18 Data abstraction (2)

- Are we hiding the wrong thing? Should we expose the protocol?
- PROS: Hides the messaging structure. Makes "module resuability" easy. We know how to define APIs.
- CONS: More code. We don't know how to describe protocols. Protocols do not exist in Erlang (or in any other language for that matter) as first class objects.

10.19 Abstracting Server Functionality

```
-module(abs_server).
-export([start/1, cast/2, rpc/2]).

start(Data) ->
```

```
    spawn(fun() -> loop(Data) end).

cast(Pid, F) -> Pid ! {cast, F}.

rpc(Pid, F) ->
    Pid ! {rpc,self(), F},
    receive
        {Pid, Reply} -> Reply
    end.

loop(State) ->
    receive
        {cast, F} -> loop(F(State));
        {rpc, From, F} ->
            {Reply, State1} = F(State),
            From ! {self(), Reply},
            loop(State1)
    end.
```

```
-module(counter5).

-export([start/0, tick/1, read/1]).

start() -> abs_server:start(0).

tick(Pid) -> abs_server:cast(Pid, fun(X) -> X+1 end).
read(Pid) -> abs_server:rpc(Pid,  fun(X) -> {X, X} end).
```

```
> Pid1 = counter5:start().
<0.36.0>
> counter5:tick(Pid1).
{cast,#Fun<counter5.0.20406117>}
> counter5:tick(Pid1).
{cast,#Fun<counter5.0.20406117>}
> counter5:read(Pid1)
2
```

10.20 Registered processes.

Any program that wants to send a message to `Pid` must have `Pid` in a local variable. This must be communicated to all functions that want to send a messages to `Pid`

This is secure (but inconvenient)

- `register` – registers a global name with a `Pid`
- `unregister` * After `register(name, Pid)`, `name ! Term` sends a message to `Pid`
- `whereis(Name)` returns `Pid` or `undefined`

10.21 Client-Server Model

```
-module(reg_counter1).
-export([start/0, tick/0, read/0]).

start()   ->
    register(counter, spawn(fun() -> counter(1) end)).

tick() -> counter ! bump.

read() ->
    counter ! {self(), read},
    receive
        {counter, N} ->  N
    end.

counter(N) ->
    receive
        bump          -> counter(N+1);
        {From, read} -> From ! {counter, N}, counter(N)
    end.
```

10.22 Warnings

If two processes do `register` with the *same name at the same time* one will succeed and one will fail.

Suppose we evaluate the following in two parallel processes:

```
make_global(Name, F) ->
    register(Name, spawn(F)).
```

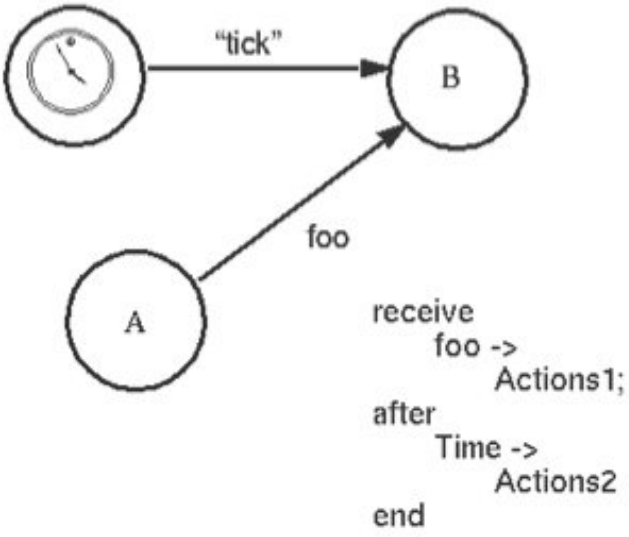
One will fail the other will succeed.

