An Erlang Course

This is the content of the Erlang course. This course usually takes four days to complete. It is divided into 5 modules and has a number of programming exercises.

Module 1 - History

A short history of the Erlang language describing how Erlang was developed and why we had to invent a new language.

Module 2 - Sequential Programming

Symbolic data representation, how pattern matching is used to pack/unpack data, how functions are combined to form programs etc.

Module 3 - Concurrent Programming

Creating an Erlang process, communication between Erlang processes.

Module 4 - Error handling

Covers error handling and the design of robust systems.

Module 5 - Advanced Topics

All those tricky things like loading code in running systems, exception handling etc.

History of Erlang

1982 - 1985

Experiments with programming of telecom using > 20 different languages. Conclusion: The language must be a very high level symbolic language in order to achive 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 asyncronous 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 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:** *>> 10 times greater gains in efficency 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.

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 homgeneous 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

Numbers

Integers

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

Floats

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

- B#Val is used to store numbers in base < B &gt.

- $Char is used for ascii values (example $A instead of 65).

Back to top

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.

Back to top

Tuples

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

- Used to store a fixed number of items.
- Tuples of any size are allowed.

Back to top

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.

Back to top

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).

Back to top

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 cre- ated.
- Data structures are created by writing them down (no explicit memory alloca- tion or deallocation is needed etc.).
- Data structures may contain bound vari- ables.

Back to top

Pattern Matching

```
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
```

Back to top

Pattern Matching (Cont)

```
[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.

Back to top

Function Calls

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

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

- Arg1 .. Argn are any Erlang data struc- tures.
- 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.

Back to top

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).

Back to top

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.

Back to top

Built 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.

Back to top

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.

Back to top

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}.
```

Back to top

Evaluation example

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

> 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
```

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

Back to top

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!!)

Back to top

Examples of Guards

```
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)
```

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

Back to top

Traversing Lists

```
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.

Two other common patterns:

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

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

Back to top

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 .

Back to top

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.

Back to top

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.

Back to top

Special Forms

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

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

- Not really needed - but useful.

Back to top

Concurrent Programming

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.
- **Timeout** - 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.

back to top

Creating a New Process

Before:

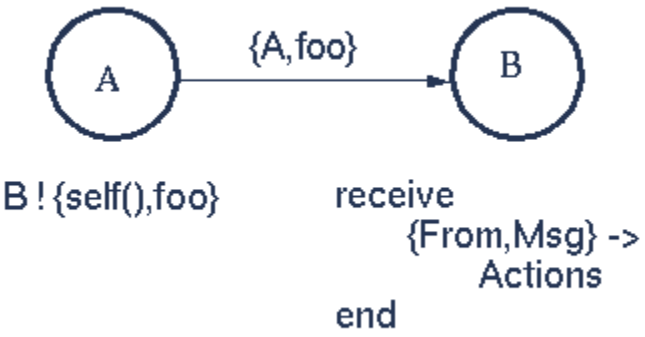
Code in Pid1

Pid2 = spawn(Mod, Func, Args)
After

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

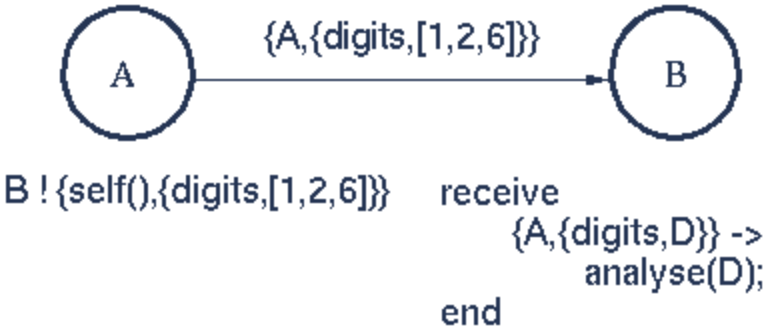
back to top

Simple Message Passing



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.



- 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.

