Erlang: Sequential & Concurrent

Part 1 – Sequential Erlang

Erlang Buzzwords

- Functional (strict)
- Single-assignment
- Dynamically typed
- Concurrent
- Distributed
- Message passing
- Soft real-time
- Fault tolerant
- No sharing
- Automatic memory management (GC)
- Virtual Machine (BEAM)
- Native code (HiPE)
- Dynamic code loading
- Hot-swapping code
- Multiprocessor support
- OTP (Open Telecom Platform) libraries
- Open source

Background

- Developed by Ericsson, Sweden
  - Experiments 1982-1986 with existing languages
    - Higher productivity, fewer errors
    - Suitable for writing (large) telecom applications
    - Must handle concurrency and error recovery
  - No good match - decided to make their own
    - 1986-1987: First experiments with own language
    - Erlang (after Danish mathematician A. K. Erlang)
    - 1988-1989: Internal use
    - 1990-1998: Erlang sold as a product by Ericsson
  - Open Source (MPL-based license) since 1998
    - Development still done by Ericsson

Erlang at Uppsala University

- High Performance Erlang (HiPE) research group
  - Native code compiler (SPARC, x86, x86_64, PowerPC, PowerPC-64, ARM)
  - Program analysis and optimization
  - Runtime system improvements
  - Language development and extensions
  - Programming and static analysis tools
- Most results from the HiPE project have been included in the official Erlang distribution

Hello, World!

```
%% File: hello.erl

-module(hello).
-export([run/0]).

run() -> io:format("Hello, World!\n").
```

- '%' starts a comment
- '.' ends each declaration
- Every function must be in a module
  - One module per source file
  - Source file name is module name + ".erl"
- ':' used for calling functions in other modules

Running Erlang

```
$ erl
Erlang (BEAM) emulator version 5.5.1

Eshell V5.5.1 (abort with ^G)
1> 6*7.
42
2> halt().
$
```

- The Erlang VM emulator is called 'erl'
- The interactive shell lets you write any Erlang expressions and run them (must end with '.')
- The "1>", "2>", etc. is the shell input prompt
- The "halt()" function call exits the emulator

Compiling a module

```
$ erl
Erlang (BEAM) emulator version 5.5.1

Eshell V5.5.1 (abort with ^G)
1> c(hello).
{ok,hello}
2>
```

- The "c(Module)" built-in shell function compiles a module and loads it into the system
  - If you change something and do "c(Module)" again, the new version of the module will replace the old
- There is also a standalone compiler called "erlc"
  - Running "erlc hello.erl" creates "hello.beam"
  - Can be used in a normal Makefile

Running a program

```
Eshell V5.5.1 (abort with ^G)
1> c(hello).
{ok,hello}
2> hello:run().
Hello, World!
ok
3>
```

- Compile all your modules
- Call the exported function that you want to run, using "module:function(...)."
- The final value is always printed in the shell
  - "ok" is the return value from io:format(...)

A recursive function

```
-module(factorial).
-export([fac/1]).

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

- Variables start with upper-case characters!
- ';' separates function clauses
- Variables are local to the function clause
- Pattern matching and guards to select clauses
- Run-time error if no clause matches (e.g., N < 0)
- Run-time error if N is not an integer

Tail recursion with accumulator

```
-module(factorial).
-export([fac/1]).

fac(N) -> fac(N, 1).

fac(N, Product) when N > 0 ->
  fac(N-1, Product*N);
fac(0, Product) ->
  Product.
```

- The *arity is part of the function name:* `fac/1≠fac/2`
- Non-exported functions are local to the module
- Function definitions cannot be nested (as in C)
- Last call optimization: the stack does not grow if the result is the value of another function call

Recursion over lists

```
-module(list).
-export([last/1]).

last([Element]) -> Element;
last([_|Rest]) -> last(Rest).
```

- Pattern matching selects components of the data
- "_" is a "don't care"-pattern (not a variable)
- "`[Head|Tail]`" is the syntax for a single list cell
- "`[]`" is the empty list (often called "nil")
- "`[X,Y,Z]`" is a list with exactly three elements
- "`[X,Y,Z|Tail]`" has three or more elements

List recursion with accumulator

```
-module(list).
-export([reverse/1]).

reverse(List) -> reverse(List, []).

reverse([Head|Tail], Acc) ->
  reverse(Tail, [Head|Acc]);
reverse([], Acc) ->
  Acc.
```

- The same syntax is used to *construct lists*
- Strings are simply lists of Unicode characters
  - `"Hello" = [$H, $e, $l, $l, $o] = [72,101,...]`
  - `"" = []`

Numbers

```
12345
-9876
16#ffff
2#010101
$A
0.0
3.1415926
6.023e+23
```

- Arbitrary-size integers (but usually just one word)
- #-notation for base-N integers
- $-notation for character codes (ISO-8859-1)
- Normal floating-point numbers (standard syntax)
  - cannot start with just a '.', as in e.g. C

Atoms