- Can you write a process `make_global(Name, Fun)` that is correct using only `register` and `whereis`?
- Think about it

10.23 Timeouts: receive ... after

```
receive
    Pattern1 -> ...
    Pattern2 -> ...
after
    TimeInMilliseconds ->
        Actions
end.
```

Timeouts



10.24 Exercise: Write an alarm process

- `alarm(Pid, Msg, Time)` sends the message `Msg` to `Pid` after `Time` milliseconds.
- Use it to write "Hello, World!" to your console every two seconds

10.25 Mutually recursive processes

Process one needs to know `Pid2` and processes two needs to know `Pid1`

```
Pid1 = spawn(fun() -> start(F1) end),
Pid2 = spawn(fun() -> start(F2) end),
Pid2 ! Pid1,
Pid1 ! Pid2,
...
start(F) ->
        receive
```

```
            Pid -> F(Pid)
        end
```

10.26 Spawn process, send the code later

```
Pid = spawn(fun() -> wait() end),
...
Pid ! {do, F}
...

wait() ->
    receive
       {do, F} ->
           F()
    end
```

10.27 SMP and processes

- Erlang can take advantage of multiple processors and cores
- Enable using `-smp` s n + options (only if compiled in)

10.28 Summary

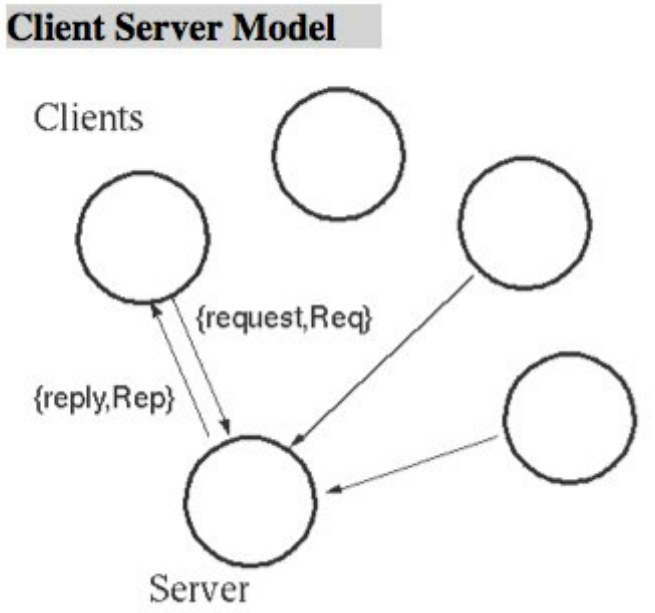
All concurrent behavior is programmed with

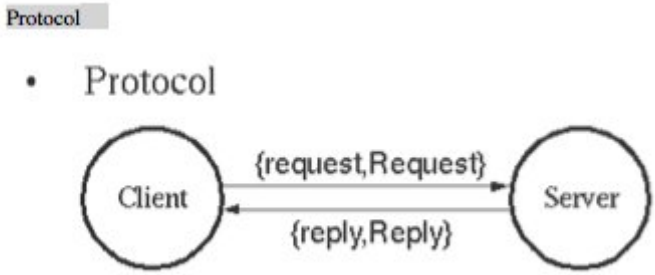
- `spawn`
- `send`
- `receive`

Instead of *blocks*, *mutexes*, *threads*, *processes*, *synchronised methods*, ...

10.29 Client-Server

The Most Important Concurrency Pattern






```
-module(s1).
-export([start/0, f1/2, f2/3, ..., stop/0]).

start() -> register(s1, spawn(fun() -> loop(State1))).

%% interface routines
f1(A, B) -> rpc(s1, {do_f1,A,B}).
```

```
f2(...)  -> rpc(s1, ...).

rpc(Q) ->
    s1 ! {self(), Q},
    receive
        {s1, Reply} -> Reply
    end.

loop(State) ->
     receive
         {From, {do_f1, A, B}} ->
              ...
              FRom ! {s1, Reply},
              loop(State1);
         ...
     end.
```

10.30 Variations: Exit client on server error

Exit the client if the RPC causes an error in the server

```
loop(State) ->
    receive
      ..
      {From, Q} ->
          try
             {Reply, State1} = f(Q, State),
             From ! {Name, ok, Reply},
             loop(State1);
          catch
             exit:Why ->
                From ! {Name, die, Why},
                loop(State)
          end

rpc(Name, Q) ->
    Name ! {self(), Q},
```

```
    receive
       {Name, ok, Reply} -> Reply;
       {Name, die, Why} -> exit(Why)
    end.
```

