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.

This course is available in various formats:

- For on-line reading at http://www.erlang.org/course/course.html
- A Zip archive that contains the HTML files.
- A gzipped TAR archive that contains the HTML files.
- A PDF file.

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
  - Floats
- Atoms
- Tuples
- Lists
- Variables
- Complex Data Structures
- Pattern Matching
- Function Calls
- The Module Systems
- Starting the system
- Built in Functions (BIFs)
- Function syntax
- An example of function evaluation
- Guarded function clauses
  - Examples of Guards
- Traversing Lists
- Lists and Accumulators
- Shell commands
- Special Functions
- Special Forms

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 >.
- $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
- Creating a new process
- Simple message passing
- An Echo Process
- Selective Message Reception
- Selection of Any Message
- A Telephony Example
- Pids can be sent in messages
- Registered Processes
- The Client Server Model
- Timeouts

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:

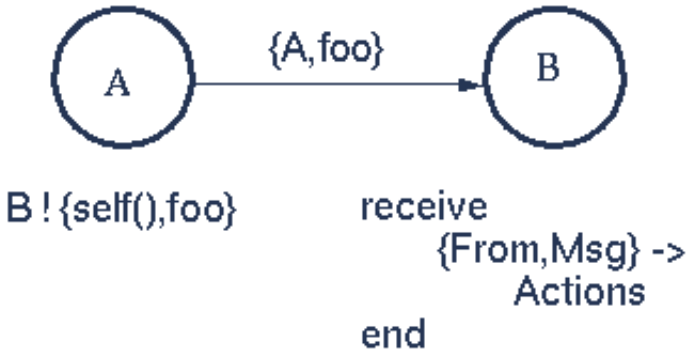
Code in Pid1