```
true                    % boolean
false                   % boolean
ok                      % used as "void" value
hello_world
doNotUseCamelCaseInAtoms
'This is also an atom'
'foo@bar.baz'
```

- Must start with lower-case character or be quoted
- Single-quotes are used to create arbitrary atoms
- Similar to hashed strings
  - Use only one word of data (just like a small integer)
  - Constant-time equality test (e.g., in pattern matching)
  - At run-time: `atom_to_list(Atom)`, `list_to_atom(List)`

Tuples

```
{}
{42}
{1,2,3,4}
{movie, "Yojimbo", 1961, "Kurosawa"}
{foo, {bar, X},
      {baz, Y},
      [1,2,3,4,5]}
```

- Tuples are the main data constructor in Erlang
- A tuple whose 1st element is an atom is called a *tagged tuple* - this is used like constructors in ML
  - Just a convention – but almost all code uses this
- The elements of a tuple can be any values
- At run-time: `tuple_to_list(Tup)`, `list_to_tuple(List)`

Other data types

- Functions
  - Anonymous and other
- Bit streams
  - Sequences of bits
  - `<<0,1,2,...,255>>`
- Process identifiers
  - Usually called 'Pids'
- References
  - Unique "cookies"
  - `R = make_ref()`
- No separate booleans
  - atoms `true/false`
- Erlang values in general are often called "terms"
- All terms are ordered and can be compared with <, >, ==, =:=, etc.

Type tests and conversions

```
is_integer(X)
is_float(X)
is_number(X)
is_atom(X)
is_tuple(X)
is_pid(X)
is_reference(X)
is_function(X)
is_list(X)   % [] or [_|_]

atom_to_list(A)
list_to_tuple(L)
binary_to_list(B)

term_to_binary(X)
binary_to_term(B)
```

- Note that `is_list` only looks at the first cell of the list, not the rest
- A list cell whose tail is not another list cell or an empty list is called an "improper list".
  - Avoid creating them!
- Some conversion functions are just for debugging: avoid!
  - `pid_to_list(Pid)`

Built-in functions (BIFs)

```
length(List)
tuple_size(Tuple)
element(N, Tuple)
setelement(N, Tuple, Val)

abs(N)
round(N)
trunc(N)

throw(Term)
halt()

time()
date()
now()

self()
spawn(Function)
exit(Term)
```

- Implemented in C
- All the type tests and conversions are BIFs
- Most BIFs (not all) are in the module "`erlang`"
- Many common BIFs are auto-imported (recognized without writing "`erlang:...`")
- Operators (+,-,*,/,...) are also really BIFs

Standard Libraries

- Application Libraries
  - kernel
    - erlang
    - code
    - file, filelib
    - inet
    - os
  - stdlib
    - lists
    - dict, ordict
    - sets, gb_sets
    - gb_trees
    - ets, dets
- Written in Erlang
- "Applications" are groups of modules
  - Libraries
  - Application programs
    - Servers/daemons
    - Tools
    - GUI system (gs, wx)

Expressions

```
%% the usual operators
(X + Y) / -Z * 10 – 1

%% boolean
X and not Y or (Z xor W)
(X andalso Y) orelse Z

%% bitwise operators
((X bor Y) band 15) bsl 2

%% comparisons
X /= Y        % not !=
X =< Y        % not <=

%% list operators
List1 ++ List2
```

- Boolean and/or/xor are *strict* (always evaluate both arguments)
- Use `andalso`/`orelse` for short-circuit evaluation
- "`=:=`" for equality, not "`=`"
- We can always use parentheses when not absolutely certain about the precedence

Fun expressions

```
F1 = fun () -> 42 end
42 = F1()

F2 = fun (X) -> X + 1 end
42 = F2(41)

F3 = fun (X, Y) ->
       {X, Y, F1}
     end

F4 = fun ({foo, X}, Y) ->
           X + Y;
         ({bar, X}, Y) ->
           X - Y;
         (_, Y) ->
           Y
     end

F5 = fun f/3

F6 = fun mod:f/3
```

- Anonymous functions (lambda expressions)
  - Usually called "funs"
- Can have several clauses
- All variables in the patterns are *new*
  - *All variable bindings in the fun are local*
  - *Variables bound in the environment can be used in the fun-body*

Pattern matching with '='

```
Tuple = {foo, 42, "hello"},
{X, Y, Z} = Tuple,

List = [5, 5, 5, 4, 3, 2, 1],
[A, A | Rest] = List,

Struct = {foo, [5,6,7,8], {17, 42}},
{foo, [A|Tail], {N, Y}} = Struct
```

- Match failure causes runtime error (badmatch)
- Successful matching binds the variables
  - But only if they are not already bound to a value!
  - Previously bound variables can be used in patterns
  - A new variable can also be repeated in a pattern

Case switches

```
case List of
  [X|Xs] when X >= 0 ->
    X + f(Xs);
  [_X|Xs] ->
    f(Xs);
  [] ->
    0;
  _ ->
    throw(error)
end

