Programming Erlang

Marcel Kyas

Freie Universität Berlin

Winter 09/10

The world is parallel.

If we want to write programs that behave as other objects behave in the real world, then these programs will have a concurrent structure.

Use a language that was designed for writing concurrent applications, and development becomes a lot easier.

Erlang programs model how we think and interact.

Joe Armstrong [1, p. i]

What Erlang is not

Erlang is not

- Imperative: there is no assignment that changes the state.
- Object-oriented: there are no classes, no inheritance and there is no late binding, not even methods
- Statically typed: There is no need to declare the type of variables or variables at all.
- Lazy: Erlang won't wait when evaluating expressions

What Erlang is

Erlang is

- Functional: Everything is a side-effect expression and produces a value.
- Concurrency-oriented: processes interact by asynchronous message passing.

Why learn Erlang?

- You want to write programs that run faster when executed on multi-core computers.
- You want to write fault-tolerant applications that can be modified without taking them out of service
- You have heard of “functional programming” and want to know whether the techniques really work.
- You want to use a language that has been battle tested in in real large-scale industrial products that has great libraries and an active user community.
- You don’t want to wear your fingers out by typing lots of lines of code.

Visit the website

The web page at `http://erlang.org` has a lot of documentation, tutorials, and other material you might find useful.

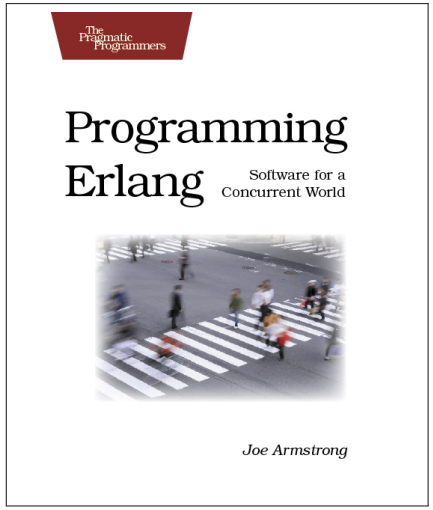
User groups

Ask people! But do not expect them to do your homework.
There is a Erlang User Group in Berlin:
`http://groups.google.com/group/erlang-berlin/`
They meet once a month (I have not been there yet).

Books

This lecture is mostly based on Joe Armstrong’s book.
There are other ones:

- Francesco Cesarini and Simon Thompson “Erlang Programming”
- Martin Logan, Eric Merrit and Robert Calco “Concurrent Programming with Erlang/OTP”



History of Erlang I

1982 – 85 Experiments with programming of Telecom using more than 20 different languages. Conclusion: The language must be a very high level symbolic language in order to achieve productivity gains! (Leaves us with: Lisp , Prolog , Parlog . . .)

1985 – 86 Experiments with Lisp, Prolog, Parlog etc. Conclusion: The language must contain primitives for concurrency and error recovery, and the execution model must not have back-tracking. (Rules out Lisp and Prolog.) It must also have a granularity of concurrency such that one asynchronous telephony process is represented by one process in the language. (Rules out Parlog.) We must therefore develop our own language with the desirable features of Lisp, Prolog and

History of Erlang II

Parlog, but with concurrency and error recovery built into the language.

1987 The first experiments with Erlang.

1988 ACS/Dunder Phase 1. Prototype construction of PABX functionality by external users. Erlang escapes from the lab!

1989 ACS/Dunder Phase 2. Reconstruction of 1/10 of the complete MD-110 system. Results: much more than 10 times greater gains in efficiency at construction compared with construction in PLEX!
Further experiments with a fast implementation of Erlang.

1990 Erlang is presented at ISS'90, which results in several new users, e.g Bellcore.

History of Erlang III

1991 Fast implementation of Erlang is released to users. Erlang is represented at Telecom'91 . More functionality such as ASN1-Compiler, graphical interface etc.

1992 A lot of new users, e.g. several RACE projects. Erlang is ported to VxWorks, PC, Macintosh etc. Three applications using Erlang are presented at ISS'92. The two first product projects using Erlang are started.

1993 Distribution is added to Erlang, which makes it possible to run a homogeneous Erlang system on a heterogeneous hardware. Decision to sell implementations Erlang externally. Separate organization in Ericsson started to maintain and support Erlang implementations and Erlang Tools.

Sequential programming

We start introducing the programming language Erlang by describing the main features of the sequential part of the language.
The sequential language of Erlang is a functional programming language, thus it shares many concepts with other languages like Haskell.
However, there are two important differences between Erlang and Haskell:

- Erlang is *dynamically typed:* functions may be applied to arguments of any type, and nonsensical expressions may cause *run-time errors* only. Only *arity* (the number of arguments) is relevant for resolving overloading.
- Erlang is *eager:* all arguments will be evaluated before passed into the function body.

Integers

```
10
-234
16#AB10F
2#110111010
$A
```

- `B#Val` is used to write numbers in base *B*.
- `$Char` is used for ASCII values (example $A instead of 65).

Floats

```
17.368
-56.654
12.34E-10.
```

A remark on dots

The dot `.` serves many purposes in an Erlang program, e.g. to terminate an expression or as part of float literals. Hence the expression `1.` is *not* a float but the expression `1` that is terminated by a `.`

Atoms

```
abcef
start_with_a_lower_case_letter
'Blanks␣can␣be␣quoted'
'Anything␣inside␣quotes␣\n\012'
```

- Indefinite length atoms are allowed.
- Any character code is allowed within an atom.

What are atoms?

Atoms are like *constants* that (usually) represent only themselves. They are similar to enumeration constants, but need not be declared. Constants can be compared by equality.

Important atoms are **`ok`** and **`error`**.

Tuples

```
{123, bcd}
{123, def, abc}
{person, 'Joe', 'Armstrong'}
{abc, {def, 123}, jkl}
{}
```

- Tuples are created when declared and garbage collected.
- Used to store a fixed number of items.
- Tuples of any size are allowed.

Extracting values from tuples

= is not really an assignment operator but a *pattern matching* operator. It is like `let .. = ..` and can be used for decomposing tuples.

```
1> Point = {point, 10, 20}.
{point,10,20}
2> {point, X, Y} = Point.
{point,10,20}
3> X.
10
4> Y.
20
```


Lists

```
[123, xyz]
[123, def, abc]
[{person, 'Joe', 'Armstrong'},
 {person, 'Robert', 'Virding'},
 {person, 'Mike', 'Williams'}
]
"abcdefghi" becomes - [97,98,99,100,101,102,103,104,105]
"" becomes - []
```

- Used to store a variable number of items.
- Lists are dynamically sized.
- "..." is short for the list of integers representing the ASCII character codes of the enclosed within the quotes.

Variables

```
Abc
A_long_variable_name
AnObjectOrientatedVariableName
```

- Start with an Upper Case Letter.
- No "funny characters".
- Variables are used to store values of data structures.
- Variables can only be bound once! The value of a variable can never be changed once it has been set (bound).

Static single assignment

In mathematics, the occurrence of an `X` in different places of an equation denotes the same value. This is fundamental to algebra and solving equations.
In programming, we see `x=x+1`, where `x` is supposed to change value.
In Erlang, variables behave like in mathematics. When you "assign" a value to it, you make an assertion—a statement in fact. The variables has a value and that value cannot change after that.