Pid2 = spawn(Mod, Func, Args)
After

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

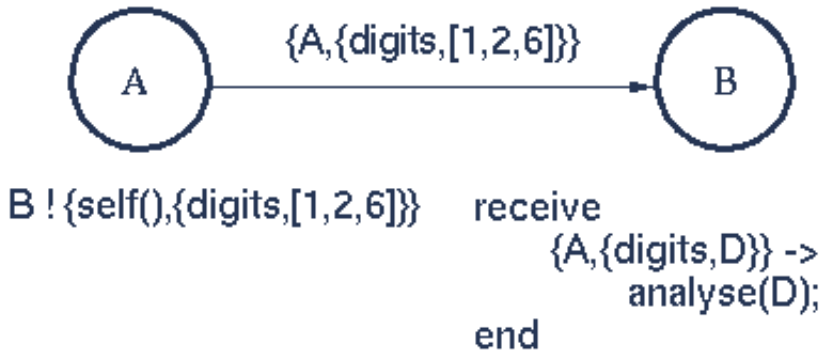
back to top

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

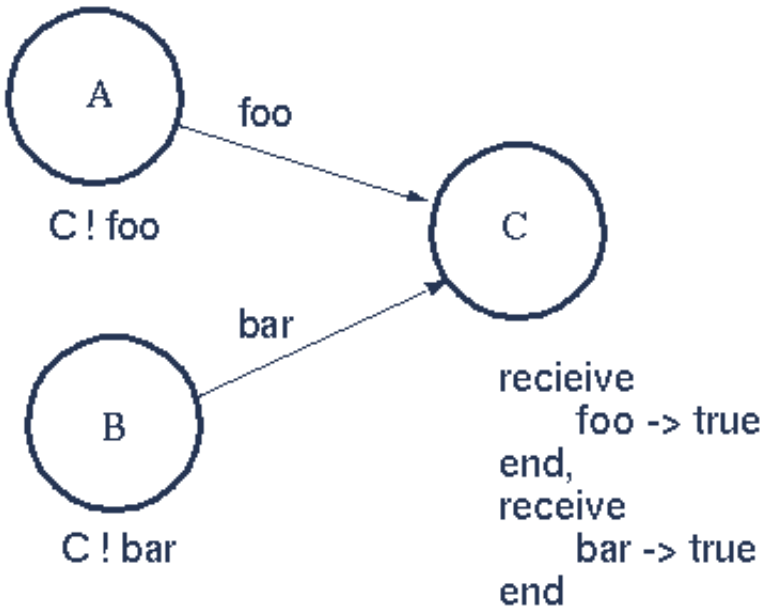
```
-module(echo).
-export([go/0, loop/0]).

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

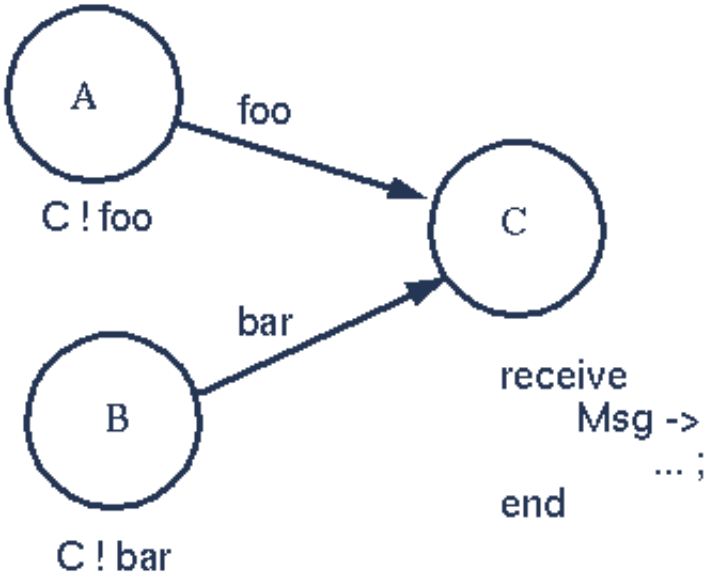
Selective Message Reception



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

back to top

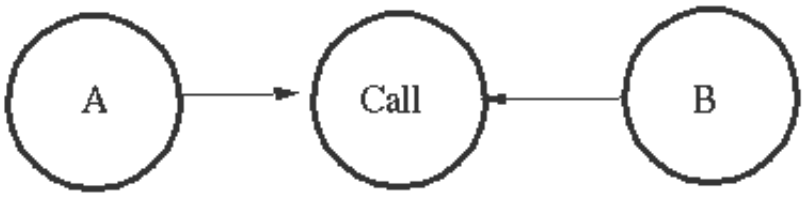
Selection of any message



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

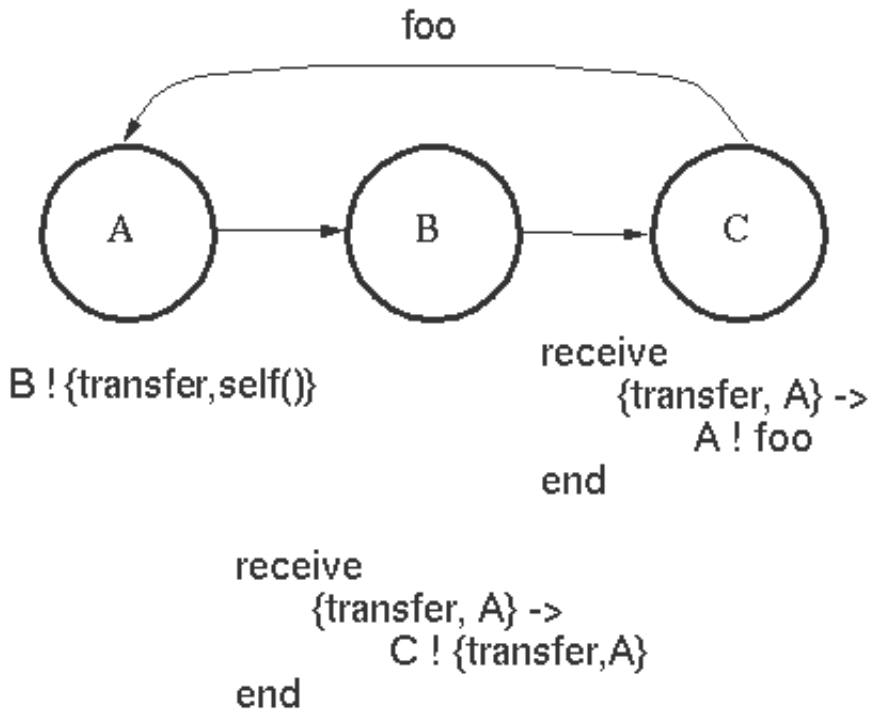



```
ringing_a(A, B) ->
        receive
                {A, on_hook} ->
                        A ! {stop_tone, ring},
                        B ! terminate,
                        idle(A);
                {B, answered} ->
                        A ! {stop_tone, ring},
                        switch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

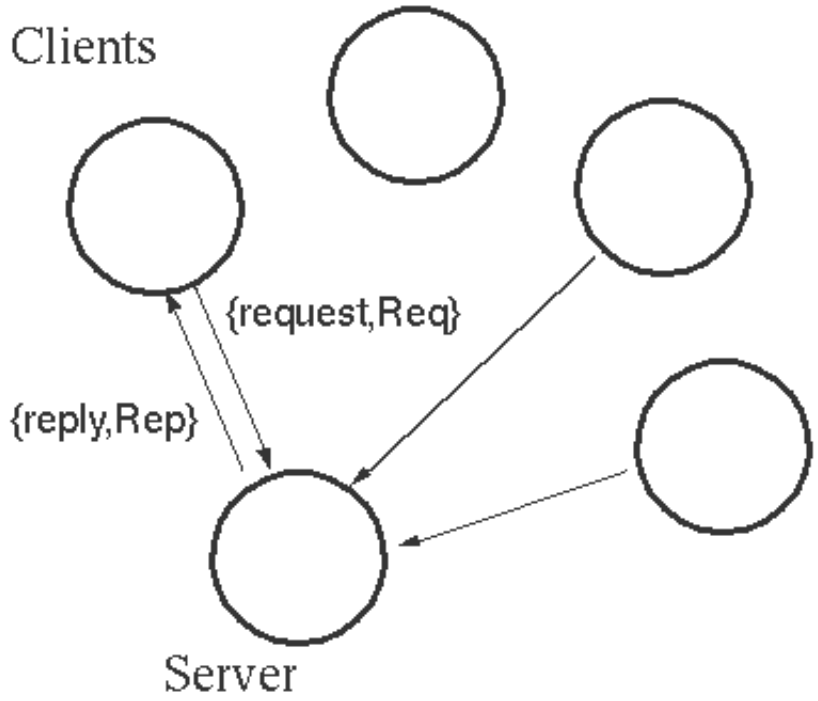
```
start() ->
        Pid = spawn(num_anal, server, [])
        register(analyser, Pid).

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.
```