10.31 Variations: timeout in client

```
rpc(Name, Q) ->
    Name ! {self(), Q},
    receive
       {Name, ok, Reply} -> Reply;
       {Name, die, Why} -> exit(Why)
    after
       Time ->
           exit(timeout)
    end.
...
```

10.32 Variations: delegation

```
loop() ->
   receive
     {From, Q} ->
        ...
        Other ! {do, Q, replyAs, Name, replyTo, From}
```

Delegated

```
loop() ->
   receive
     {do, Q, replyAs, Name, replyTo, From} ->
        ...
        From ! {Name, Reply}
```

Responder

```
loop() ->
   receive
     {From, Q} ->
        ...
        From ! {Name, Reply}
```

Original

10.33 Variations: Mobile code

```
loop(State) ->
   ...
   {From, Func} ->
        {Reply, State1} = Func(State),
        From ! {self(), Reply},
        loop(State1);
   ...
```

10.34 Variations: Mobile code with transactions

```
loop(State) ->
   ...
   {From, Func} ->
        try Func(State) of
            {Reply, State1} ->
                From ! {self(), Reply},
                loop(State1);
        catch
           _:_ ->
                exit(From, evilLaugh),
                loop(State)
        end
   ...
```

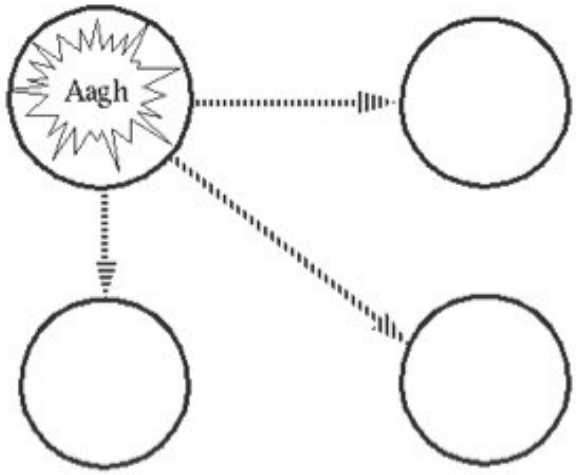
10.35 Rolling your own

- As you can see there are many variations of the client-server theme.
- No two ways of writing the client-server loop do exactly the same thing. These programs can be made to do exactly what you want.
- If you understand this then you can easily design your own middleware system

11 Links

- `link` – link to a process
- `unlink` – remove the link
- `process_flag(trap_exit, true)` – receive a signal as a message

11.1 What is a link?

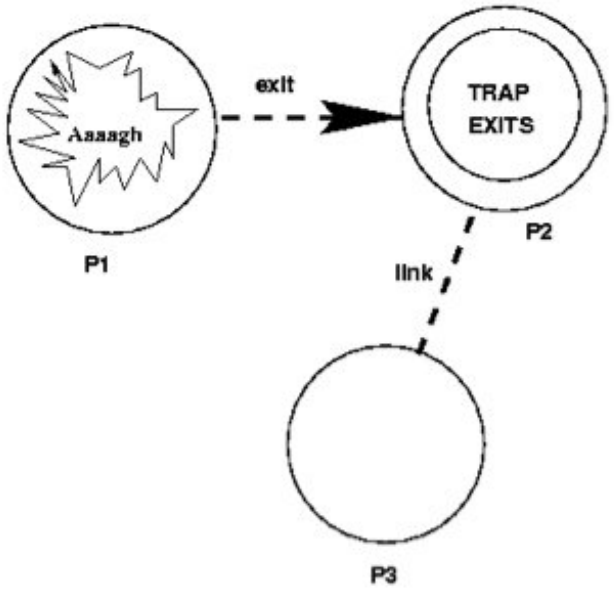


- `links` define error propagation paths

- If `A` is linked to `B` then: *If* `A` *fails then* `B` *will be notified If* `B` *fails then* `A` *will be notified*
- links are not "stacked" - calling `link` ten times has the same effect as calling it once.
- links propagate at the speed of messages - they are not instantaneous (important in distributed systems)