Complex data structures

```
[{{person,'Joe', 'Armstrong'},
{telephoneNumber, [3,5,9,7]},
{shoeSize, 42},
{pets, [{cat, tubby},{cat, tiger}]},
{children,[{thomas, 5},{claire,1}]}},
 {{person,'Mike','Williams'},
{shoeSize,41},
{likes,[boats, beer]},
...
```

- Arbitrary complex structures can be created.
- Data structures are created by writing them down (no explicit memory allocation or deallocation is needed etc.).
- Data structures may contain bound variables.

Pattern matching I

- `A = 10` Succeeds — binds A to 10
- `{B, C, D}= {10, foo, bar}` Succeeds — binds B to 10, C to foo and D to bar
- `{A, A, B}= {abc, abc, foo}` Succeeds — binds A to abc, B to foo
- `{A, A, B}= {abc, def, 123}` Fails
- `[A,B,C]= [1,2,3]` Succeeds — binds A to 1, B to 2, C to 3
- `[A,B,C,D]= [1,2,3]` Fails

Pattern matching II

- `[A,B|C]= [1,2,3,4,5,6,7]` Succeeds — binds A = 1, B = 2, C = [3,4,5,6,7]
- `[H|T]= [1,2,3,4]` Succeeds — binds H = 1, T = [2,3,4]
- `[H|T]= [abc]` Succeeds — binds H = abc, T = []
- `[H|T]= []` Fails
- `{A,_, [B|_],{B}}= {abc,23,[22,x],{22}}` Succeeds — binds A = abc, B = 22

Note the use of "_", the anonymous (don't care) variable.

Function calls

```
module:func(Arg1, Arg2, ... Argn)

func(Arg1, Arg2, .. Argn)
```

- Arg1 ... Argn are any Erlang data structures.
- The function and module names (`func` and **`module`** in the above) must be atoms.
- A function can have zero arguments. (e.g. **`date`**`()` returns the current date).
- Functions are defined within *modules.*
- Functions must be exported before they can be called from outside the module where they are defined.

List comprehension

```
qsort([]) -> [] ;
qsort([A|L]) -> qsort([X || X <- L, X < A])
                ++ [A] ++
                qsort([X || X <- L, X >= A]) .
```

List comprehension

The expression `[X || X <- L, X < A]` is a list comprehension expression and means: "construct a list of elements `X` such that `X` comes from `L` and `X < A`".

Module system

```
-module(demo).
-export([double/1]).

double(X) ->
times(X, 2).

times(X, N) ->
X * N.
```

- `double` can be called from outside the module, `times` is local to the module.
- `double/1` means the function `double` with one argument (Note that `double/1` and `double/2` are two different functions).

Punctuation in Erlang

- Commas (`,`) separate arguments in function calls, data constructors and patterns.
- Periods (`.`) (followed by a white space) separate entire functions and expressions in the shell.
- Semicolons (`;`) separate *clauses.* We find a clause in several contexts: in function definitions and in **`case`**, **`if`**, `try .. catch`, and **`receive`** expressions.
  Whenever we see sets if patterns followed by expressions, we will see semicolons as separators:

```
Pattern2 -> Expression1;
Pattern2 -> Expression2;
...
```


Starting the system

```
unix> erl
Eshell V2.0
1> c(demo).
double/1 times/2 module_info/0
compilation_succeeded
2> demo:double(25).
50
3> demo:times(4,3).
** undefined function:demo:times[4,3] **
** exited: {undef,{demo,times,[4,3]}} **
4> 10 + 25.
35
5>
```

- c(File) compiles the file `File.erl`.
- `1>` , `2>`, ... are the shell prompts.
- The shell sits in a read-eval-print loop.

Builts-in functions (BIFs)

```
date()
time()
length([1,2,3,4,5])
size({a,b,c})
atom_to_list(an_atom)
list_to_tuple([1,2,3,4])
integer_to_list(2234)
tuple_to_list({})
```

- Are in the module Erlang.
- Do what you cannot do (or is difficult to do) in Erlang.
- Modify the behaviour of the system.
- Described in the BIFs manual.

Function syntax

Is defined as a collection of clauses.

```
func(Pattern1, Pattern2, ...) ->
... ;
func(Pattern1, Pattern2, ...) ->
... ;
...
func(Pattern1, Pattern2, ...) ->
... .
```


Evaluation rules

- Clauses are scanned sequentially until a match is found.
- When a match is found all variables occurring in the head become bound.
- Variables are local to each clause, and are allocated and deallocated automatically.
- The body is evaluated sequentially.

Functions (cont)

```
-module(mathStuff).
-export([factorial/1, area/1]).

factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).

area({square, Side}) -> Side * Side;
area({circle, Radius}) ->
   % almost :-)
  3 * Radius * Radius;
area({triangle, A, B, C}) ->
  S = (A + B + C)/2,
  math:sqrt(S*(S-A)*(S-B)*(S-C));
area(Other) -> {invalid_object, Other}.
```


Evaluation example

```
factorial(0) -> 1;
factorial(N) ->
  N * factorial(N-1)
```

- Variables are local to each clause.
- Variables are allocated and deallocated automatically.

`> factorial(3)`
matches `N = 3` in clause 2
⇝ `3 * factorial(3 - 1)`
⇝ `3 * factorial(2)`
matches `N = 2` in clause 2
⇝ `3 * 2 * factorial(2 - 1)`
⇝ `3 * 2 * factorial(1)`
matches `N = 1` in clause 2
⇝ `3 * 2 * 1 * factorial(1 - 1)`
⇝ `3 * 2 * 1 * factorial(0)`
⇝ `3 * 2 * 1 * 1` (clause 1)
⇝ `6`

Guarded function clauses

```
factorial(0) -> 1;
factorial(N) when N > 0 ->
N * factorial(N - 1).
```

- The reserved word when introduces a guard.
- Fully guarded clauses can be re-ordered.

```
factorial(N) when N > 0 ->
N * factorial(N - 1);
factorial(0) -> 1.
```

- This is NOT the same as:

```
factorial(N) ->
N * factorial(N - 1);
factorial(0) -> 1.
```

(incorrect!!)

Examples of guards I

```
number(X)          % X is a number
integer(X)         % X is an integer
float(X)           % X is a float
atom(X)            % X is an atom
tuple(X)           % X is a tuple
list(X)            % X is a list

length(X) == 3     % X is a list of length 3
size(X) == 2       % X is a tuple of size 2.

X > Y + Z          % X is > Y + Z
X == Y             % X is equal to Y
X =:= Y            % X is exactly equal to Y
                   % (i.e. 1 == 1.0 succeeds but
                   %  1 =:= 1.0 fails)
```

Examples of guards II

- All variables in a guard must be bound.
- See the User Guide for a full list of guards and allowed function calls.

Traversing lists I

```
average(X) -> sum(X) / len(X).

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

len([_|T]) -> 1 + len(T);
len([]) -> 0.
```

Note the pattern of recursion is the same in both cases. This pattern is very common.

Traversing lists II

Two other common patterns:

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

member(H, [H|_]) -> true;
member(H, [_|T]) -> member(H, T);
member(_, []) -> false.
```


Lists and accumulators

```
average(X) -> average(X, 0, 0).

average([H|T], Length, Sum) ->
average(T, Length + 1, Sum + H);
average([], Length, Sum) ->
Sum / Length.
```

- Only traverses the list *once*
- Executes in constant space (tail recursive)
- The variables `Length` and `Sum` play the role of accumulators
- N.B. `average([])` is not defined — (you cannot have the average of zero elements) — evaluating `average([])` would cause a run-time error — we discuss what happens when run time errors occur in the section on error handling.

Functions are values

- Every function can be used as a value and passed to other functions

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

- Functions may be anonymous, like the identity function
  `fun I ->I end.`
- Together, we can express many algorithms quite concisely.