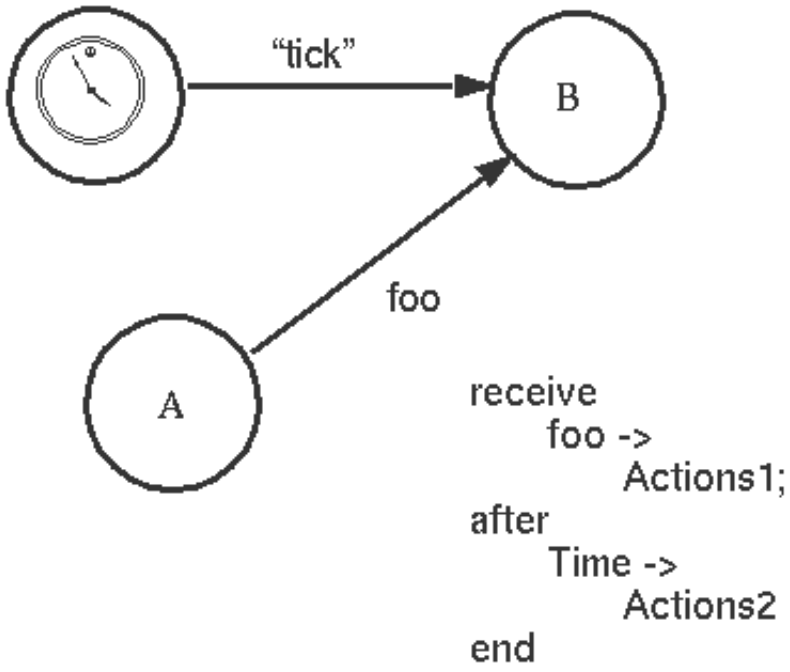
Interface Library

```
-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

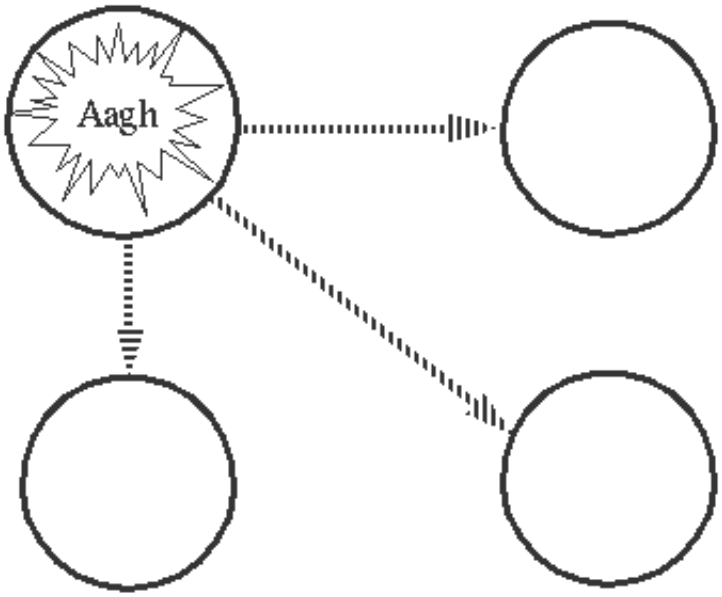
Definitions

- **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.

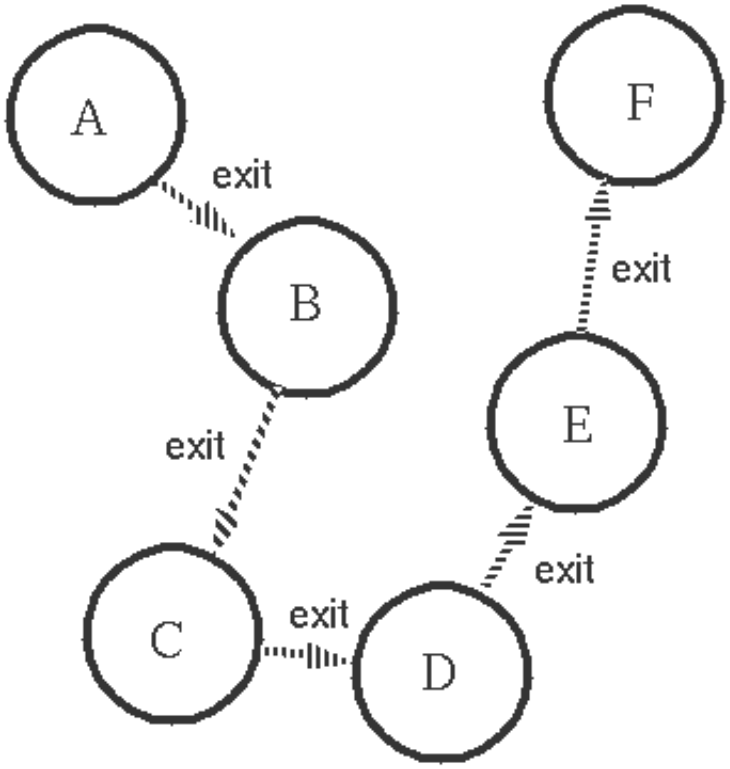


back to top

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:



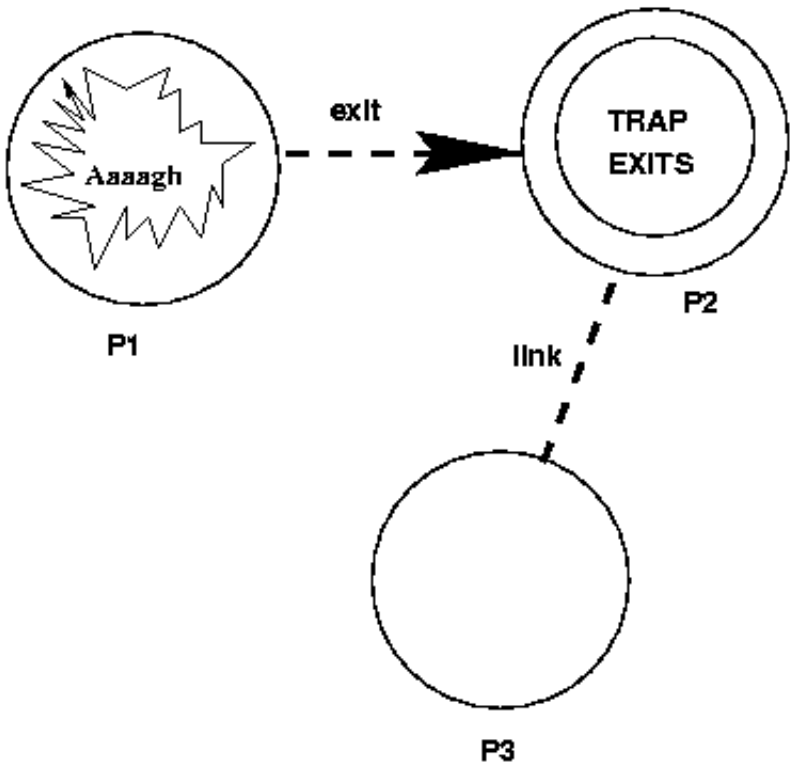
These exit signals eventuall reach all the processes which are linked together.

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.



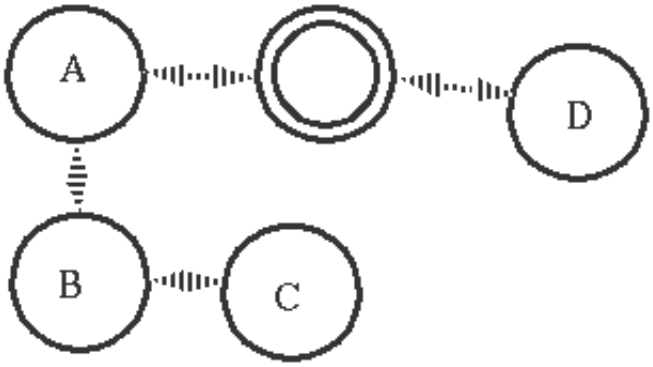
P2 has the following code:

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