%% boolean switch:
case Bool of
  true -> ...;
  false -> ...
end
```

- Any number of clauses
- Patterns and guards, just as in functions
- ";" separates clauses
- Use "_" as catch-all
- Variables may also begin with underscore
  - Signals "I don't intend to use this value"
  - Compiler won't warn if variable is not used

If switches and guard details

```
if
   X >= 0, X < 256 ->
     X + f(Xs);
   true ->
     f(Xs)
end
```

- Like a case switch without the patterns and the "`when`" keyword
- Use "`true`" as catch-all
- Guards are special
  - Comma-separated list
  - Only specific built-in functions (and all operators)
  - No side effects

List comprehensions

```
%% map
[f(X) || X <- List]

%% filter
[X || X <- Xs, X > 0]


%% quicksort example
qsort([P|Xs]) ->
  qsort([X || X <- Xs,
              X < P])
  ++ [P]  % pivot element
  ++ qsort([X || X <- Xs,
                 X >= P]);
qsort([]) ->
  [].
```

- Left of the "||" is an *expression template*
- "Pattern <- List" is a *generator*
  - Elements are picked from the list in order
- The other expressions are *boolean filters*
- If there are multiple generators, you get all combinations of values

Bitstreams and comprehensions

- Bit stream pattern matching:

```
case <<8:4, 42:6>> of
  <<A:7/integer, B/bits>> -> {A,B}
end
```

```
case <<8:4, 42:6>> of
  <<A:3/integer, B:A/bits, C/bits>> -> {A,B,C}
end
```

- Bit stream comprehensions:

```
<< <<X:2>> || <<X:3>> <= Bits, X < 4 >>
```

- Of course, one can also write:

```
[ <<X:2>> || <<X:3>> <= Bits, X < 4 ]
```

Catching exceptions

```
try
  lookup(X)
catch
  not_found ->
    use_default(X);
  exit:Term ->
    handle_exit(Term)
end

%% with 'of' and 'after'
try lookup(X, File) of
  Y when Y > 0 -> f(Y);
  Y -> g(Y)
catch
  ...
after
  close_file(File)
end
```

- Three classes of exceptions
  - throw: user-defined
  - error: runtime errors
  - exit: end process
  - Only catch throw exceptions, normally (implicit if left out)
- Re-thrown if no catch-clause matches
- "after" part is always run (side effects only)

Old-style exception handling

```
Val = (catch lookup(X)),

case Val of
  not_found ->
    %% probably thrown
    use_default(X);
  {'EXIT', Term} ->
    handle_exit(Term);
  _ ->
    Val
end
```

- "catch Expr"
  - Value of "Expr" if no exception
  - Value X of "throw(X)" for a throw-exception
  - "{'EXIT',Term}" for other exceptions
- Hard to tell what happened (not safe)
- Mixes up errors/exits
- In lots of old code

Record syntax

```
-record(foo, {a=0, b}).

{foo, 0, 1} = #foo{b=1}

R = #foo{}
{foo, 0, undefined} = R

{foo, 0, 2} = R#foo{b=2}

{foo, 2, 1} = R#foo{b=1,
                    a=2}

0 = R#foo.a
undefined = R#foo.b

f(#foo{b=undefined}) -> 1;
f(#foo{a=A, b=B})
  when B > 0 -> A + B;
f(#foo{}) -> 0.
```

- Records are just a syntax for working with tagged tuples
- You don't have to remember element order and tuple size
- Good for internal work within a module
- Not so good in public interfaces (users must have same definition!)

Preprocessor

```
-include("defs.hrl").


-ifndef(PI).
-define(PI, 3.1415926).
-endif.


area(R) -> ?PI * (R*R).


-define(foo(X), {foo,X+1}).

{foo,2} = ?foo(1)


%% pre-defined macros
?MODULE
?LINE
```

- C-style token-level preprocessor
  - Runs after tokenizing, but before parsing
- Record definitions often put in header files, to be included
- Use macros mainly for constants
- Use functions instead of macros if you can (compiler can inline)

Type declarations

- Erlang has a notation for declaring types out of the "built-in" ones

```
-type fruit() :: 'apple' | 'banana' | 'orange'.
```

```
-type fruit_list() :: [fruit()].
```

```
-type atom_int_list() :: [atom() | integer()].
```

- These types can then be used to declare the type of record fields

```
-record(my_rec, {a = 0  :: integer(),
                 b      :: fruit(),
                 c = [] :: atom_int_list()}).
```

Spec declarations

- Types can also be used to declare the type of function arguments and return type

```
-spec price(fruit()) -> 8..10.

price(apple) -> 10;
price(banana) -> 9;
price(orange) -> 8.
```

- ... and they can be used to impose constraints that are not necessarily present in the code but reflect programmers' intentions

```
-spec my_app([atom()], [integer()]) -> atom_int_list().