11.2 Messages and signals

- Messages are sent with `send`
- Signals are sent when processes die. Signals are not messages
- If `A` is linked to `B, C, ...` then `B,C,...` will be sent an exit signal if `A` dies
- All processes die when they receive signals, unless they are set to trap exits
- To trap exits a process evaluates `process_flag(trap_exit, true)`

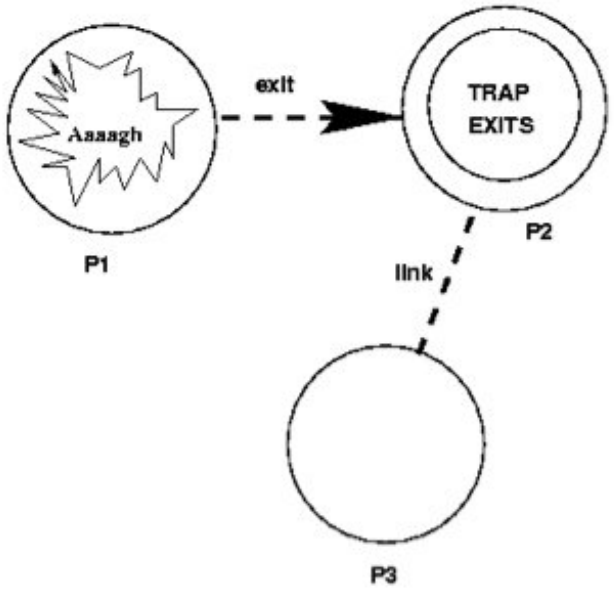


11.3 Exit messages

- `{'EXIT', Pid, Why}` can be received if you are trapping exits
- If you are *not* trapping exits and `Why` is not the atom `normal` you die

11.4 Example (1) - monitor

`monitor(Pid)` observer `Pid` and print a message if `Pid` dies.

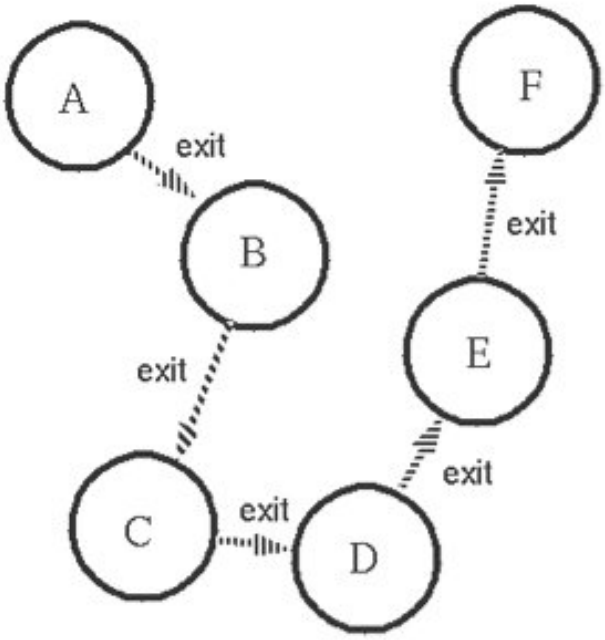


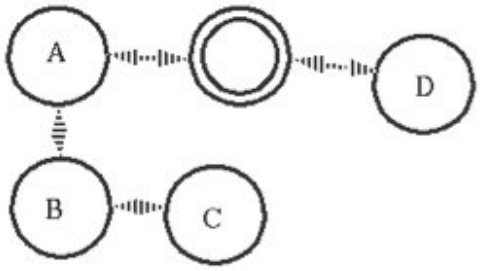

```
monitor(Pid) ->
    link(Pid),
    process_flag(trap_exit, true),
    receive
        {'EXIT',Pid,Why} ->
            io:format("Process died Reason:~p~n",[Pid,Why])
    end.
```