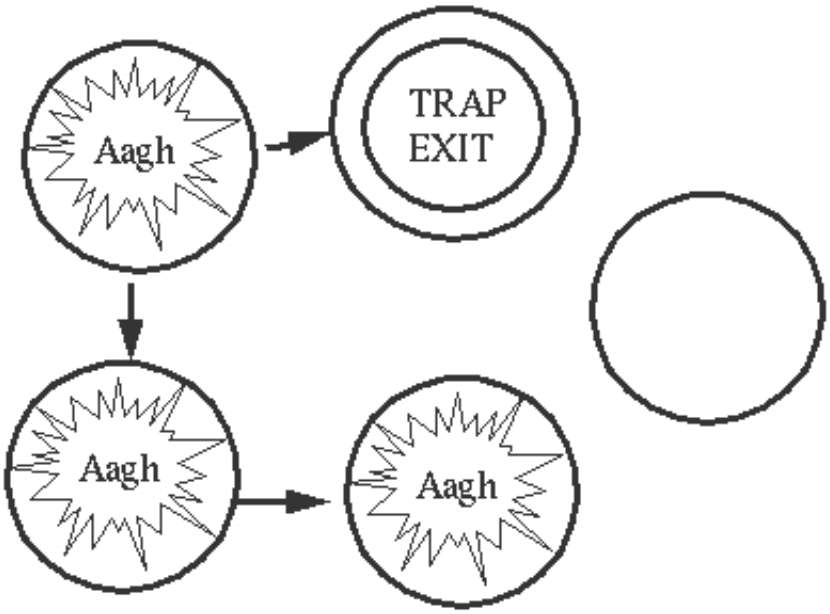
back to top

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.

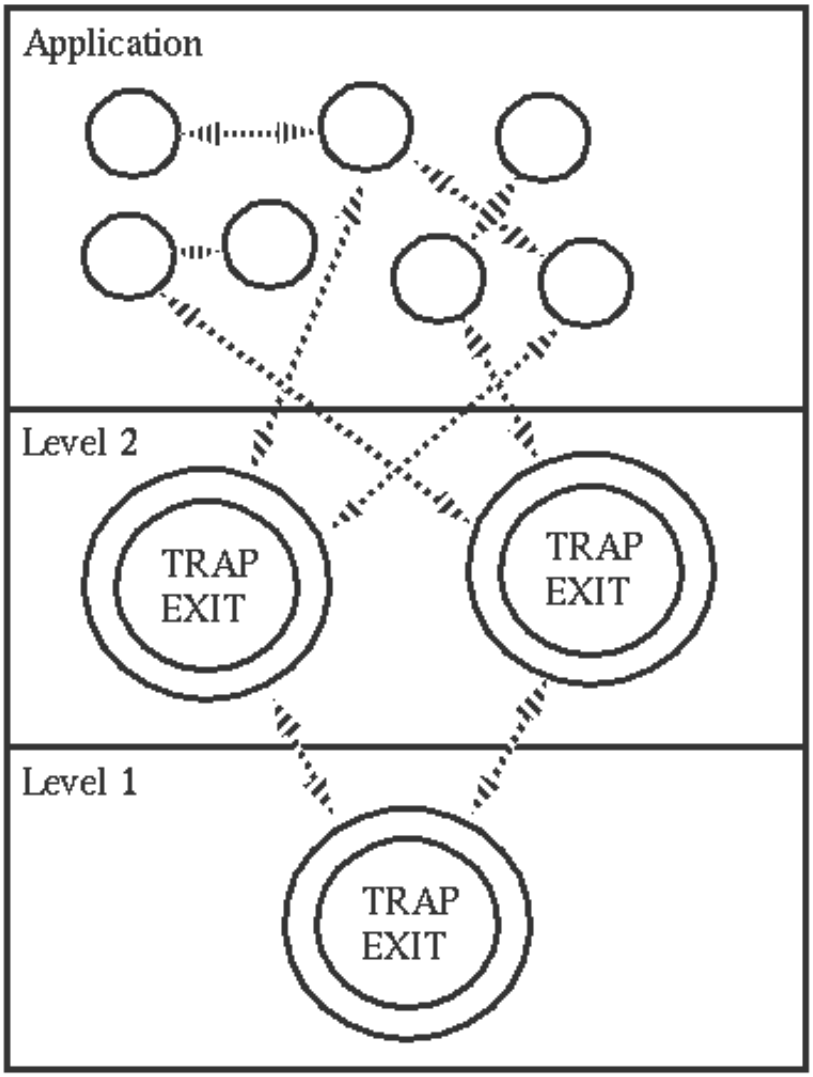
- Errors in BIFs or pattern matching errors send automatic exit signals to the link set of the process where the error occured.

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

Advanced Topics

- Scope of variables
- Catch/throw
- Use of Catch and Throw
- The module error_handler
- The Code loading mechanism
- Ports
- Port Protocols
- Binaries
- References
- Space saving optimisations
- Last Call Optimisation
- Process Dictionary
- Obtaining System Information

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),
    ...
```