my_app([], Is) -> Is;
my_app([A|As], Is) -> [A | my_app(As, Is)].
```

Dialyzer: A defect detection tool

- Uses static analysis to identify discrepancies in Erlang code bases
  - code points where something is wrong
    - often a bug
    - or in any case something that needs fixing

- Fully automatic
- Extremely easy to use
- Fast and scalable
- Sound for defect detection
  - "Dialyzer is never wrong"

Dialyzer

- Part of the Erlang/OTP distribution since 2007
- Detects
  - Definite type errors
  - API violations
  - Unreachable and dead code
  - Opacity violations
  - Concurrency errors

    Data races (`-Wrace_conditions`)

- Experimental extensions with
  - Stronger type inference: type dependencies
  - Detection of message passing errors & deadlocks

How to use Dialyzer

- First build a PLT (needs to be done once)

```
> dialyzer --build_plt --apps erts kernel stdlib
```

- Once this finishes, analyze your application

```
> cd my_app
> erlc +debug_info -o ebin src/*.erl
> dialyzer ebin
```

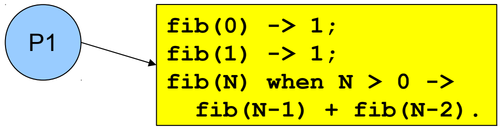
- If there are unknown functions, you may need to add more stuff to the PLT

```
> dialyzer --add_to_plt --apps mnesia inets
```

Erlang: Sequential & Concurrent

Part 2 – Concurrent and Distributed Erlang

Processes

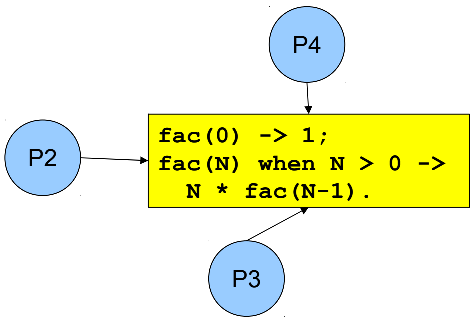


- Whenever an Erlang program is running, the code is executed by a *process*
- The process keeps track of the current program point, the values of variables, the call stack, etc.
- Each process has a unique *Process Identifier (“Pid”),* that can be used to identify the process
- *Processes are concurrent* (they can run in parallel)

Implementation

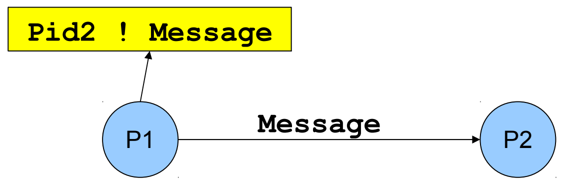
- Erlang processes are implemented by the VM’s runtime system, not by operating system threads
- Multitasking is *preemptive* (the virtual machine does its own process switching and scheduling)
- Processes use very little memory, and switching between processes is very fast
- Erlang can handle large numbers of processes
  - Some applications use more than 100.000 processes
- On a multiprocessor/multicore machine, Erlang processes can be scheduled to run in parallel on separate CPUs/cores

Concurrent process execution



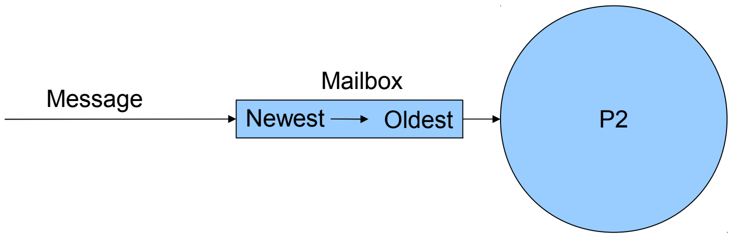
- Different processes may be reading the same program code at the same time
  - They have their own data, program point, and stack – only the text of the program is being shared (well, almost)
  - *The programmer does not have to think about other processes updating the variables*

Message passing



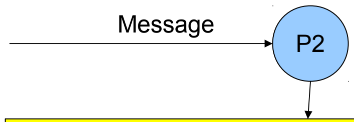
- “!” is the *send operator* (often called “bang!”)
  - The Pid of the receiver is used as the address
- Messages are sent *asynchronously*
  - The sender continues immediately
- Any value can be sent as a message

Message queues



- Each process has a *message queue* (mailbox)
  - Arriving messages are placed in the queue
  - *No size limit* – messages are kept until extracted
- A process *receives* a message when it extracts it from the mailbox
  - Does not have to take the first message in the queue

Receiving a message




```
receive
  Msg -> io:format("~w\n", [Msg])
end
```

- `receive`-expressions are similar to `case` switches
  - Patterns are used to match messages in the mailbox
  - Messages in the queue are tested in order
    - The first message that matches will be extracted
    - A variable-pattern will match the first message in the queue
  - Only one message can be extracted each time

Selective receive