```
for(1, 10, fun I -> I end).
for(1, 10, fun I -> I * I end).
for(1, 10, fun I -> I < 5 end).
```


Shell commands

- h() history. Print the last 20 commands.
- b() bindings. See all variable bindings.
- f() forget. Forget all variable bindings.
- f(Var) forget. Forget the binding of variable X. This can *only* be used as a command to the shell — *not* in the body of a function!
- e(n) evaluate. Evaluate the *n*th command in history.
- e(-1) evaluate the previous command.

- Edit the command line as in Emacs
- See the User Guide for more details and examples of use of the shell.

Special functions

`apply(Mod, Func, Args)`

- Apply the function `Func` in the module `Mod` to the arguments in the list `Args`.
- `Mod` and `Func` must be atoms (or expressions which evaluate to atoms).

```
1> apply( lists1,min_max,[[4,1,7,3,9,10]]).
{1, 10}
```

- Any Erlang expression can be used in the arguments to apply.

Special forms

```
case lists:member(a, X) of
true ->
... ;
false ->
...
end,
...

if
integer(X) -> ... ;
tuple(X) -> ...
end,
...
```

Not really needed - but useful.

Definitions

Process A concurrent activity. A complete virtual machine. The system may have many concurrent processes executing at the same time.

Message A method of communication between processes.

Time-out Mechanism for waiting for a given time period.

Registered Process Process which has been registered under a name.

Client/Server Model Standard model used in building concurrent systems.

Creating a new process I

Figure: Before: Single process

Code in Pid1:

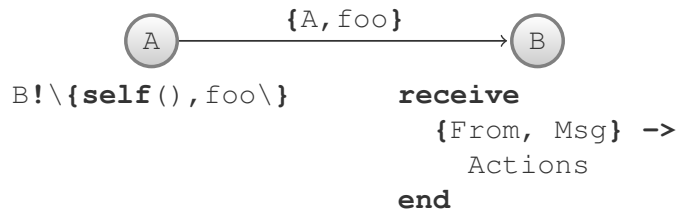
```
Pid2 = spawn(Mod, Func, Args)
```


Creating a new process II

Figure: After: Two process

`Pid2` is process identifier of the new process — this is known only to process `Pid1`.

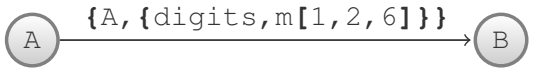
Simple message with `self()`



self() returns the Process Identity (Pid) of the process executing this function.

`From` and `Msg` become bound when the message is received. Messages can carry data.

Message with data




```
B!\{self(),{digits,[1,2,6]\}
```

```
receive
  {From,{digits,D}} ->
    process(D);
end
```

- Messages can carry data and be selectively unpacked.
- The variables A and D become bound when receiving the message.
- If A is bound before receiving a message then only data from this process is accepted.

An echo process

```
-module(echo).
-export([go/0, loop/0]).

go() ->
  Pid2 = spawn(echo, loop, []),
  Pid2 ! {self(), hello},
  receive
    {Pid2, Msg} ->
      io:format("P1␣~w~n",[Msg])
  end,
  Pid2 ! stop.
```

```
loop() ->
  receive
    {From, Msg} ->
      From ! {self(), Msg},
      loop();
    stop ->
      true
  end.
```


Selective message reception

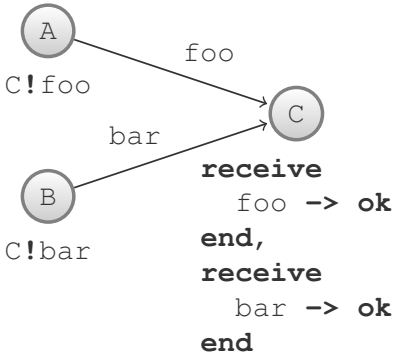


Figure: Two sender and one reader using two receive statements

The message `foo` is received — then the message `bar` — irrespective of the order in which they were sent.

Selection of any message

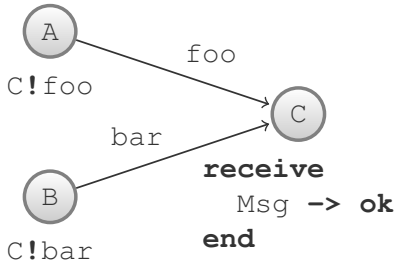


Figure: Two sender and one reader using one receive statement

The first message to arrive at the process C will be processed — the variable Msg in the process C will be bound to one of the atoms foo or bar depending on which arrives first.

A telephony example

Figure: A and B sends to Call

```
ringing_a(A, B) ->
  receive
    {A, on_hook} ->
      A ! {stop_tone, ring},
      B ! terminate,
      idle(A);
    {B, answered} ->
      A ! {stop_tone, ring},
      switch ! {connect, A, B},
      conversation_a(A, B)
  end.
```

This is the code in the process `Call`. `A` and `B` are local, bound variables in the process `Call`.

Pids can be sent in messages

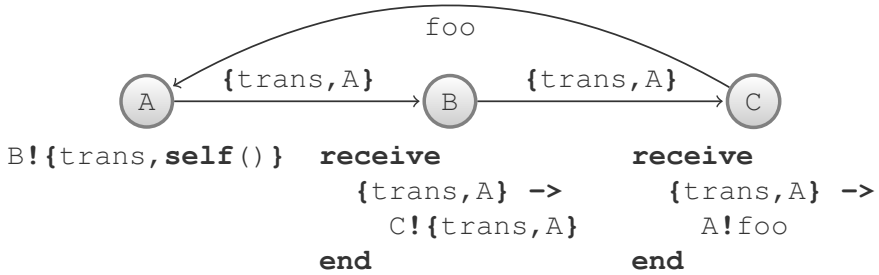


Figure: Pass messages

- A sends a message to B containing the Pid of A.
- B sends a transfer message to C.
- C replies directly to A.

Registered processes

register(Alias, Pid) registers the process `Pid` with the name `Alias`.

```
start() ->
Pid = spawn(num_anal, server, [])
register(analyser, Pid).

analyse(Seq) ->
analyser ! {self(),{analyse,Seq}},
receive
{analysis_result,R} ->
R
end.
```

Any process can send a message to a registered process.

Client server model

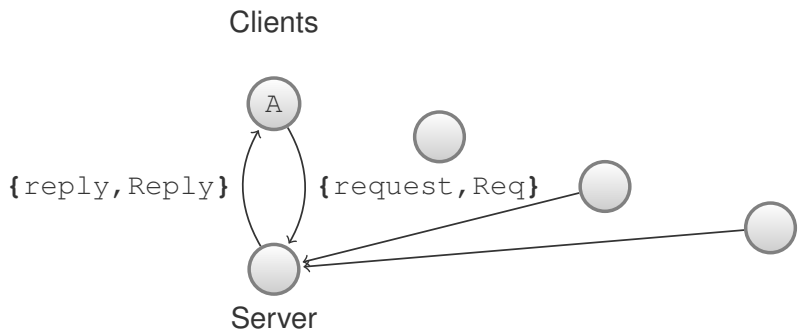


Figure: Clients and server

Protocol

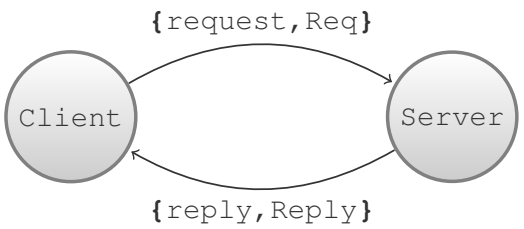


Figure: Protocol

Server code

