IN.5022 — Concurrent and Distributed Computing

Erlang

Prof. P. Felber

pascal.felber@unine.ch

Agenda

- Introduction to Erlang
- Language essentials
- Concurrent programming with Erlang
- Distributed programming with Erlang

ERLANG

The Pragmatic Programmers
Programming Erlang
Software for a Concurrent World
Second Edition
Joe Armstrong
Edited by Susannah Davidson Pfalzer
Learn You Some Erlang for Great Good!
A Beginner's Guide
Fred Hébert
Foreword by Joe Armstrong

Credits:
`https://learnyousomeerlang.com/`
`https://pragprog.com/titles/jaerlang2/`

What is Erlang?

- A functional concurrent programming language
  - Invented in Ericsson (Sweden) in the 80s
  - Concurrent: designed to manage calls in a phone switch
  - Fault-tolerant: can detect and handle SW/HW errors
  - Got its origins in Prolog
- Erlang is a relatively small and simple language
  - Runs on top of a VM
  - Only few basic data types built in the language
- Designed for concurrency and distribution
  - Process part of the language
  - Communication via message passing (local or remote)

Designed for concurrency

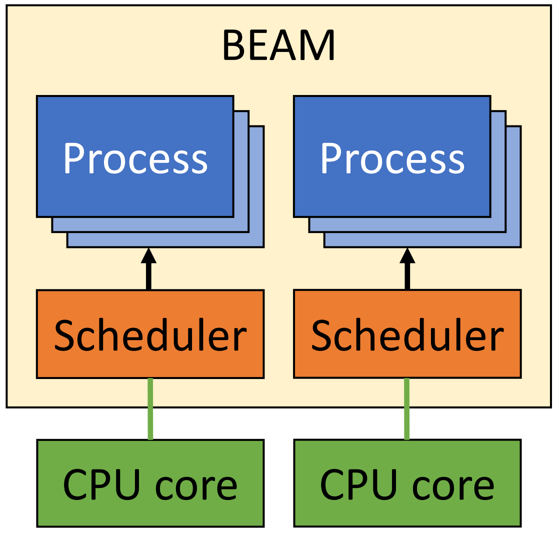
- The process is the main “component”
  - Lightweight and cheap to create/destroy
  - Processes can “link” each other: parent/child, supervisor/worker
  - Processes monitor each other (failures reported as messages)
- Isolation: no corruption between processes
  - No memory sharing: heap is per process
  - No global variables
  - No “pass by reference”
  - No locks: synchronization via message passing only
- “Let it Crash” coding style: process performs its task or fails

Designed for distribution

- Message passing for communication with other processes
  - All communication is asynchronous
  - Communication is handled with callbacks
  - Each process has a “mailbox” (queue)
  - Communication with local/remote processes look the same
- “Actor” model
  - Processes have request/response, internal state
  - All interactions go through mailbox
  - Scaling is easy: processes can be replicated and distribution is transparent

The Erlang virtual machine

- The Erlang virtual machine, BEAM, is a single OS process
  - It uses its own schedulers to distribute execution of Erlang processes (*unit of concurrent execution*) on CPU cores
  - The scheduler is an OS thread responsible for executing multiple Erlang processes
  - BEAM uses multiple schedulers to parallelize the work over available CPU cores



Erlang shell

- Start with
  `erl`
- Stop with
  `^C ^C`
- Shutdown cleanly with
  `q().`
- Expressions end with dot

```
% erl
Erlang/OTP 23 [erts-11.0.3] …
Eshell V11.0.3  (abort with ^G)
1> ^C
BREAK: (a)bort (A)bort with dump …
^C
%
% erl
Erlang/OTP 23 [erts-11.0.3] …
Eshell V11.0.3  (abort with ^G)
1> q().
ok
%
```

Some useful shell commands

- History: `h()`
  - Print last 20 commands
    `erl`
- Bindings: `b()`
  - See all variable bindings
- Forget: `f()`, `f(Var)`
  - Forget all or some variable
- Evaluate: `e(n)`
  - Repeat command **n** from history (**-1** for last)

```
1> A=42.
42
2> h().
1: A = 42
-> 42
ok
3> b().
A = 42
ok
4> f(A).
ok
5> b().
ok
6> e(1).
42
7> b().
A = 42
ok
```

Numbers

- Erlang supports integer and floating-point values alike
  - Classical arithmetic operators
  - Apply to both except integer division/remainder
  - Large integers, 64-bit floats
- Precedence rules apply (overridden by parentheses)
- Integers expressed in any base 2..36 as `Base#Value`
  - Powerful tools for binaries

```
1> 2 + 15.
2> 49 * 100.
3> 1892 - 1472.
4> 5 / 2.
5> 5 div 2.
6> 5 rem 2.

7> (50 * 100) - 4999.
8> -(50 * 100 - 4999).
9> -50 * (100 - 4999).

10> 2#101010.
11> 8#0677.
12> 16#AE.
13> <<16#F09A29:24>>.

14> 1111…1 + 1.
15> 1111…1 * 1111…1.
```

```
17
4900
420
2.5
2
1

1
-1
244950

42
447
174
<<240,154,41>>

1111…2
123…321
```