```
receive
  {foo, X, Y} -> ...;
  {bar, X} when ... -> ...;
  ...
end
```

- Patterns and guards let a programmer control the priority with which messages will be handled
  - Any other messages will remain in the mailbox
- The `receive`-clauses are tried in order
  - If no clause matches, the next message is tried
- If *no* message in the mailbox matches, the process *suspends*, waiting for a new message

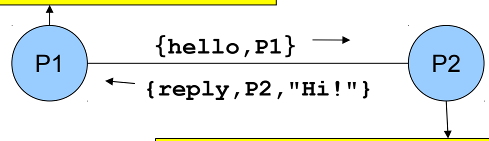
Receive with timeout

```
receive
  {foo, X, Y} -> ...;
  {bar, X} when ... -> ...
after 1000 ->
  ...         % handle timeout
end
```

- A `receive` expression can have an `after`-part
  - The timeout value is either an integer (milliseconds), or the atom `'infinity'` (wait forever)
  - 0 (zero) means “just check the mailbox, then continue”
- The process will wait until a matching message arrives, or the timeout limit is exceeded
- **Soft real-time**: approximate, no strict timing guarantees

Send and reply

```
Pid ! {hello, self()},
receive
   {reply, Pid, String} ->
     io:put_chars(String)
end
```

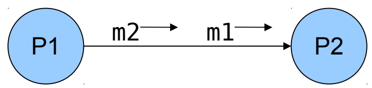



```
receive
   {hello, Sender} ->
     Sender ! {reply, self(), "Hi!"}
end
```

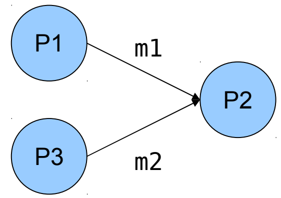
- Pids are often included in messages (**self()**), so the receiver can reply to the sender
  - If the reply includes the **Pid** of the second process, it is easier for the first process to recognize the reply

Message order

FIFO order
(same pair of sender and receiver)

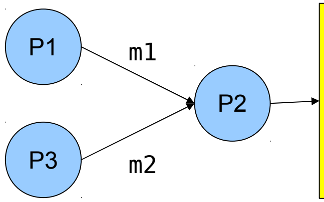


No guaranteed order
(different senders, same receiver)



- Within a node, the only guaranteed message order is when both the sender and receiver are the same for both messages (First-In, First-Out)
  - In the left figure, m1 will always arrive before m2 in the message queue of P2 (if m1 is sent before m2)
  - In the right figure, the arrival order can vary

Selecting unordered messages




```
receive
   m1 -> io:format("Got m1!")
end,
receive
   m2 -> io:format("Got m2!")
end
```

- Using selective receive, we can choose which messages to accept, even if they arrive in a different order
- In this example, P2 will always print "Got m1!" before "Got m2!", even if m2 arrives before m1
  - m2 will be ignored until m1 has been received

Starting processes

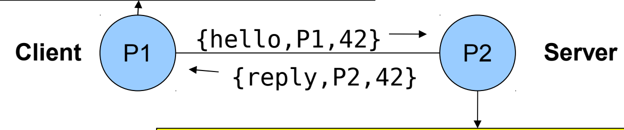
- The **'spawn'** function creates a new process
- There are several versions of '**spawn**':
  - **spawn(fun() -> ... end)**
    - can also do **spawn(fun f/0)** or **spawn(fun m:f/0)**
  - **spawn(Module, Function, [Arg1, ..., ArgN])**
    - **Module:Function/N** must be an exported function
- The new process will run the specified function
- The spawn operation always returns immediately
  - The return value is the Pid of the new process
  - The "parent" always knows the Pid of the "child"
  - The child will not know its parent unless you tell it

Process termination

- A process *terminates* when:
  - It finishes the function call that it started with
  - There is an exception that is not caught
    - The purpose of 'exit' exceptions is to terminate a process
    - "`exit(normal)`" is equivalent to finishing the initial call
- All messages sent to a terminated process will be thrown away, without any warning
  - No difference between throwing away and putting in mailbox just before process terminates
- The same process identifier will not be used again for a long time

A stateless server process

```
run() ->
  Pid = spawn(fun echo/0),

  Pid ! {hello, self(), 42},
  receive
    {reply, Pid, 42} ->
      Pid ! stop
  end.
```




```
echo() ->
  receive
    {hello, Sender, Value} ->
      Sender ! {reply, self(), Value},
      echo();   % loop!
    stop ->
      ok
  end.
```

A server process with state

```
server(State) ->
  receive
    {get, Sender} ->
      Sender ! {reply, self(), State},
      server(State);
    {set, Sender, Value} ->
      Sender ! {reply, self(), ok},
      server(Value);    % loop with new state!
    stop ->
      ok
  end.
```

- The parameter variables of a server loop can be used to remember the current state
- Note: the recursive calls to `server()` are *tail calls (last calls) – the loop does not use stack space*
- *A server like this can run forever*

Hot code swapping

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

start() -> spawn(fun() -> loop(0) end).

loop(State) ->
  receive
    {get, Sender} ->
      ...
      server:loop(State);
    {set, Sender, Value} ->
      ...,
      server:loop(Value);
  ...
```

- When we use "`module:function(...)`", Erlang will always call the latest version of the module
  - If we recompile and reload the `server` module, the process will jump to the new code after handling the next message – we can fix bugs without restarting!