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 an error depends upon what happens after the case function. In this example it is an error, 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.

back to top

Catch and Throw

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.

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}**.

back to top

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} ->
            ... error in BIF ....
            ........ BUG...........
        {exception1, Args} ->
            ... planned exception ....
        Normal ->
            .... normal case ....
    end.

func(X) ->
    ...

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

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

back to top

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*

back to top

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.

back to top

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:

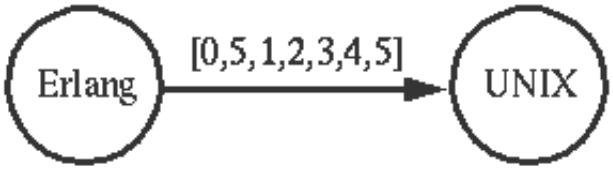
Port = open_port ({spawn,Process} , {packet,2})

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.

back to top

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.



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

back to top

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

back to top

References

References are erlang objects with exactly two properties:

- They can be created by a program (using **make_ref/0**), and,
- 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, for example:

```
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.
```

back to top

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 canot Evaluate '+' until both its arguments are known. This formulation of sum(X) evaluates in space O(length(X)).

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.

back to top

Last Call Optimisation

The last call optimisation **must** be used in persistant servers.

For example:

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

back to top

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 referencial transparency
- Makes debugging difficult
- Survives Catch/Throw

So:

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

back to top

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 dic tionary 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 like:

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

back to top

Erlang Programming Exercises

Entering a program

Type the **demo:double** example into a file called **demo.erl**. Use your favourite text editor.

Start Erlang.

Give the command **c:c(demo)**. to compile the file.

Try running the query:

```
demo:double(12).
```

This is just to test if you can get the system started and can use the editor together with the Erlang system.