11.5 The small print

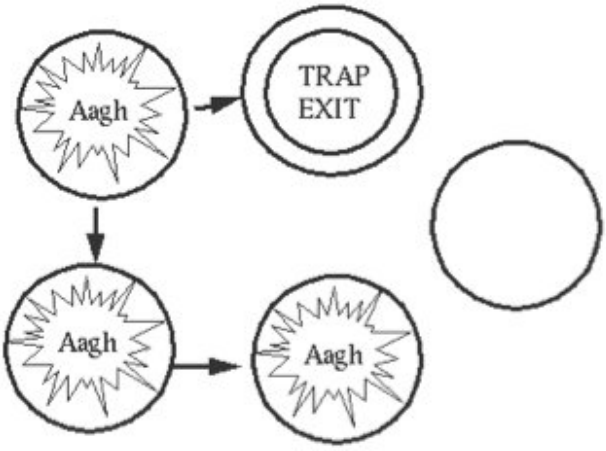
- You can fake an exit message by calling `exit(Pid1, Why)` – If you are `Pid2` then `Pid1` sees `{'EXIT',Pid2, Why}` and assumes you have died - but you are really alive. *Do not abuse.*
- exit reason `kill` is unstoppable.

11.6 Crashing and exit propagation





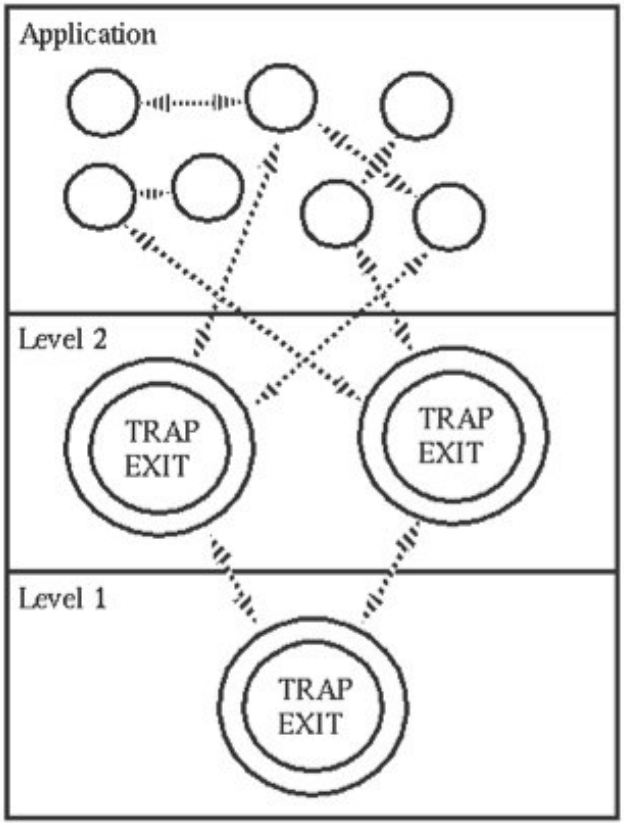
The process marked with a *double ring* is an error trapping process.



- Exits propagate through linked processes.
- If any of A..F crash all processes crash
- System processes stop the propagation of errors.
- Process D does not crash

11.7 Why do we do this?

We can build the system in layers to make it fault tolerant.



11.8 Things to think about

- If you do `spawn` followed by `link` the process might die very quickly (ie, before getting to the `link` statement)
- Solution: `spawn_link` is like `spawn` followed by `link` only the two are performed atomically
- Need to make sure `trap_exit` is evaluated *before* we try to catch errors – think about synchronization here

11.9 Exercise - Keep Alive

Write a function `make_global(Name, Fun/0)` that starts a "global" process named `Name`. If `Name` dies for any reason, restart it. Record the times and reasons for starting and restarting in an error log.

Useful stuff:

- `time() -> {Hour, Minute, Second}`
- `error_logger:format(String, List)` writes to the error log

11.10 The principle of remote error handling

- To make a fault-tolerant system you need two (separated) computers.
- Let one process do the work
- Let some other process fix the error
- You can think of exit signals as uncaught exceptions that escape from the process and propagate to some other process
- Processes that fail should die-early
- We can make very reliable systems this way
- We can test on one node and deploy on multiple nodes (everything works the same way)

11.11 half links (monitor)

- `erlang:monitor(process, Pid)`
- asymmetric
- countable (if a process is monitored N times it must be demonitored N times before the monitor is released - unlike link/unlink)

11.12 Summary

- `link` - create a link
- `unlink` - release the link
- `process_flag(trap_exit, true)` – trap exits
- `{'EXIT',Pid,Why}` then message sent when a process dies

All of this works in distributed Erlang. These mechanism are orthogonal to the `spawn`, `send` and `receive`

Now we've seen all that you need to make a powerful fault-tolerant system.