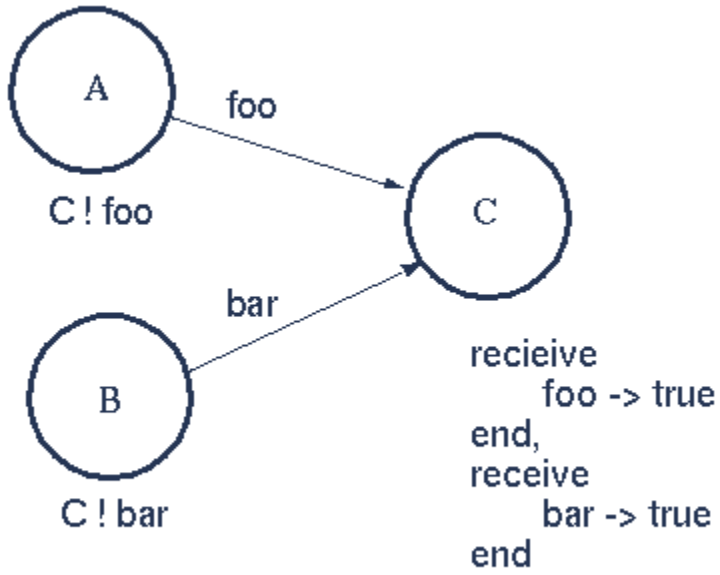
back to top

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.
```

back to top

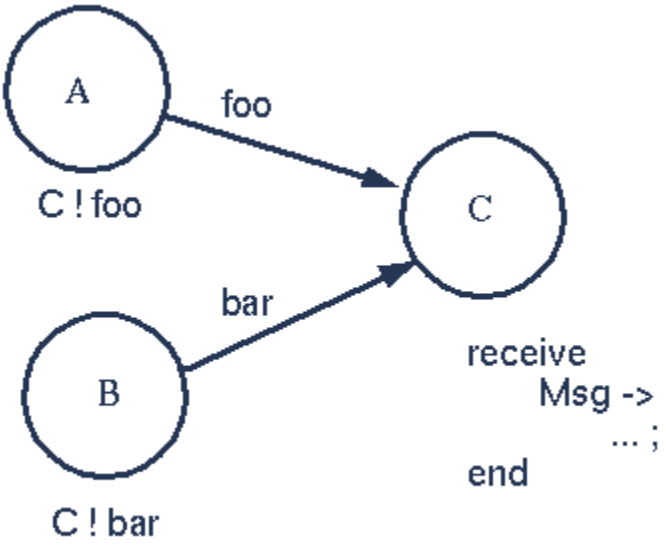
Selective Message Reception



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

back to top

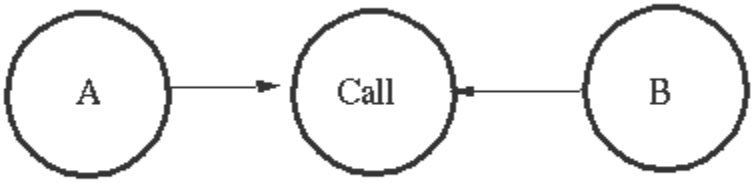
Selection of any message



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.

back to top

A Telephony Example

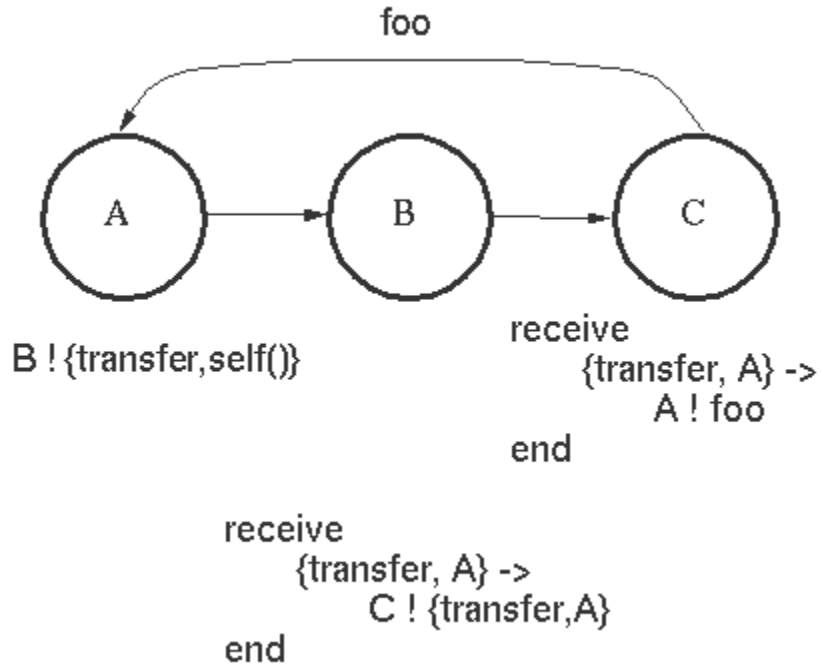



```
ringing_a(A, B) ->
        receive
                {A, on_hook} ->
                        A ! {stop_tone, ring},
                        B ! terminate,
                        idle(A);
                {B, answered} ->
                        A ! {stop_tone, ring},
                        switch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

back to top

Pids can be sent in messages



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

back to top

Registered Processes

register(Alias, Pid) Registers the process **Pid** with the name **Alias**.