Hiding message details

```
get_request(ServerPid) ->
  ServerPid ! {get, self()}.

set_request(Value, ServerPid) ->
  ServerPid ! {set, self(), Value}.

wait_for_reply(ServerPid) ->
  receive
    {reply, ServerPid, Value} -> Value
  end.

stop_server(ServerPid) ->
  ServerPid ! stop.
```

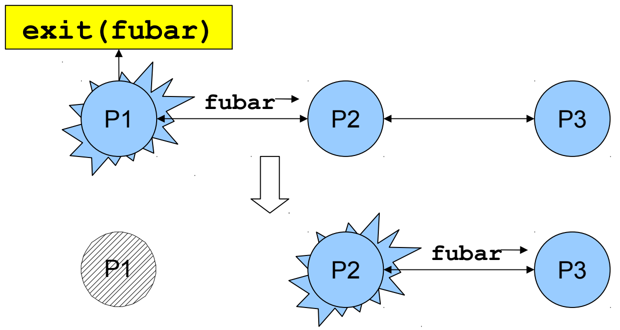
- Using interface functions keeps the clients from knowing about the format of the messages
  - You may need to change the message format later
- It is the client who calls the **`self()`** function here

Registered processes

```
Pid = spawn(...),

register(my_server, Pid),

my_server ! {set, self(), 42},

42 = get_request(my_server),

Pid = whereis(my_server)
```

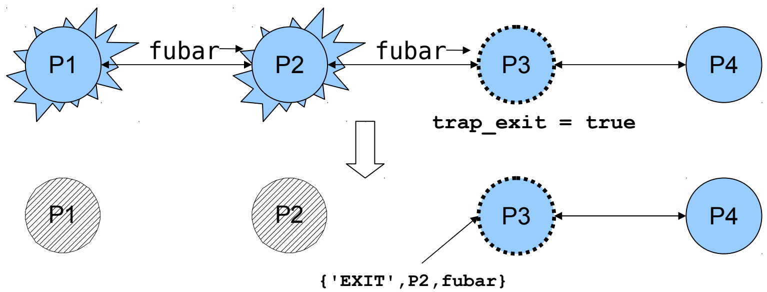
- A process can be registered under a name
  - the name can be any atom
- Any process can send a message to a registered process, or look up the Pid
- The Pid might change (if the process is restarted and re-registered), but the name stays the same

Links and exit signals



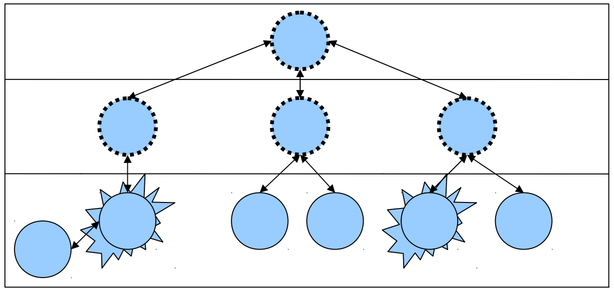
- Any two processes can be *linked*
  - Links are always bidirectional (two-way)
- When a process dies, an *exit signal* is sent to all linked processes, which are also killed
  - Normal exit does not kill other processes

Trapping exit signals



- If a process sets its `trap_exit` flag, all signals will be caught and turned into normal messages
  - **`process_flag(trap_exit, true)`**
  - **`{'EXIT', Pid, ErrorTerm}`**
- This way, a process can watch other processes
  - 2-way links guarantee that sub-processes are dead

Robust systems through layers



- Each layer supervises the next layer and restarts the processes if they crash
- The top layers use well-tested, very reliable libraries (OTP) that practically never crash
- The bottom layers may be complicated and less reliable programs that can crash or hang

Distribution

```
[foo.bar.se] $ erl -name fred
Erlang (BEAM) emulator version 5.5.1

Eshell V5.5.1 (abort with ^G)
(fred@foo.bar.se)1> node().
'fred@foo.bar.se'
(fred@foo.bar.se)2>
```

- Running "**`erl`**" with the flag "**`-name xxx`**"
  - starts the Erlang network distribution system
  - makes the virtual machine emulator a "*node*"
    - the node name is the atom '**`xxx@host.domain`**'
- Erlang nodes can communicate over the network
  - but first they must find each other
  - simple security based on secret cookies

Connecting nodes

```
(fred@foo.bar.se)2> net_adm:ping('barney@foo.bar.se').
pong
(fred@foo.bar.se)3> net_adm:ping('wilma@foo.bar.se').
pang
(fred@foo.bar.se)4>
```

- Nodes are connected the first time they try to communicate – after that, they stay in touch
  - A node can also supervise another node
- The function "**`net_adm:ping(Node)`**" is the easiest way to set up a connection between nodes
  - returns either "**`pong`**" or "**`pang`**" ☺
- We can also send a message to a registered process using "**`{Name,Node} ! Message`**"