```
-module(myserver).

server(Data) ->
  receive
    {From,{request,X}} ->
      {R, Data1} = fn(X, Data),
      From ! {myserver,{reply, R}},
      server(Data1)
  end.
```


Interface library

```
-export([request/1]).

request(Req) ->
  myserver ! {self(),{request,Req}},
  receive
    {myserver,{reply,Rep}} ->
     Rep
  end.
```


Time-outs

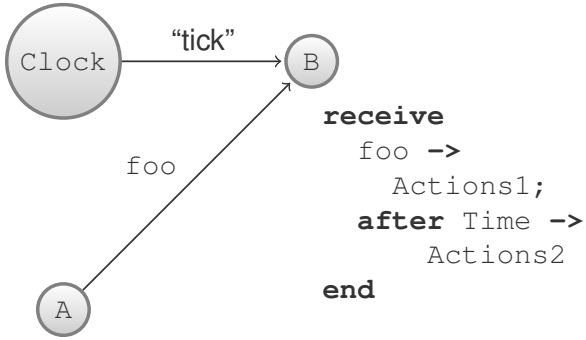


Figure: Action after time-out

If the message `foo` is received from `A` within the time `Time` perform `Actions1` otherwise perform `Actions2`.

`sleep(T)` process suspends for T ms

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


`suspend()` process suspends indefinitely

```
suspend() ->
  receive
  after
    infinity ->
      true
  end.
```


`alarm(T, What)`

The message `What` is sent to the current process in `T` milliseconds from now

```
set_alarm(T, What) ->
  spawn(timer, set, [self(),T,What]).

set(Pid, T, Alarm) ->
  receive
  after
    T ->
      Pid ! Alarm
  end.

receive
  Msg ->
    ... ;
end
```


`flush()` flushes the message buffer

```
flush() ->
  receive
    Any ->
      flush()
  after
    0 ->
      true
  end.
```

A value of 0 in the time-out means: check the message buffer first and if it is empty execute the following code.

Definitions

Link A bi-directional propagation path for exit signals.

Exit Signal Transmit process termination information.

Error trapping The ability of a process to process exit signals as if they were messages.

Exit signals are sent when processes crash

When a process crashes (e.g. failure of a BIF or a pattern match) Exit Signals are sent to all processes to which the failing process is currently linked. If nobody listens, the system goes down.

```
1> F = fun () -> receive X -> list_to_atom(X) end end .
#Fun<erl_eval.20.67289768>
2> Pid = spawn(F) .
<0.34.0>
3> Pid ! bla .
bla

=ERROR REPORT==== 28-Sep-2009::22:46:43 ===
Error in process <0.34.0> with exit value: {badarg,[{erlang,l
```


Monitoring using an `on_exit` handler

```
-module(lib_misc).
-export([on_exit/2]).
on_exit(Pid,Fun) ->
  spawn(fun() ->
          process_flag(trap_exit, true),
          link(Pid),
          receive
            {'EXIT', Pid, Why} -> Fun(Why)
          end
        end).
```


Monitoring using an `on_exit` handler II

```
1> F = fun () -> receive X -> list_to_atom(X) end end .
#Fun<erl_eval.20.67289768>
2> Pid = spawn(F) .
<0.34.0>
3> lib_misc:on_exit(Pid, fun(Why) ->
                           io:format("␣~p␣died␣with:␣~p~n", [
                         end).
4> Pid!hello.
 <0.35.0> died with: {badarg,[{erlang,list_to_atom,[hello]}]}
hello
```

Exit signals are sent when processes crash

When a process crashes (e.g. failure of a BIF or a pattern match) Exit Signals are sent to all processes to which the failing process is currently linked.

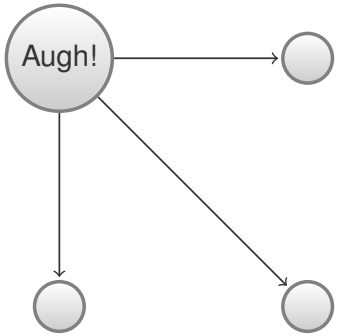


Figure: Dies and sends signal to linked processes

Exit signals propagate through links I

Suppose we have a number of processes which are linked together, as in the following diagram. Process A is linked to B, B is linked to C (The links are shown by the arrows).
Now suppose process A fails - exit signals start to propagate through the links:

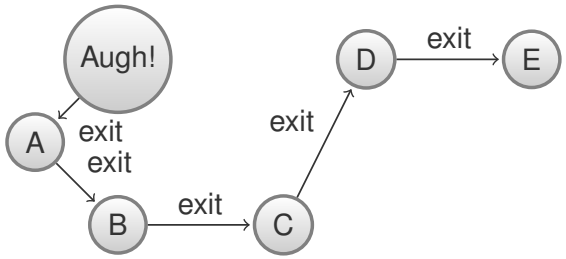


Figure: Exit signals propagating, A to B to C. . .

Exit signals propagate through links II

These exit signals eventually reach all the processes which are linked together.

The rule for propagating errors is:

If the process which receives an exit signal, caused by an error, is not trapping exits then the process dies and sends exit signals to all its linked processes.

Processes can trap exit signals

In the following diagram P1 is linked to P2 and P2 is linked to P3. An error occurs in P1 - the error propagates to P2. P2 traps the error and the error is not propagated to P3.

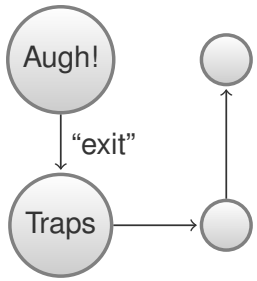


Figure: Process traps exit and does not propagate exit

Processes can trap exit signals II

P2 has the following code:

```
receive
    {'EXIT', P1, Why} ->
... exit signals ...
    {P3, Msg} ->
... normal messages ...
end
```


Complex exit signal propagation I

Suppose we have the following set of processes and links:

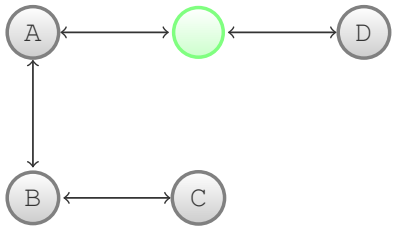


Figure: Bidirectional links in chain of processes

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

Complex exit signal propagation II

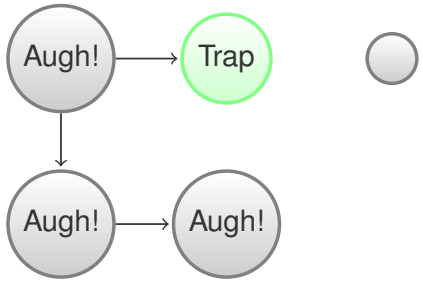


Figure: Process that traps exit stops propagation

If an error occurs in any of `A`, `B`, or `C` then *all* of these process will die (through propagation of errors). Process `D` will be unaffected.

Exit signal propagation semantics

- When a process terminates it sends an exit signal, either normal or non-normal, to the processes in its link set.
- A process which is not trapping exit signals (a normal process) dies if it receives a non-normal exit signal. When it dies it sends a non-normal exit signal to the processes in its link set.
- A process which is trapping exit signals converts all incoming exit signals to conventional messages which it can receive in a receive statement.
- Errors in BIFs or pattern matching errors send automatic exit signals to the link set of the process where the error occurred.

Robust systems can be made by layering

By building a system in layers we can make a robust system. Level 1 traps and corrects errors occurring in Level 2. Level 2 traps and corrects errors occurring in the application level.
In a well designed system we can arrange that application programmers will not have to write any error handling code since all error handling is isolated to deeper levels in the system.

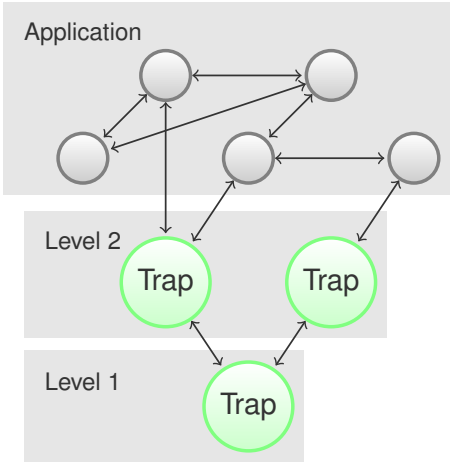


Figure: Hierarchical layered trapping

Primitives for exit signal handling I

link(Pid) Set a bi-directional link between the current process and the process Pid

process_flag(trap_exit, true) Set the current process to convert exit signals to exit messages, these messages can then be received in a normal receive statement.

exit(Reason) Terminates the process and generates an exit signal where the process termination information is Reason.

Primitives for exit signal handling II

What really happens is as follows: Each process has an associated mailbox - `Pid !Msg` sends the message `Msg` to the mailbox associated with the process `Pid`.
The **`receive .. end`** construct attempts to remove messages from the mailbox of the current process. Exit signals which arrive at a process either cause the process to crash (if the process is not trapping exit signals) or are treated as normal messages and placed in the process mailbox (if the process is trapping exit signals). Exit signals are sent implicitly (as a result of evaluating a BIF with incorrect arguments) or explicitly (using **`exit`**`(Pid, Reason)`, or **`exit`**`(Reason)`).
If `Reason` is the atom `normal` the receiving process ignores the signal (if it is not trapping exits). When a process terminates without an error it sends normal exit signals to all linked processes. *Don't say you didn't ask!*

Overview of `process_flags`

`trap_exit`	Exit Signal	Action
`true`	`kill`	Die: Multi-cast the exit signal `killed` to the link set.
`true`	`X`	Add `EXIT',Pid,X'EXIT',Pid,X` to the mailbox.
`false`	normal	Continue: Do-nothing signal vanishes.
`false`	`kill`	Die: Multi-cast the exit signal `killed` to the link set.
`false`	`X`	Die: Multi-cast the exit signal `X` to the link set.

Table: Process flag `trap_exit` and exit behaviour

`kill` cannot be trapped. It kills any process, including system processes. It is used to kill rogue processes. `killed` *can* be trapped.

I don't care if a process I create crashes

```
Pid = spawn(fun () -> .. end).
```

If the spawned process crashes, the current process continues.

I want to die if a process I create crashes

```
Pid = spawn_link(fun () -> .. end).
```

If the spawned process crashes with a *non-normal,* the current process crashes, too.
To be strict: The process must crash using a non-normal exit to crash the creator. Additionally, the creating process must *not* have been set to trap exits.

I want to handle errors if a process I create crashes

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

loop(State) ->
  receive
    {'EXIT', SomePid, Why} ->
      %% do something with the error
      loop(State1);
    ...
  end.
```

If the spawned process crashes with a *non-normal,* the current process receives the error in `loop`.

A robust server I

The following server assumes that a client process will send an `alloc` message to allocate a resource and then send a `release` message to deallocate the resource.
This is unreliable! What happens if the client crashes before it sends the release message?

A robust server II

```
top(Free, Allocated) ->
  receive
    {Pid, alloc} ->
      top_alloc(Free, Allocated, Pid);
    {Pid ,{release, Resource}} ->
      Allocated1 = delete({Resource,Pid}, Allocated),
      top([Resource|Free], Allocated1)
    end.

top_alloc([], Allocated, Pid) ->
  Pid ! no,
  top([], Allocated);

top_alloc([Resource|Free], Allocated, Pid) ->
  Pid ! {yes, Resource},
  top(Free, [{Resource,Pid}|Allocated]).
```


A robust server III

This is the top loop of an allocator with no error recovery. Free is a list of unreserved resources. Allocated is a list of pairs `{Resource, Pid}`, showing which resource has been allocated to which process.

Allocator with error recovery

The following shows the code of a reliable server. If a client crashes after it has allocated a resource and before it has released the resource, then the server will automatically release the resource.
The server is linked to the client during the time interval the resource is allocated. If an exit message is received from the client during this time the resource is released.

```
top_recover_alloc([], Allocated, Pid) ->
  Pid ! no, top_recover([], Allocated);

top_recover_alloc([Resource|Free], Allocated, Pid) ->
  %% No need to unlink.
  Pid ! {yes, Resource},
  link(Pid),
  top_recover(Free, [{Resource,Pid}|Allocated]).
```



```
top_recover(Free, Allocated) ->
  receive
    {Pid , alloc} ->
      top_recover_alloc(Free, Allocated, Pid);
    {Pid, {release, Resource}} ->
      unlink(Pid),
      Allocated1 = delete({Resource, Pid}, Allocated),
      top_recover([Resource|Free], Allocated1);
    {'EXIT', Pid, Reason} ->
      %% No need to unlink.
      Resource = lookup(Pid, Allocated),
      Allocated1 = delete({Resource, Pid}, Allocated),
      top_recover([Resource|Free], Allocated1)
  end.
```

Not done: when multiple multiple resources are allocated to the same process we should check to see that the process has not allocated more than one resource before doing the `unlink(Pid)`.

Allocator utilities

```
delete(H, [H|T]) ->
    T;
delete(X, [H|T]) ->
    [H|delete(X, T)].

lookup(Pid, [{Resource,Pid}|_]) ->
    Resource;
lookup(Pid, [_|Allocated]) ->
    lookup(Pid, Allocated).
```


Distributed programming

- Erlang provides a couple of methods for distributed programming, i.e. programs that are to be executed on many computers that are connected by a network.
- First, Erlang provides its native RPC based system; we'll look at this one here.
- Erlang supports communication using sockets. This is useful when we deal with heterogeneous applications.
- We introduce "distributed Erlang" by way of an example.