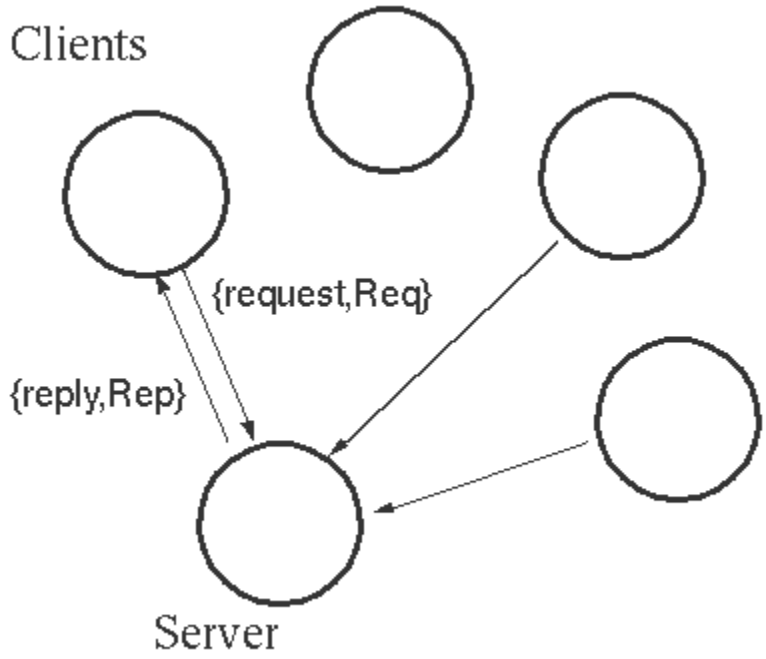
```
start() ->
        Pid = spawn(num_anal, server, [])
        register(analyser, Pid).

analyse(Seq) ->
```

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

Any process can send a message to a registered process.

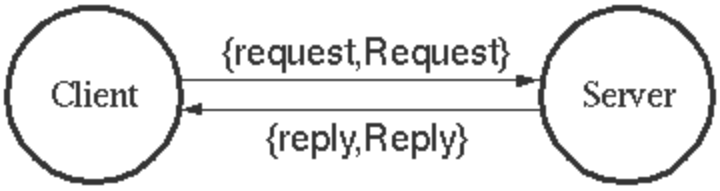
back to top

Client Server Model



Protocol

- Protocol



Server code

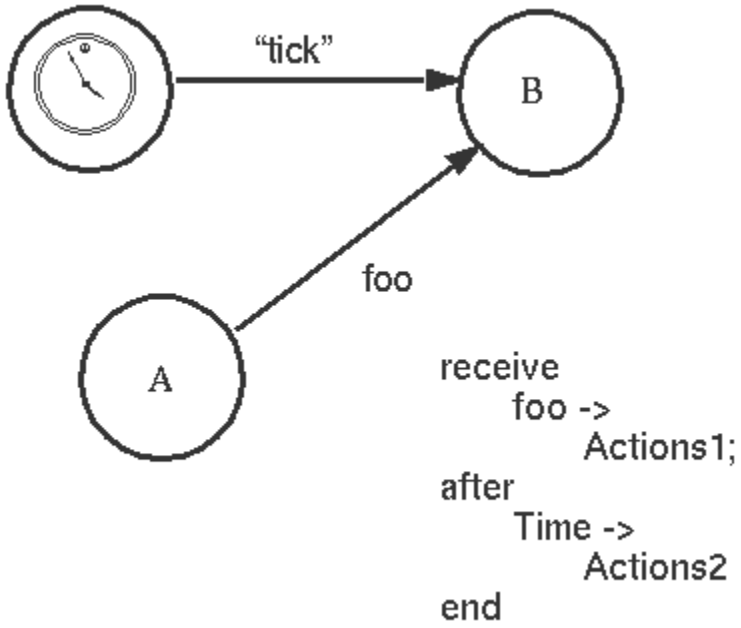
```
-module(myserver).

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.
```

back to top

Timeouts



If the message **foo** is received from **A** within the time **Time** perform **Actions1**

otherwise perform **Actions2**.