Distribution is transparent

- One can send a Pid from one node to another
  - Pids are unique, even over different nodes
- We can send a message to *any* process through its Pid – even if the process is on another node
  - There is no difference (except that it takes more time to send messages over networks)
  - We don't have to know where processes are
  - We can make programs work on multiple computers with no changes at all in the code (no shared data)
- We can run several Erlang nodes (with different names) on the same computer – good for testing

Running remote processes

```
P = spawn('barney@foo.bar.se', Module, Function, ArgList),

global:register_name(my_global_server, P),

global:send(my_global_server, Message)
```

- We can use variants of the **`spawn`** function to start new processes directly on another node
- The module **`'global'`** contains functions for
  - registering and using named processes over the whole network of connected nodes
    - not same namespace as the local "**`register(...)`**"
    - must use "**`global:send(...)`**", not "**`!`**"
  - setting global locks

Ports – talking to the outside

```
PortId = open_port({spawn, "command"}, [binary]),

PortId ! {self(), {command, Data}}

PortId ! {self(), close}
```

- Talks to an external (or linked-in) C program
- A port is connected to the process that opened it
- The port sends data to the process in messages
  - binary object
  - packet (list of bytes)
  - one line at a time (list of bytes/characters)
- A process can send data to the port

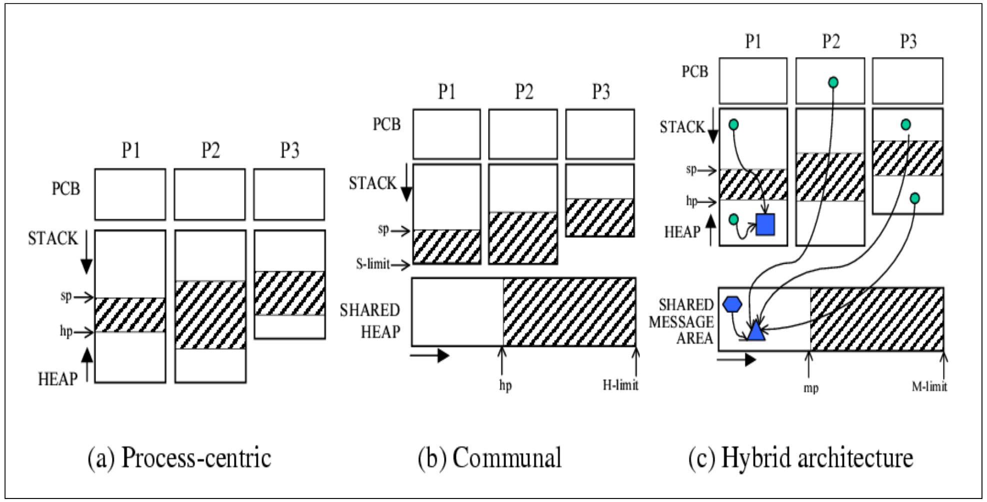
Erlang: Sequential & Concurrent

Part 3 – A Glimpse of Erlang's Implementation

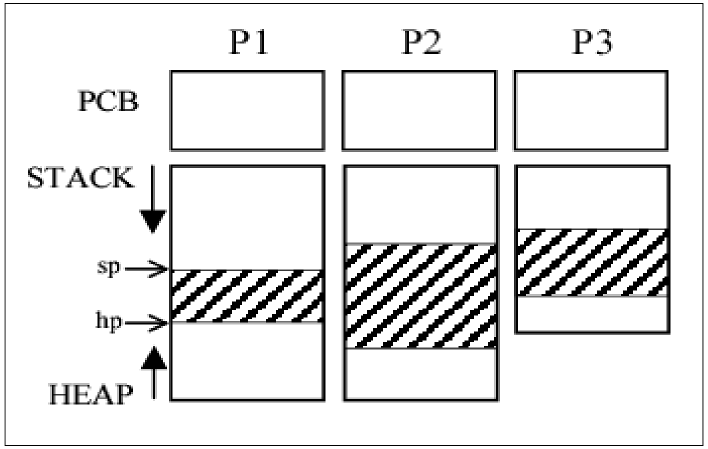
Erlang's Runtime System

- Handles the basic "built-in" things:
  - memory allocation
  - garbage collection
  - process creation
  - message passing
  - context switching
- Several possible ways of structuring
- Some trade-offs have been studied
  - mainly on single core machines!