12 Sockets Based Distribution

12.1 Uses gen_tcp or gen_udp

Book `chapter 14`

12.2 Client

```
{ok, Socket} = gen_tcp:connect(Host, Port, [Options]),
ok = gen_tcp:send(Socket, Data),
...
receive
    {tcp, Socket, Data} ->
         %% do something with the data
         ...
    {tcp_closed, Socket} ->
         %% take care of this ...
```

12.3 Server

A "single-shot" server (accepts one connection)

```
start_server() ->
    {ok, Listen} = gen_tcp:listen(Port, [Options]),
    {ok, Socket} = gen_tcp:accept(Listen),
    %% We use same calls as the client to read and write
    %% the socket
```

12.4 A sequential server

```
{ok, Listen} = gen_tcp:listen(Port, ...)
seq_loop(Listen).

seq_loop(Listen) ->
    {ok, Socket} = gen_tcp:accept(Listen),
    loop(Socket),
    seq_loop(Listen).
```

12.5 A parallel server

Now turn this into a parallel server

```
{ok, Listen} = gen_tcp:listen(Port, ...)
spawn(fun() -> par_loop(Listen) end).

par_loop(Listen) ->
    {ok, Socket} = gen_tcp:accept(Listen),
    spawn(fun() -> par_loop(Listen) end),
    loop(Socket).
```

12.6 Packet lengths

A 4 byte length header is automatically added/removed by the system when calling `gen_tcp:send` and messages are assembled to the correct length before `{tcp, Sock, Data}` messages are send to the controlling process

```
gen_tcp:connect(Host, Port, [..., {packet,4}, ...])
gen_tcp:listen(Port, [..., {packet, 4}, ...])
```

12.7 Sending Erlang terms

```
gen_tcp:send(Socket, term_to_binary(Term)) ...

receive
        {tcp, Socket, Data} ->
              Term = binary_to_term(Data),
              ...
```

12.8 The middle man pattern

```
loop(Pid, Socket) ->
    receive
        {Pid, Msg} ->
            gen_tcp:send(Socket, term_to_binary(Msg)),
            loop(Pid, Socket);

        {tcp, Socket, Data} ->
            Pid ! binary_to_term(Data),
            loop(Pid, Socket);

        {'EXIT', Pid} ->
            gen_tcp:close(Socket);

        {tcp_closed, Socket} ->
            exit(Pid, socket_closed)
    end.
```

12.9 Exercise

Do this at home!

`multi_server` over sockets

Write `server.erl` and `client.erl`.

Use a middle man to connect a client on one machine to a server on another machine. Use `middle_man.erl` to connect the client to a socket on one machine and on the other machine to connect a process owning a socket to an instance of `multi_server`.

Hint: Book page 248 (and these notes) have the parallel server pattern. The middle man pattern is described in the course notes and on page 404 of the book in the description of `lib_chan_mm`

12.10 The Bit Syntax

Pack/Match bit strings from binaries

```
1> Red = 2, Green=61, Blue=20.
20
```

```
%% Red takes 5 bits, Green 6, and blue 5
2> Bin = <<Red:5, Green:6, Blue:5>>.
<<23,180>>
3> io:format("~8.2.0B ~8.2.0b~n", binary_to_list(Bin)).
   00010111 10110100
%% 00010 111101 10100
```

Size must be a multiple of 8 bits

12.11 Tips

- Can handle different endian words, big, little, ...
- complex syntax
- experiment in shell then cut and paste into program

12.12 Examples

- Book page 86 - find MP3 headers (for syncing with SHOUTCAST server)
- Too numerous to mention
- Fun just now. Decode/Encode AMF (Actionscript binary protocol)
- Useful for writing assemblers, protocol convertors etc.