Uses of Timeouts

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 iin **T** miliseconds 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 timeout means check the message buffer first and if it is empty execute the following code.

back to top

Error Handling

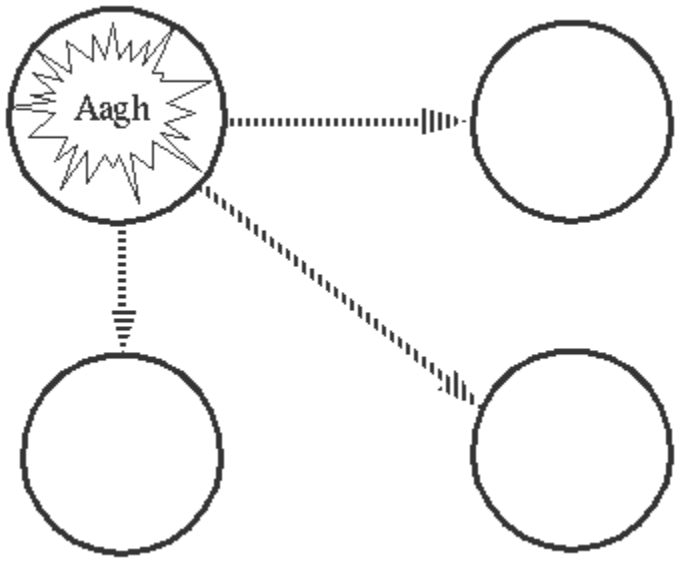
Definitions

- **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.

back to top

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.

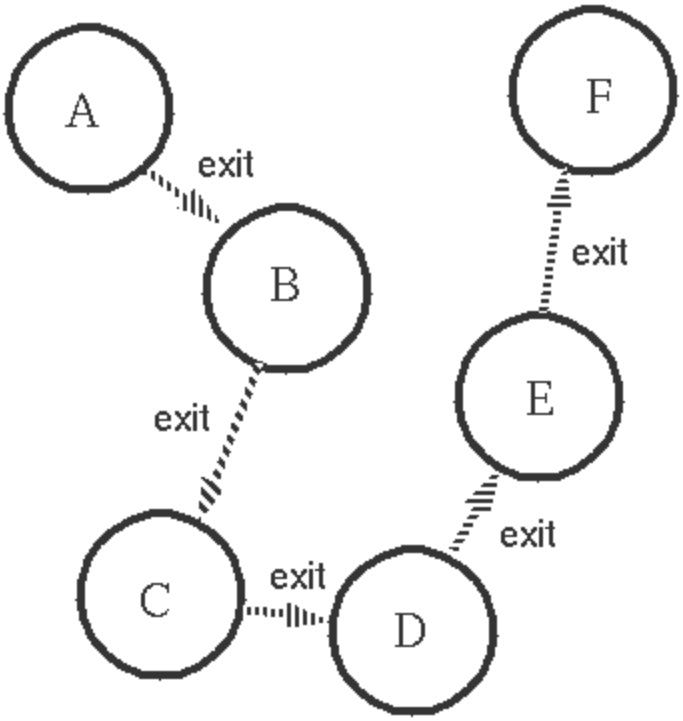


back to top

Exit Signals propagate through Links

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 propogate through the links:



These exit signals eventuall 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.*

back to top

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.

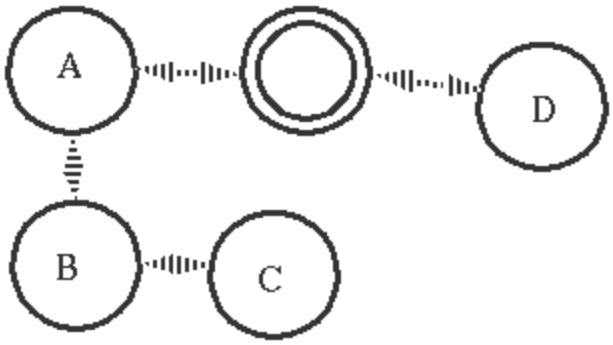
P2 has the following code:

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

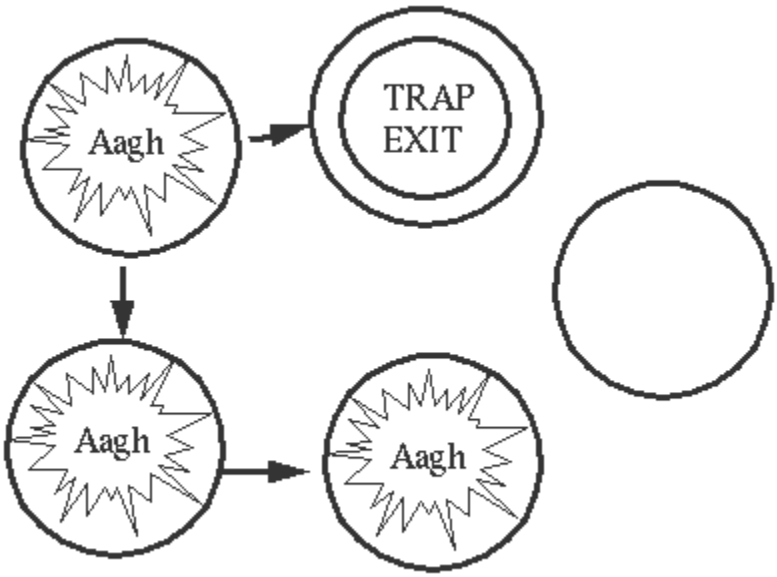
back to top

Complex Exit signal Propagation

Suppose we have the following set of processes and links:



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



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.

back to top

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.

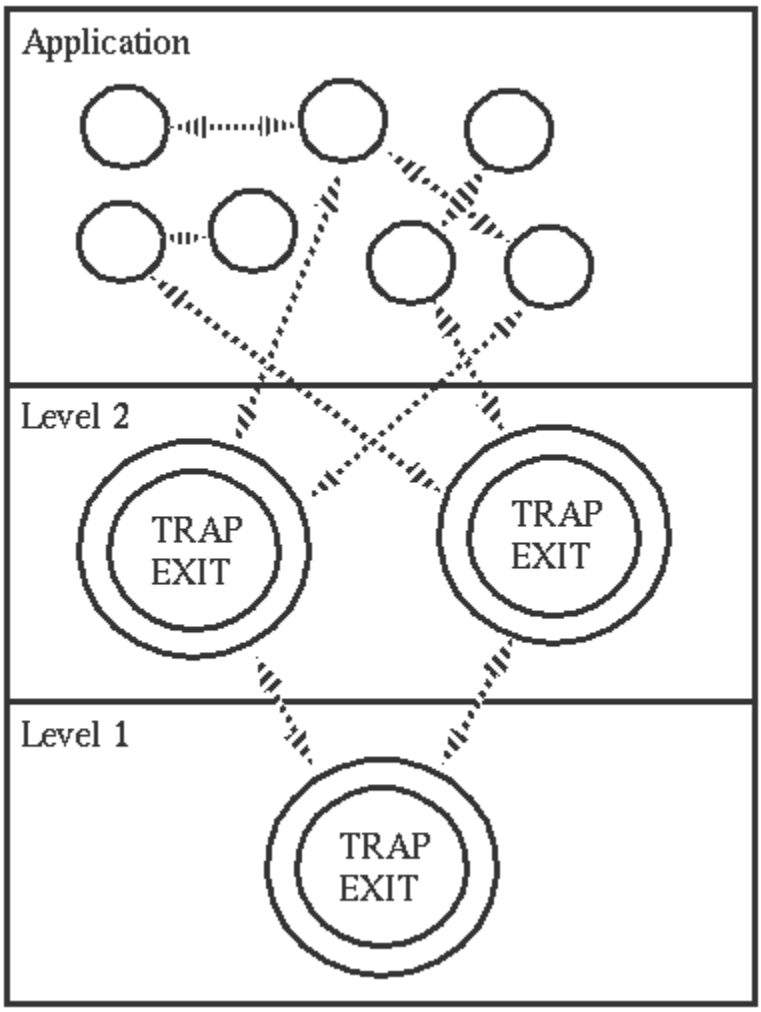
- A process which is trapping exit signals converts all incoming exit signals to conventional messages which it can receive in a receive statement.
- Errors in BIFs or pattern matching errors send automatic exit signals to the link set of the process where the error occured.

back to top

Robust Systems can be made by Layering

By building a system in layers we can make a robust system. Level1 traps and corrects errors occuring in Level2. Level2 traps and corrects errors ocuring in the application level.

In a well designed system we can arrange that application programers will not have to write any error handling code since all error handling is isolated to deper levels in the system.



back to top

Primitives For Exit Signal Handling

- **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.**

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!*

back to top

A Robust Server

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?*

```
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]).
```

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.

back to top

Allocator with Error Recovery

The following is a *reliable* server. If a client craches *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 when the resource is allocted. If an exit message comes 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 -- multiple allocation to same process. i.e. before doing the **unlink(Pid)** we should check to see that the process has not allocated more than one device.

back to top

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).
```

back to top