Runtime System Architectures



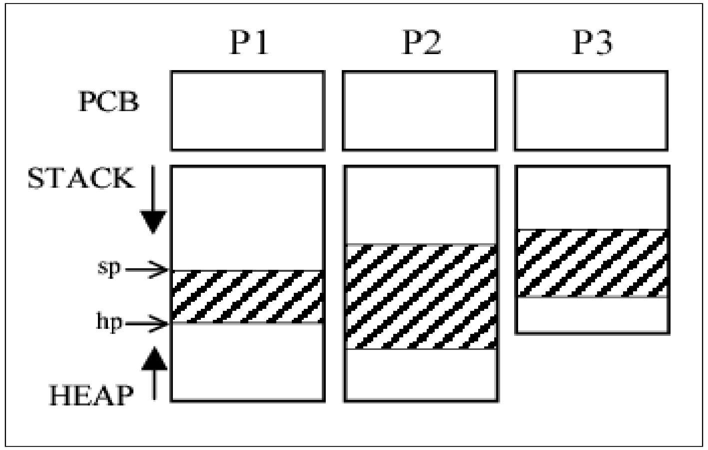
Process local heaps



Process local heaps

- Pros:
  - + Isolation and robustness
  - + Processes can be GC-ed independently
  - + Fast memory deallocation when a process terminates; processes used as regions/arenas
- Cons:
  - - Messages always copied, even between processes on the same machine
    - Sending is O(n) in the size of the message
  - - Memory fragmentation high

The truth...

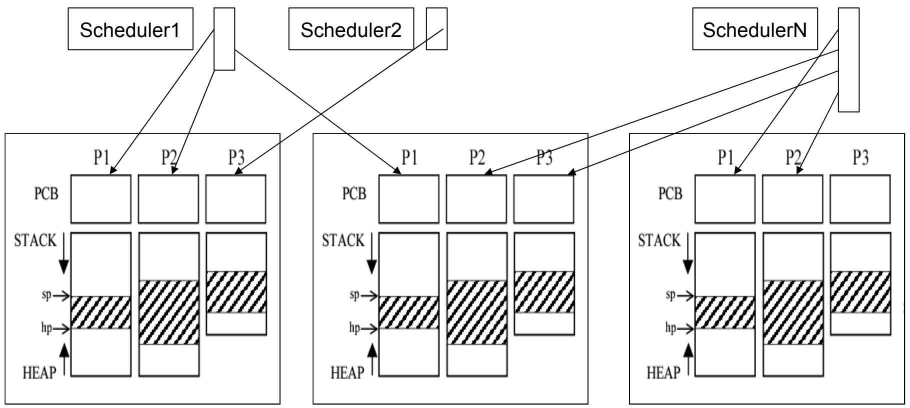


Global areas:
· Atom table
· Process registry

Erlang Term Storage

“Big” Binary Area

SMP Architecture

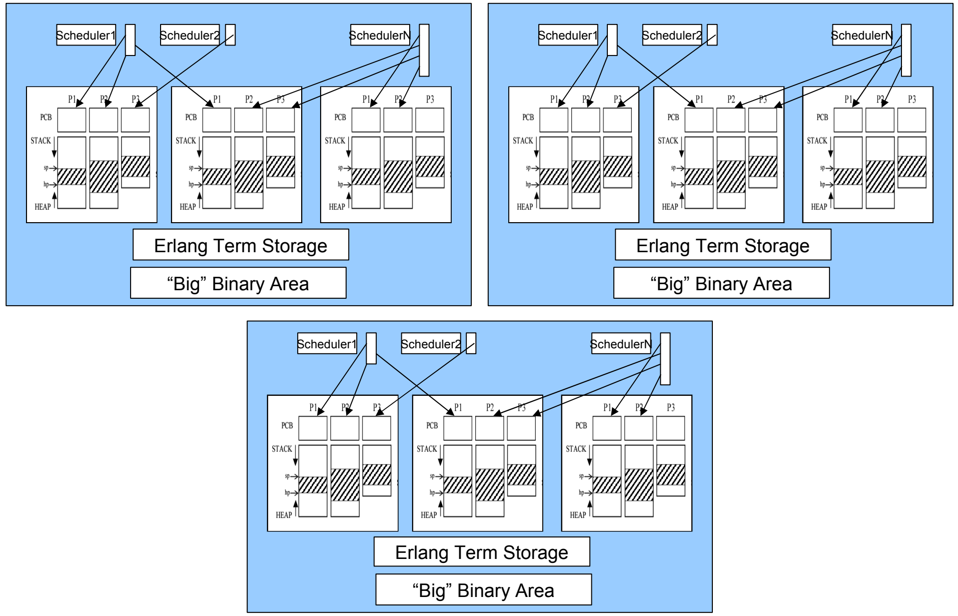


Global areas:
· Atom table
· Process registry

Erlang Term Storage

“Big” Binary Area

Distributed Architecture



More information

Resources:
www.erlang.org

- Getting Started
- Erlang Reference Manual
- Library Documentation

Papers about Erlang and its implementation at:
http://www.it.uu.se/research/group/hipe

Information about Dialyzer at:
http://www.it.uu.se/research/group/hipe/dialyzer/
http://dialyzer.softlab.ntua.gr

More information

Information and tutorials about PropEr at:
http://proper.softlab.ntua.gr

Information about Concuerror at:
https://github.com/mariachris/Concuerror

A paper that shows the use of Concuerror:
Alkis Gotovos, Maria Christakis, and Konstantinos Sagonas.
Test-Driven Development of Concurrent Programs using Concuerror.