13 Distributed Erlang

- Distributed Erlang – several `nodes` belong to the same system. `spawn` has an extra argument `spawn(Node, Mod, Fun, Args)`
- SMP Erlang – symmetric multiprocessing. Two or more identical processors are connected to a single shared main memory.
- Socket Distribution – Not really distributed at all

13.1 Distribution primitives

Adds three new primitives and a load of libraries

- `spawn(Node, Mod, Func, Args)` – *everything* works as before regarding `links` exit messages an so on.
- `alive(Node)` – tell the system you are alive

The main libraries

- `rcp` – doing remote procedure calls
- `global` – global operation over all nodes

13.2 Distributed Erlang

- Cookie based security
- Distribution over TCP (can be over secure sockets, but installation is more complex)
- Only suitable in a cluster
- Used in enterprise software behind a firewall

13.3 Distributed Erlang

The most common architectural pattern is to write traffic handling as non distributed applications on a single node. We use `mnesia` and the error loggers running replicated on multiple nodes.

`mnmesia` the (Erlang database) is can be configured to provide table replication over multiple nodes.

Fault tolerant applications usually use replicated pairs of nodes. One node does the work the other is a hot standby.

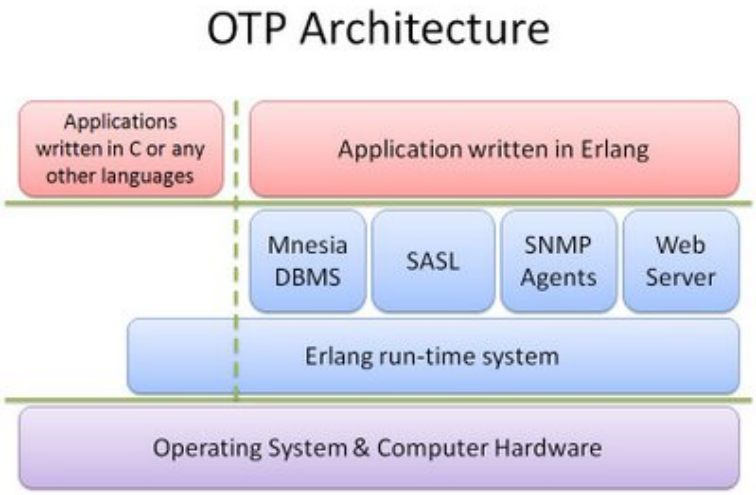
13.4 Two nodes, one machine

```
$ erl -sname one          %%  in one terminal window
(one@joe-armstrongs-computer)1>

$ erl -sname two        %% in another window
> (two@joe-armstrongs-computer)1> net_adm:ping('two@joe-armstrongs-computer').
pong
> (two@joe-armstrongs-computer)1> net_adm:ping('one@joe-armstrongs-computer').
pong
(two@joe-armstrongs-computer)2> node().
'two@joe-armstrongs-computer'
(two@joe-armstrongs-computer)3> rpc:call('one@joe-armstrongs-computer',
                                         erlang, node, []).
'one@joe-armstrongs-computer'
```

13.5 Shell can connect to remote nodes

```
$erl -name one
..
(one@joe-armstrongs-computer.local)1> ^G
User switch command
 --> j
   1* {shell,start,[init]}
 --> r 'server+doris.myerl.home.net'
 --> j
   1  {shell,start,[init]}
   2* {'server@doris.myerl.home.net',shell,start,[]}
 --> c 2
Eshell V5.5.5  (abort with ^G)
(server+doris.myerl.home.net)1> node().
'server@doris.myerl.home.net' ^G
User switch command
 --> c 1
(one+joe-armstrongs-computer.local)1> node().
'one@joe-armstrongs-computer.local'
```

14 OTP



- Open Telecomms Platform
- Maintained by product group inside Ericsson
- Releases 2-3 times/year
- "http://www.erlang.org/":http://www.erlang.org/
- "http://www.erlang.org/doc.html":http://www.erlang.org/doc.html

14.1 Structure

Generic structure

```
/appname/ebin  %% beam code
        /src   %% erlang source
        /priv  %% everything else
```