The property map server I

```
-module(propmap).
-export([start/0, lookup/1, store/2]).

start() ->
    register(propmap, spawn(fun () -> loop() end)).

lookup(Key) -> rpc({lookup, Key}).

store(Key, Value) -> rpc({store, Key, Value}).

rpc(C) ->
    propmap!{self(),C},
    receive
        {propmap, Reply} ->
            Reply
    end.
```


The property map server II

```
loop() ->
    receive
        {From, {lookup, Key}} ->
            From ! {propmap, get(Key)},
            loop();
        {From, {store, Key, Value}} ->
            put(Key, Value),
            From!{propmap, ok},
            loop()
    end.
```

Starting two Erlang nodes on the same machine

1. In one shell execute: `erl -sname arthur`
2. In another shell execute: `erl -sname ford`
3. Note the prompt of the Erlang shell: `(arthur@arthur)1>`
4. Do some distributed computation:

```
(ford@arthur)1> propmap:start().
true
(arthur@arthur)1> rpc:call(ford@arthur, propmap,
                           store, [wodka, 5]).
ok
(ford@arthur)2> propmap:lookup(wodka).
5
```


Lets try different machines

```
ssh xian erl -name xian -setcookie homemade
ssh harbin erl -name harbin -setcookie homemade

(harbin@harbin)1> propmap:start().
true
(xian@xian)1> rpc:call(harbin@harbin.mi.fu-berlin.de, propmap,
                       store, [gin, 10]).
ok
(harbin@harbin)2> propmap:lookup(gin).
10
```

If all machines are on the same subnet or we do not have DNS, we use `-sname` and might need to qualify with IP numbers. The file `propmap.{erl,beam}` must be in the same directory erl is executed in.

The module `rpc`

- `rpc:call(Node, Module, Function, Args)` evaluates a function on a different node.
- Other functions are provided that offer much more. Have a look for yourself.

Other distribution primitives I

- **`spawn`**`(Node, Fun)` works like **`spawn`**`(Fun)` but creates the new process on `Node`
- **`spawn`**`(Node, Mod, Fun, ArgList)` works likewise.
- `spawn_link(Node, Mod)` works likewise.
- `spawn_link(Node, Mod, Fun, ArgList)` works likewise.
- `disconnect_node(Node)` disconnects a node by force.
- `monitor(Node, Flag)` sets the monitoring status. When `Flag` is **`true`**, the caller will be sent `odeup, Nodenodeup, Node` and `odedown, Nodenodedown, Node` messages when the node joins or leaves the network.

Other distribution primitives II

- **node**() returns the name of the local node. `nonode@nohost` means that the node is not distributed.
- **node**(`Arg`) returns the node where `Arg` is located (`Arg` is either a PID, a reference, or a port).
- **nodes**() returns a list of nodes this node is connected to.
- `is_alive()` returns true if the local node is alive and can be part of a distributed system.
- `egName, NodeRegName, Node!Msg` sends the message `Msg` to the registered process `RegName` on node `Node`.

The cookie protection system

- For two Erlang nodes to communicate they must have the same *magic cookie*
- Lets look at some methods

Cookie file

- Store a string in the file `$HOME/.erlang.cookie`
- Make sure, that the file is only readable to its owner.
- That file should be distributed to all machines we want to join to a network.
- Note: The file can be conveniently distributed by NFS.

Cookie on the command line

- Start the Erlang shell with `erl -setcookie ...`

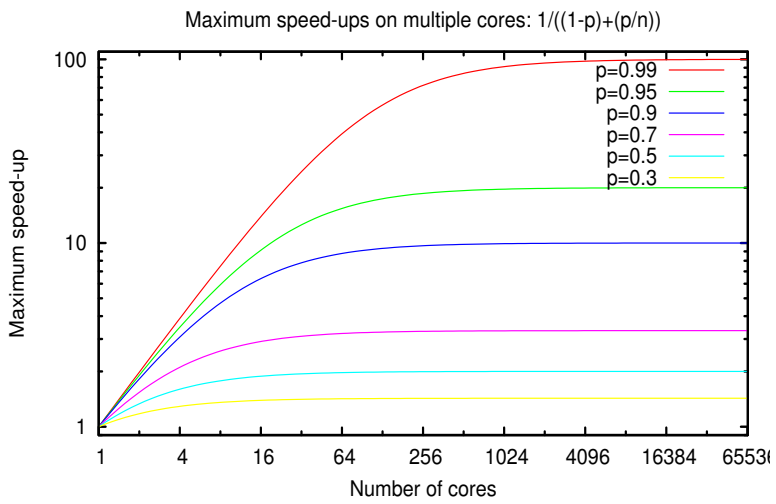
Setting the cookie from Erlang

- The built-in function `set_cookie(Node, C)` sets the cookie of the node `Node` to the atom `C`.
- Example: `set_cookie(node(), cookie`

Getting most out of multi-core I

- To round up this course, we discuss some strategies for writing Erlang code that scales well on multi-core processors.
- Most sequential tasks cannot be sped up arbitrarily, because of Amdahl's law: If $p \in [0, 1]$ is the portion that can be executed in parallel and n is the number of cores, the expected speed-up is: $\frac{1}{(1-p)+\frac{p}{n}}$

Getting most out of multi-core II



Getting most out of multi-core II

- The graphs show *maximum* speed-up. Synchronisation between threads/processes will lower that value.
- Observe: $\lim_{n\to\infty} \frac{1}{(1-p)+\frac{p}{n}} = \frac{1}{1-p}$ and the lower the value of p is, the faster we get close to that limit.
- We want to maximise the value of p, which leads to a bigger potential speed-up.
- A large value of p means that the tasks are independent.
- Avoid synchronisation by "long-running" computations (for a suitable definition of "long-running").
- Avoid side-effects! Side effects are the main reason for slow-downs, because processes must synchronise.
- Create many processes (but not too many, since they do not contribute to the speed-up)

A parallel map

```
pmap(F, L) ->
    S=self(),
    Ref=erlang:make_ref(),
    Pids=map(fun(I) ->
                    spawn(fun() -> do_f(S, Ref, F, I) end)
            end, L),
    gather(Pids, Ref).

do_f(Parent, Ref, F, I) ->
    Parent ! {self(), Ref, (catch F(I))}.

gather([Pid|T], Ref) ->
    receive
        {Pid, Ref, Ret} -> [Ret|gather(T, Ref)]
    end;
gather([], _) -> [].
```


When to use parallel map?

- Don't use `pmap`, when the amount of work done in `F` is small. `F` should do substantially more work than setting up the process and waiting for the reply. Don't write `pmap(fun(I) ->2*I end, L)`.
- Don't create too many processes. `pmap(F,L)` creates `length(L)` processes, which could exhaust system resources for large values. It is usually not smart to create many more processes than the machine has cores.
- Think about the abstraction you need: When you do not care for the order of elements when mapping, allow permutations. That saves time in gather.