Back to top

Simple sequential programs

1. Write functions **temp:f2c(F)** and **temp:c2f(C)** which convert between centigrade and Fahrenheit scales. (hint 5(F-32) = 9C)

2. Write a function **temp:convert(Temperature)** which combines the functionality of **f2c** and **c2f**. Example:

```
> temp:convert({c,100}).
=> {f,212}
> temp:convert({f,32}).
=> {c,0}
```

3. Write a function **mathStuff:perimeter(Form)** which computes the perimeter of different forms. **Form** can be one of:

```
{square,Side}
{circle,Radius}
{triangle,A,B,C}
```

Back to top

Simple recursive programs

1. Write a function **lists1:min(L)** which returns the mini- mum element of the list **L**.

2. Write a function **lists1:max(L)** which returns the maximum element of the list **L**.

3. Write a function **lists1:min_max(L)** which returns a tuple containing the min and max of the list **L**.

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

4. Write the function **time:swedish_date()** which returns an atom containing the date in swedish YYMMDD format:

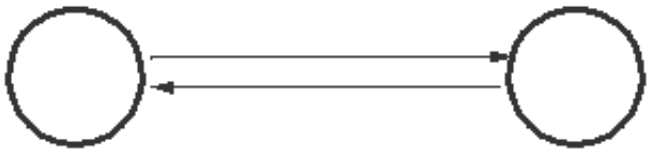
```
> time:swedish_date()
'901114'
```

*Hints: trying looking up **date()** and **time()** in the manual, You may also need **number_to_list/1**, **list_to_atom**/1. Try giving the shell queries to see what these BIF's do.)*

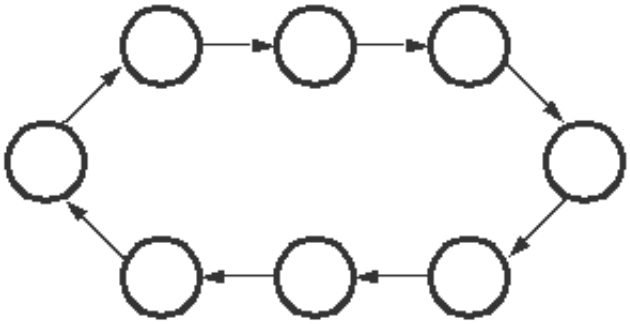
Back to top

Interaction between processes, Concurrency

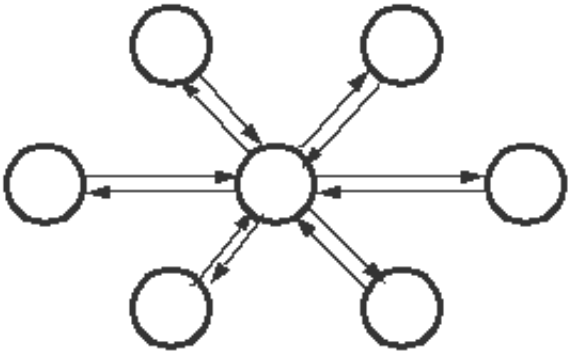
1. Write a function which starts 2 processes, and sends a message M times forewards and backwards between them. After the messages have been sent the processes should terminate gracefully.



2) Write a function which starts N processes in a ring, and sends a message M times around all the processes in the ring. After the messages have been sent the processes should terminate gracefully.



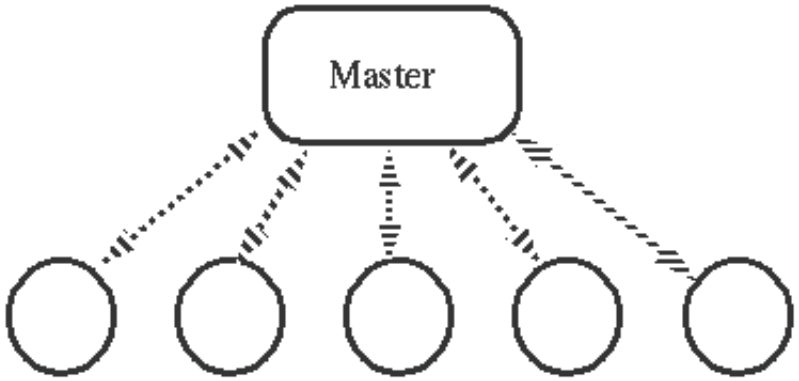
3) Write a function which starts N processes in a star, and sends a message to each of them M times. After the messages have been sent the processes should terminate gracefully.



Back to top

Master and Slaves, error handling

This problem illustrates a situation where we have a process (the master) which supervises other processes (the slaves). In a real example the slave could, for example, be controlling different hardware units. The master's job is to ensure that all the slave processes are alive. If a slave crashes (maybe because of a software fault), the master is to recreate the failed slave.



Write a module **ms** with the following interface:

start(N) - Start the master and tell it to start **N** slave proc- esses. Register the master as the registered process **master**.

to_slave(Message, N) - Send a message to the master and tell it to relay the message to slave **N**. The slave should exit (and be restarted by the master) if the message is **die**.

The master should detect the fact that a slave processe diea nd restart it and print a message that it has done so.

The slave should print all messages it recieves except the message **die**

Hints:
The master should trap exit messages and create links to all the slave processes.