Variables

- Start with uppercase letter
  - "Camel case" by convention
  - No "funny" characters
- Store values, functions, data structures, process IDs
- "Invariable" variables
  - Can be assigned only once
  - Bind right-side to left-side
  - Variable _ ("don't care") cannot be bound

```
1> One.                          % error
2> One = 1.
3> Un = Uno = One = 1.
4> Two = One + One.
5> Two = 2.
6> Two = Two + 1.                % error
7> two = 2.                      % error

8> 47 = 45 + 2.
9> 47 = 45 + 3.                  % error

10> _ = 14+3.
11> _.                           % error
```

Atoms

- Atoms are literals
  - Constants with their own name for value
  - Starts by a lowercase letter or enclosed between quotes (`'`)
  - Some atoms are reserved names of the language and cannot be used
    - `and`, `if`, `else`, …

```
1> atom.
2> atoms_rule.
3> atoms_rule@erlang.
4> 'Atoms can be cheated!'.
5> atom = 'atom'.
```

Boolean algebra and comparison operators

- Usual Boolean algebra (like in other languages)
  - Lazy **and**, **or** (short-circuit variants **andalso**, **orelse**)
  - The terms **true** and **false** are atoms
- Usual comparison operators (but unusual syntax)
  - Equality: strict (**=:=** and **=/=**) or not (**==** and **/=**)
  - Less than or equal to: **=<**

```
1> true and false.
2> false or true.
3> true xor false.
4> not false.
5> not (true and true).
6> 5 =:= 5.
7> 1 =:= 0.
8> 1 =/= 0.
9> 5 =:= 5.0.                % strict
10> 5 == 5.0.            % not strict
11> 5 /= 5.0.
12> 1 < 2.
13> 1 < 1.
14> 1 >= 1.
15> 1 =< 1.
```

Tuples

- Tuples group elements

  `{El1,El2,…,ElN}`

  - Number of elements known (constant length)
  - Assign values to elements by binding the tuple
  - Retrieve elements by binding them to variables
  - Also useful with single values (e.g., use atom as “tag”)
  - Tuples can contain any type, even another tuple

```
1> X = 10, Y = 4.
2> Point = {X,Y}.      % point is 2 terms

 > f().                % clear variables
3> Point = {4,5}.         % assign tuple
4> {X,Y} = Point.        % read elements
5> X.
6> {X,_} = Point.          % read only X

7> {_,_} = {4,5}.                   % ok
8> {_,_} = {4,5,6}.              % error

9> Temperature = {celsius,23.213}.
10> {kelvin,T} = Temperature.   % error
11> {celsius,T} = Temperature.     % ok

12> {point,{X,Y}}.     % “tagged” tuple
```

Lists

- Lists are sequences of terms

  `[El1,El2,…,ElN]`

  - Can contain anything
  - Implemented as linked lists (unbounded)
  - Operators `++` and `--` to glue and remove elements (right-associative)
- Strings are lists!
  - Printed as string if they hold only printable characters

```
1> [1,2,3,{numbers,[4,5,6]},5.34,atom].

2> [97,98,99].           % also a string

3> [97,98,99,4,5,6].
4> [125].

5> [1,2,3] ++ [4,5].
6> [1,2,3,4,5] -- [1,2,3].
7> [2,4,2] -- [2].
8> [2,4,2] -- [2,4,2].

9> [1,2,3] -- [1,2] -- [3].
10> [1,2,3] -- [1,2] -- [2].
```

Lists

- Lists are recursive
  - First element is head (**hd**)
  - Rest is tail (**tl**): a list
  - Simplified syntax to separate head from tail with “cons” (constructor) operator **|**
  - Head and tail accessible with pattern matching (as tuples)
  - Lists can be built recursively with cons operator (last element must be empty list)

```
11> hd([1,2,3,4]).
12> tl([1,2,3,4]).

13> List = [2,3,4].
14> NewList = [1|List].

15> [Head|Tail] = NewList.
16> Head.
17> Tail.
18> [NewHead|NewTail] = Tail.
19> NewHead.

20> [1|[]].
21> [2|[1|[]]].
22> [3|[2|[1|[]]]].
```

List comprehensions

- Ways to build or modify lists

  ```
  NewList = [Expr ||
  Pattern <- List, Cond1,
  Cond2, …, CondN].
  ```