An alternative parallel map

```
pmap(F, L) ->
    S=self(),
    Ref=erlang:make_ref(),
    Pids=map(fun(I) ->
                    spawn(fun() -> do_f(S, Ref, F, I) end)
            end, L),
    gather(length(L), Ref, []).

do_f(Parent, Ref, F, I) ->
    Parent ! {Ref, (catch F(I))}.

gather(0, _, L) -> L;
gather(N, Ref, L) ->
    receive
        {Ref, Ret} -> gather(N-1, Ref, [Ret|L])
    end.
```

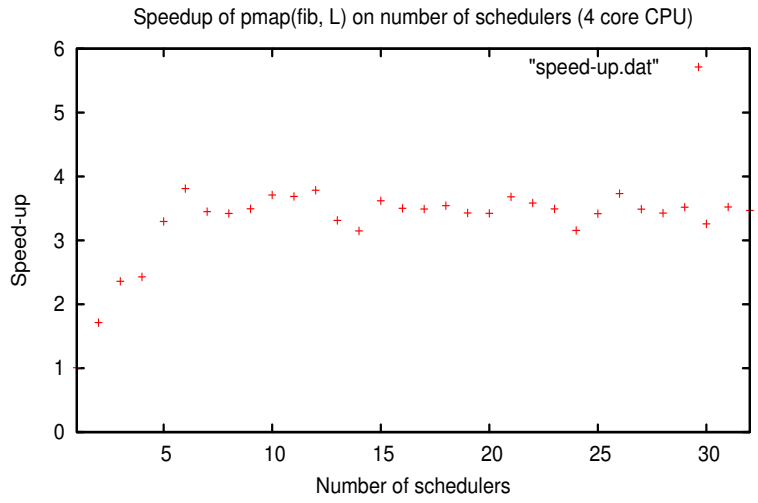

Let's do some measurement

Let `L` be a list of 100 twenty-sevens. We want to compute `map(fib, L)` where `fib(n)` is the `n`th Fibonacci number.
Computing the 27th Fibonacci number is a big computation compared to creating a process, sending it the number 27 and receiving a result.
Thus, we can expect that computing `map(fib, L)` can be parallelised well by executing `pmap(fib, L)`

Running SMP Erlang

- Try to use HIPE where available, it may provide superior performance.
- The command line parameter `-smp` starts the Erlang interpreter in SMP mode.
- The command line parameter `+S N` specifies the number of *schedulers* to use.
  - A scheduler is a complete virtual machine that knows about all the other virtual machines.
  - The default value is the number of cores in the machine.
  - The parameter can be controlled:
    - When doing performance measurements we can vary the number of cores.
    - On a single core machine we can emulate a multiprocessor machine
    - Using more schedulers than cores on a multi-core machine may increase throughput and lets the system behave "better". The effect is not understood and subject to research.

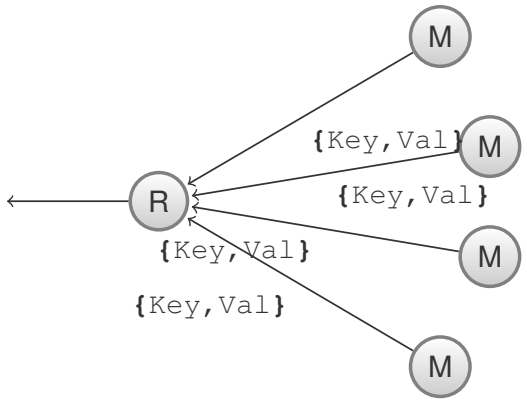
Measuring `pmap(fib, L)`



`mapreduce` I

- `mapreduce` is a parallel higher-order function.
- It is *not* related to the `map` function.
- It was proposed by Jeffrey Dean and Sanjay Ghemawat of Google [**?**]
- It is said to be used daily on Google clusters
- One can implement `mapreduce` in many ways with different semantics. `mapreduce` is more of a *family* of algorithms

`mapreduce` II



- A number of *mapping processes* generate a stream of `{Key,Val}` pairs
- A *reduce* process reduces these pairs, combining pairs with the same `Key`

mapreduce III

Lets look at an implementation. Specification:

- `mapreduce(F1, F2, Acc0, L)`
- `F1(Pid, X)` is the mapping function. The job of `F1` is to send a stream of `ey,ValKey,Val` to `Pid` and then terminate. `mapreduce` spawns one `F1` process per member of `L`.
- `F2(Key, [Value], Acc0)` is the reduction function. When all mapping processes have terminated, the reduce process will have merged all values. Then, `F2` *folds* the `ey,[Value]Key,[Value]` pairs with initial accumulator `Acc0`.
- `Acc0` is the initial accumulator to `F2`
- `L` is a list of values `X` which will be passed to the `F1` processes.

mapreduce IV

```
-module(phofs).
-export([mapreduce/4]).

-import(lists, [foreach/2]).

%% F1(Pid, X) -> sends {Key,Val} messages to Pid
%% F2(Key, [Val], AccIn) -> AccOut

mapreduce(F1, F2, Acc0, L) ->
    S = self(),
    Pid = spawn(fun() -> reduce(S, F1, F2, Acc0, L) end),
    receive
        {Pid, Result} ->
            Result
    end.
```


mapreduce V

```
reduce(Parent, F1, F2, Acc0, L) ->
    process_flag(trap_exit, true),
    ReducePid = self(),
    %% Create the Map processes
    %%   One for each element X in L
    foreach(fun(X) ->
                    spawn_link(fun() -> do_job(ReducePid, F1, X
            end, L),
    N = length(L),
    %% make a dictionary to store the Keys
    Dict0 = dict:new(),
    %% Wait for N Map processes to terminate
    Dict1 = collect_replies(N, Dict0),
    Acc = dict:fold(F2, Acc0, Dict1),
    Parent ! {self(), Acc}.
```


mapreduce VI

```
%% collect_replies(N, Dict)
%%     collect and merge {Key, Value} messages from N process
%%     When N processes have terminated return a dictionary
%%     of {Key, [Value]} pairs

collect_replies(0, Dict) ->
    Dict;
collect_replies(N, Dict) ->
    receive
        {Key, Val} ->
            case dict:is_key(Key, Dict) of
                true ->
                    Dict1 = dict:append(Key, Val, Dict),
                    collect_replies(N, Dict1);
```

mapreduce VII

```
                false ->
                    Dict1 = dict:store(Key,[Val], Dict),
                    collect_replies(N, Dict1)
            end;
        {'EXIT', _,  Why} ->
            collect_replies(N-1, Dict)
    end.

%% Call F(Pid, X)
%%   F must send {Key, Value} messsages to Pid
%%     and then terminate

do_job(ReducePid, F, X) ->
    F(ReducePid, X).
```


Scope of Variables

Variables in a clause exist between the point where the variable is first bound and the last textual reference to the variable.

Consider the following code:

```
1... f(X) ->
2...   Y = g(X),
3...   h(Y, X),
4...   p(Y),
5...   f(12).
```

- line 1 - the variable X is defined (i.e. it becomes bound when the function is entered).
- line 2 - X is used, Y is defined (first occurrence).
- line 3 - X and Y are used.
- line 4 - Y is used. The space used by the system for storing X can be reclaimed.
- line 5 - the space used for Y can be reclaimed.

Scope of variables in if/case/receive

The set of variables introduced in the different branches of an if/case/receive form must be the same for all branches in the form except if the missing variables are not referred to after the form.

```
f(X) ->
    case g(X) of
true -> A = h(X), B = 7;
false -> B = 6
    end,
    ...,
    h(A),
    ...
```


Scope of variables in if/case/receive

If the **true** branch of the form is evaluated, the variables `A` and `B` become defined, whereas in the **false** branch only `B` is defined.
Whether or not this is an error depends upon what happens after the case function. In this example it is an error, when a future reference is made to `A` in the call `h(A)` — if the **false** branch of the case form had been evaluated then `A` would have been undefined.

Catch and Throw I

Suppose we have defined the following:

```
-module(try).
-export([foo/1]).

foo(1) -> hello;
foo(2) -> throw({myerror, abc});
foo(3) -> tuple_to_list(a);
foo(4) -> exit({myExit, 222}).
```

`try:foo(1)` evaluates to hello.
`try:foo(2)` tries to evaluate `throw({myerror, abc})` but no catch exists. The process evaluating `foo(2)` exits and the signal `{'EXIT',Pid,nocatch}` is broadcast to the link set of the process.
`try:foo(3)` broadcasts `{'EXIT', Pid, badarg}` signals to all linked processes.