The master should keep a list of the process id's (pid's) of the slave processes and their associated numbers.

Example:

```
> ms:start(4).
=> true
> ms:to_slave(hello, 2).
=> {hello,2}
Slave 2 got message hello
> ms:to_slave(die, 3).
=> {die,3}
master restarting dead slave3
```

Back to top

Robustness in Erlang, and use of a graphics package

A robust system makes it possible to survive partial failure, i.e if some parts of the system crashes, it should be possible to recover instead of having a total system crash. In this exercise we will build a tree-like hierachy of processes that can recover if any of the tree-branches should crash.

To illustrate this we are going to use the Interviews interface and having each process represented by a window on the screen.

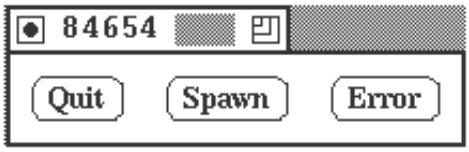
Excercise: Create a window containing three buttons: **Quit , Spawn , Error**.

The **Spawn** button shall create a child process which displays an identical window.

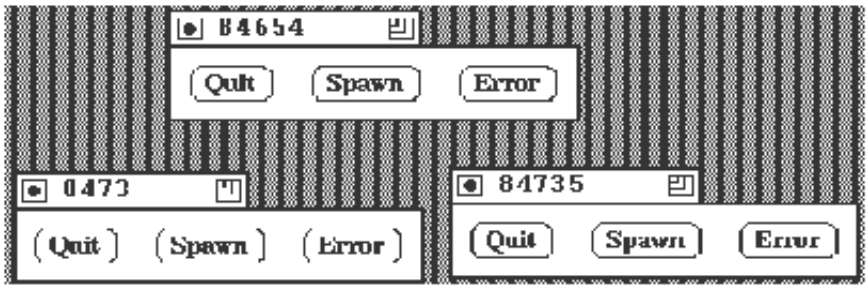
The **Quit** button should kill the window and its child windows.

The **Error** button should cause a runtime error that kills the window (and its children), this window shall then be restarted by its parent.

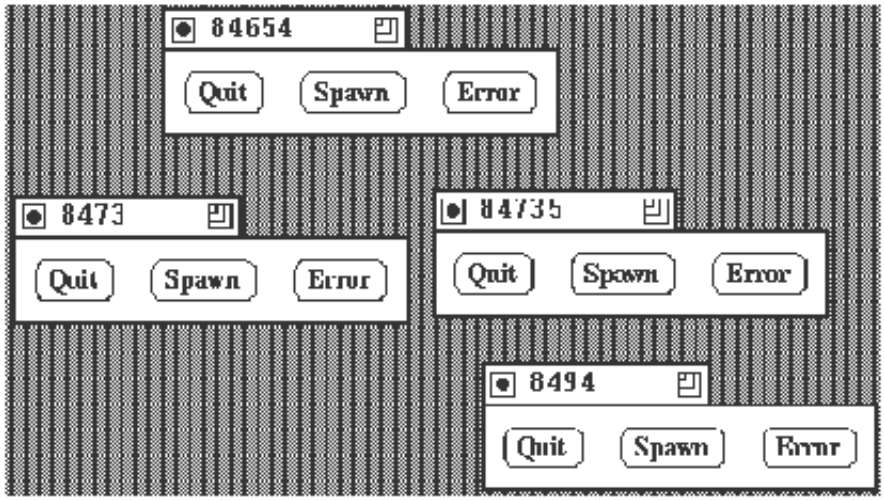
Example: When we start our program, a window like this should appear:



Let us now press the **Spawn** button twice and as result, two child windows will pop up on our screen. Our screen will now look something like this:



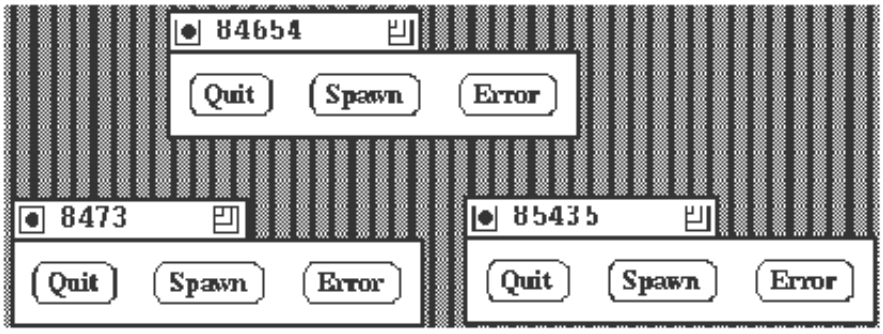
As you can see the windows are tagged with a number so it is easier for us to refer to them. The parent window has the number 84654 and its two childs number 8473 and 84735. Each child may have childs of its own, e.g press the **Spawn** button on window 84735 and we will have the following picture:



The window 8494 is a child to window 84735.

Now lets press the **Error** button in window 84735. Now a runtime error will occur in the process for that window and the process (and the window) will die. This shall cause the child 8494 also to die and the parent 84654 to start up a new child window.