14.2 Principles

- "http://www.erlang.org/doc/design_principles/part_frame.html":http://www.erlang.org/doc
- Overview

- Client-server `gen_server`
- Finite State machines `gen_fsm`
- Event handling `gen_event`
- Supervisor `gen_sup`
- Applications
- Releases
- Application upgrade

14.3 Getting started with applications

- Read book `chapters 16 and 18` *especially* +16.1
- Read PhD thesis "http://www.sics.se/ joe/thesis":http://www.sics.se/ joe/thesis
- Read *design principles* "http://www.erlang.org/doc/design_principles/part_frame.html"
- Forthcoming O'Reilly book

14.4 behaviors

- The OTP name for "design patterns"
- Callback modules for client-servers, supervision trees etc.
- Encapsulates "best practice" from many individual projects
- 3'rd generation - ie the third rewrite of basic servers
- Used in practice

14.5 Case studies

See Armstrong PhD thesis.

- AXD301 – gen_server (122), gen_event (36), supervisor (20), gen_fsm (10), application (6).
- Nortel Networks – gen_server (56), supervisor (19), application (15), gen_event (9), rpc_server (2), gen_fsm (1), supervisor_bridge (1).

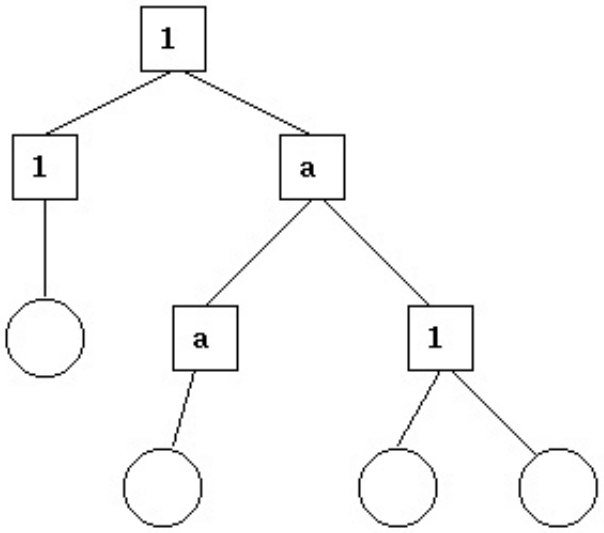
`gen_server` is the most used behavior

14.6 gen_server

- A generic client-server model
- `gen_server:start_link(Name, Mod, InitArgs, Opts)`
- `Mod:handle_call(...)` gets called for all RPCs
- `Mod:handle_cast(...)` gets called for all casts
- ...
- `Mod:code_change(...)` gets called when you want to change the code

More later...

14.7 gen_supervisor



Supervision Tree

- Supervision trees
- Hierarchical tree of workers and supervisors
- Supervisors monitor the workers
- If a worker fails this is noticed by the supervisor
- Supervisors can start, restart and kill workers
- 1:1 and 1:N supervision

15 Windup

15.1 Sequential Erlang

- Simple functional language
- Lists, tuples, atoms, bignums, floats, ...
- Function selection is by pattern matching

- Data selection is by pattern matching
- Variables are immutable

15.2 Concurrent Erlang

- Adds `spawn`, `send` and `receive` to sequential Erlang.
- `register` [`unregister`] can be used to associate a name with a process

15.3 Fault-tolerant Erlang

- `catch` .. `throw` and `try` ... `catch` ... `end` added to sequential Erlang
- `link`, `process_flag(trap_exit, true)` added to concurrent Erlang

15.4 Distributed Erlang

- Add +`spawn(Node, Mod, Func, Args)` to concurrent Erlang
- Or use explicit term passing over sockets

15.5 Benefits of Erlang

- Multi-core ready
- Processes in the language (not OS)
- Designed for fault-tolerant distributed programming
- Battle tested

15.6 Some Projects to research

- AXD301 (Ericsson)
- Kreditor (kreditor.se)
- SimpleDB (Amazon)
- CouchDB (text db)

- MociWeb (Mochimedia)
- Ejabberd (jabber server)
- ErlyWeb (Erlang on Rails :-)

15.7 And Remember...

Make It Fun!