Catch and Throw II

`try:foo(4)` since no catch is set the signal `{'EXIT',Pid,{myexit, 222}}` is broadcast to all linked processes.
`try:foo(5)` broadcasts the signal `{'EXIT',Pid,function_clause}` to all linked processes.
`catch try:foo(1)` evaluates to hello.
`catch try:foo(2)` evaluates to `{myError,abc}`.
`catch try:foo(3)` evaluates to `{'EXIT',badarg}`.
`catch try:foo(4)` evaluates to `{'EXIT',{myExit,222}}`.
`catch try:foo(5)` evaluates to `{'EXIT',function_clause}`.

Use of Catch and Throw

Catch and throw can be used to:

- Protect from bad code
- Cause non-local return from a function

Example

```
f(X) ->
  case catch func(X) of
    {'EXIT', Why} ->
      % BUG: error in BIF...
    {exception1, Args} ->
      % planned exception
    Normal ->
      % normal case
  end.
```

```
func(X) ->
  bar(X),
  ...
  ...

bar(X) ->
  throw({exception1, ...}),
  ...
```

The module `error_handler`

The module `error_handler` is called when an undefined function is called. If a call is made to `Mod:Func(Arg0,...,ArgN)` and no code exists for this function then `undefined_call(Mod, Func,[Arg0,...,ArgN])` in the module `error_handler` will be called.

The code in `error_handler` is almost like this:

```
-module(error_handler).
-export([undefined_call/3]).

undefined_call(Module, Func, Args) ->
  case code:if_loaded(Module) of
    true ->
      %% Module is loaded but not the function
      ...
      exit({undefined_function, {Mod, Func, Args}});
    false ->
      case code:load(Module) of
        {module, _} ->
          apply(Module, Func, Args);
        false ->
          ....
    end.
```


By evaluating `process_flag(error_handler, MyMod)` the user can define a private error handler. In this case the function `MyMod:undefined_function` will be called instead of `error_handler:undefined_function`.
Note: This is extremely dangerous

The Code loading mechanism

Consider the following:

```
-module(m).
-export([start/0,server/0]).

start() ->
    spawn(m,server,[]).

server() ->
    receive
Message ->
            do_something(Message),
            m:server()
    end.
```

When the function `m:server()` is called then a call is made to the latest version of code for this module.

If the call had been written as follows:

```
server() ->
    receive
Message ->
            do_something(Message),
            server()
    end.
```

Then a call would have been made to the current version of the code for this module.

Prefixing the module name (i.e. using the : form of call) allows the user to change the executing code on the fly.
The rules for evaluation are as follows:

- Must have the module prefix in the recursive call (`m:server()`) if we want to change the executing code on the fly.
- Without prefix, the executing code will not be exchanged with the new one.
- We can't have more than two versions of the same module in the system at the same time.

Ports

Ports:

- Provide byte stream interfaces to external UNIX processes.
- Look like normal Erlang processes, that are not trapping exits, with a specific protocol. That is, they can be linked to, and send out/react to exit signals.
- Communicates with a single Erlang process, this process is said to be connected.

The command:

```
Port = open_port ( {spawn,Process} , {packet,2} )
```

Starts an external UNIX process — this process reads commands from Erlang on file descriptor 0 and sends commands to Erlang by writing to file descriptor 1.

Port Protocols

Data is passed as a sequence of bytes between the Erlang processes and the external UNIX processes. he number of bytes passed is given in a 2 bytes length field.

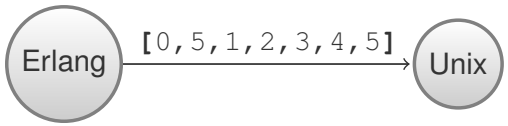


Figure: Erlang process passing data to Unix process

"C" should check return value from read. See p.259 in the book for more info.

Binaries

- A binary is a reference to a chunk of untyped memory.
- Binaries are primarily used for code loading over the network.
- Also useful when applications wants to shuffle around large amount of raw data.
- Several BIF's exist for manipulating binaries, such as: **binary_to_term**/1, term_to_binary/1, **binary_to_list**/1, split_binary/2, concat_binary/1, etc.
- open_port/2 can produce and send binaries.
- There is also a guard called **binary**(B) which succeeds if its argument is a Binary

References

References are Erlang objects with exactly two properties:

- They can be created by a program (using make_ref/0)
- They can be compared for equality.

Erlang references are unique, the system guarantees that no two references created by different calls to make_ref will ever match. The guarantee is not 100% - but differs from 100% by an insignificantly small amount :-).

References can be used for writing a safe remote procedure call interface

```
ask(Server, Question) ->
    Ref = make_ref(),
    Server ! {self(), Ref, Question},
    receive
        {Ref, Answer} ->
    Answer
    end.

server(Data) ->
    receive
{From, Ref, Question} ->
            Reply = func(Question, Data),
            From ! {Ref, Reply},
            server(Data);
...
    end.
```


Space Saving Optimisations

Here are two ways of computing the sum of a set of numbers contained in a list. The first is a recursive routine:

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

Note that we cannot Evaluate '+' until both its arguments are known. This formulation of sum(X) evaluates in space O(length(X)).

Tail-recursive version

The second is a tail recursive which makes use of an accumulator Acc:

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

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

The tail recursive formulation of `sum(X)`. Evaluates in constant space.
Tail recursive = the last thing the function does is to call itself.

Last Call Optimisation

The last call optimisation must be used in persistent servers.
For example:

```
server(Date) ->
  receive
    {From, Info} ->
      Data1 = process_info(From, Info, Data),
      server(Data1);
    {From, Ref, Query} ->
      {Reply, Data1} = process_query(From, Query,Data),
      From ! {Ref, Reply},
      server(Data1)
  end.
```

Note that the last thing to be done in any thread of computation must be to call the server.

Process Dictionary

Each process has a local store called the "Process Dictionary". The following BIFs are used to manipulate the process dictionary:

- get() returns the entire process dictionary.
- get(Key) returns the item associated with Key (Key is any Erlang data structure), or, returns the special atom undefined if no value is associated with Key.
- put(Key, Value) associate Value with Key. Returns the old value associated with Key, or, undefined if no such association exists.
- erase() erases the entire process dictionary. Returns the entire process diction before it was erased.
- erase(Key) erases the value associated with Key. Returns the old value associated with Key, or, undefined if no such association exists.
- get_keys(Value) returns a list of all keys whose associated value is Value.

Note that using the Process Dictionary

- Destroys referential transparency
- Makes debugging difficult
- Survives Catch/Throw

So:

- Use with care
- Do not over use - try the clean version first

Obtaining System Information

The following calls exist to access system information:

processes() returns a list of all processes currently know to the system.

process_info(Pid) returns a dictionary containing information about Pid.

Module:module_info() returns a dictionary containing information about the code in module Module.

If you use these BIFs remember:

- Use with extreme care
- Don't assume fixed positions for items in the dictionaries.

But you can do some fun things

- Writing real filthy programs, e.g. message sending by remote polling of dictionaries Why should anybody want to do this?
- Killing random processes
- Write Meta-system programs
- Poll system regularly for zombie processes
- Poll system to detect or break deadlock
- Analyse system performance

Bibliography I

Joe Armstrong.
Programming Erlang: Software for a Concurrent World.
Pragmatic Bookshelf, Raleigh, NC, 2007.