The result will look something like this:



The new child got number 85435.

Back to top

Erlang using UNIX sockets

Do you want to talk with a friend on another machine? Shouldn’t it be nice to have a shell connected to your friend and transfer messages in between?

This can be implemented using the client/server concept with a process on each side listening to a socket for messages.

- Write a distributed (client/server) message passing system. The system shall be built upon the Erlang interface to the BSD unix sockets.
- The server host name will be given as input argument in order to start the client.
- A prompt shall be displayed both on the server and the client side. The user shall give a string followed by a RETURN as a message. The message will be transfered and displayed to the user on the other side.
- An empty input string (on either side) will end the session.

*(Hints: read the man-page for the socket interface, also in order to read the command line, use **io:get_line/1**)*

Back to top

The use of open_port/1

Use the **open_port({spawn,Name})** BIF in order to start an external UNIX process.

- The Erlang program shall open a port, sending some data through the port, have the data echoed back and then printout the received data.
- The C program that will run in the UNIX process shall consist of an eternal loop starting with a read from file descriptor 0 (stdin) and end with a write on file descriptor 1 (stdout) before it iterates again.

Note: The first two bytes read in the C program contains the length in bytes of the data to follow *(Make sure that you are reading as much as the length indicates).*

(Hints: read the User's Guide p.37 and the BIF Guide p.11)

Back to top

Socket comunication between Erlang and C

Write Erlang and C programs which no the following:

- The Erlang program shall create a socket, waiting for accept, sending some data through the socket, have the data echoed back, printout the received data, and then close the socket.
- The C program that will run in the UNIX process shall take the hostname of the host you are going to commu- nicate with as a parameter. Set up a socket to that host and then echo data as in the previous exercise.

(Hints: Include the files listed below. The main routine will look much the same as in the previous example. The code to setup the socket is found on the next page

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <errno.h>
```

```
/*--- Setup a socket to Host using Port , return the filedesc. ---*/

static int setup_socket( hostname , port )
     char *hostname;
     int port;
{
  struct sockaddr_in serv_addr;
  int sockfd;
  struct hostent *hp;

  /*--- Get the address of the host ---*/

  if ((hp = gethostbyname( hostname )) == (struct hostent*) 0) {
    perror("From gethostbyname \n");
    exit(-1);
  }

  /*--- Fill in the address to the remote system ---*/

  bzero((char *) &serv_addr , sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;             /* Protocol family */
  serv_addr.sin_port   = htons( port );       /* The port number */
  bcopy(hp->h_addr_list[0] ,                  /* The net address to the host */
         (char *) &serv_addr.sin_addr.s_addr ,
         hp->h_length);

  /*--- Create the socket ---*/

  if ( (sockfd = socket( AF_INET , SOCK_STREAM , 0 )) < 0) {
    perror("setup_socket: socket");
    exit(-1);
  }

  /*--- Connect to the other system ---*/

  if (connect(sockfd, &serv_addr, sizeof(serv_addr)) < 0) {
    perror("setup_socket: connect");
    exit(-1);
  }
  else
    return sockfd;

}; /* setup_socket */
```

Back to top

Implementing Talk with Distributed Erlang

Make a simple Talk program that makes it possible to chat with friends at other nodes/hosts.

- First you and your friend must create identical **.erlang.cookie** files, e.g:

  ```
  echo -n "dh32d8yhd8" > ~/.erlang.cookie
  ```

- Make sure nobody else can read it:

```
chmod 400 ~/.erlang/cookie ).
```

- Then start a distributed Erlang node:

```
erl -name bill -cookie
```

- Now start your program. It should begin with promting for the other node name, and then promting for messages that will be sent to your friend for each carriage-return.

Hints: Your program should consist of two registered processes one for reading from the terminal and the other one for recieving messages from the other node then writing them to the terminal.

Back to top

Generating a parser for Datalog

By using the Yecc parser generator we will generate a parser that will accept Datalog programs according to the specified grammar below. We will also have to use/modify a scanner (lexical analyser) to suit our purpose.

The syntax for Datalog can be described by the following grammar:

```
PGM -> e
PGM -> CLAUSE PGM
CLAUSE -> LITERAL TAIL .
LITERAL -> predsym PARLIST
PARLIST -> e
PARLIST -> ( ARGLIST )
ARGLIST -> TERM ARGTAIL
ARGTAIL -> e
ARGTAIL -> , ARGLIST
TERM -> csym
TERM -> varsym
TAIL -> e
TAIL -> :- LITLIST
LITLIST -> LITERAL LITTAIL
LITTAIL -> e
LITTAIL -> , LITLIST
```

This grammar will accept Datalog programs, for example:

```
path(stockholm,uppsala).
```

or:

```
route(X,Y) :- path(X,Z),path(Z,Y).
```

The tokens produced by the scanner are defined as:

```
predsym = lc(lc + uc +digit)* | digit*
csym = lc(lc + uc + digit)* | digit*
varsym = uc(lc + uc + digit)*
lc = any lowercase letter
uc = any uppercase letter
digit = any digit
```

To be able to solve this exercise you will have to read the man-page for Yecc (**erl -man yecc**). Good

Luck !!

Back to top