  - Build sets from sets (list “set notation” in mathematics)
  - Can add constraints using Boolean operations
  - Can have multiple “generator expression”

  ```
  NewList = [Expr ||
  GenExpr1, … GenExprN,
  Cond1, …, CondM].
  ```

```
1> [2*N || N <- [1,2,3,4]].
2> [X || X <- [1,2,3,4,5,6,7,8,9,10],
   X rem 2 =:= 0].
                 % price in $3-10 incl. 7% tax
3> RestaurantMenu = [{steak,5.99},
   {beer,3.99}, {poutine,3.50},
   {kitten,20.99}, {water,0.00}].
4> [{Item,Price*1.07} ||
   {Item,Price} <- RestaurantMenu,
   Price >= 3, Price =< 10].

5> [X+Y || X <- [1,2], Y <- [2,3]].

6> Weather = [{toronto,rain},
   {montreal,storms},{london,fog},
   {paris,sun},{boston,fog},
   {vancouver,snow}].
7> FoggyPlaces = [X ||
   {X,fog} <- Weather].
```

Modules

- A module is a “program” (attributes and code)
  - Attributes start with `-`
    - Module name
    - Imported functions
    - Exported functions
    - Macros…
  - Functions are Erlang code
- Module can be compiled
  - From OS shell (`erlc`)
  - From Erlang shell (`c()`)

```
-module(useless).              % module name
-export([add/2,        % exported functions
  hello/0]).  % function w/out arguments

add(A, B) ->                     % add 2 numbers
  A + B.

hello() ->                      % print greeting
  io:format("Hello, world!~n").
```

```
1> c(useless).         % file: useless.erl
{ok,useless}
2> useless:add(7, 2).
9
3> useless:hello().
Hello, world!
ok
4> useless:module_info().      % try that…
```

Functions

- Evaluate conditions based on pattern matching
  - A function is declared as a sequence of “function clauses” (“head -> body”) separated by **;**

    ```
    func(X) -> Expr;
    func(Y) -> Expr₁, Expr₂;
    func(_) -> Expr.
    ```
  - The name is an atom
  - The parameters are patterns
  - The body can contain several expressions separated by **,**

```
def greet(gender, name):           # Python
  if gender == "male":
    print(f"Hello, Mr. {name}!")
  elif gender == "female":
    print(f"Hello, Mrs. {name}!")
  else:
    print(f"Hello, {name}!")
```

```
greet(male, Name) ->                    % Erlang
  io:format("Hello, Mr. ~s!", [Name]);
greet(female, Name) ->
  io:format("Hello, Mrs. ~s!", [Name]);
greet(_, Name) ->
  io:format("Hello, ~s!", [Name]).
```

Pattern matching

- Pattern matching is powerful (and can be more complex)
  - Match elements of list
  - Match multiple parameters
  - …

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

head([H|_]) -> H.
second([_,X|_]) -> X.

same(X, X) -> true;
same(_, _) -> false.
```

```
1> functions:head([1,2,3,4]).
1
2> functions:second([1,2,3,4]).
2
3> functions:same(3, 3).
true
4> functions:same(3, A).
* 1: variable 'A' is unbound
5> A=3.
3
6> functions:same(3, A).
true
7> functions:same(B, B).
* 1: variable 'B' is unbound
8> functions:same(hello, hello).
true
9> functions:same([1,2,3], [1,3,2]).
false
```

Guards

- Guards are additional clauses in a function's head
  - Condition must be verified for clause to be evaluated

    ```
    func(X) when Guard ->
      Expr;
    ```
  - Makes pattern matching more expressive
  - Warning: guards separated by **,** meaning "and else" or by **;** meaning "or else"

```
old_enough(0) -> false;
old_enough(1) -> false;
% …
old_enough(6) -> false;
old_enough(_) -> true.

old_enough(X) when X >= 7 -> true;
old_enough(_) -> false.

right_age(X) when X >= 7, X =< 77 ->
  true;                           % "and else"
right_age(_) -> false.

wrong_age(X) when X < 7; X > 77 ->
  true;                            % "or else"
wrong_age(_) -> false.
```

Function evaluation

- Upon function call: **m:F/N**
  - Code for **F** is located
  - Clauses are sequentially scanned until finding one such that…
    - the patterns in the head match arguments, and…
    - the guards (if any) are true
  - If clause is found, expressions in the body are evaluated sequentially
  - If clause is not found (or code is missing), runtime error

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

fac(N) when N > 0 ->    % 1st clause head
  N*fac(N-1);           % 1st clause body
fac(0) ->               % 2nd clause head
  1.                    % 2nd clause body
```

```
1> m:fac(0).                      % what happens?
```

```
2> m:fac(1).                      % what happens?
```

If and case

- Erlang also provides `if` and `case` statements
  - They behave similarly to guards but are not restricted to the function head
  - “If” is like a guard
  - “Case… of” is like a whole function head

```
help_me(Animal) ->
  Talk = if
    Animal == cat  -> "meow";
    Animal == beef -> "mooo";
    Animal == dog  -> "bark";
    true -> "<...>"
  end,
  {Animal, "says " ++ Talk ++ "!"}.

beach(Temperature) ->
  case Temperature of
    {celsius,N} when N >= 20,
      N =< 45 -> 'good';
    {fahrenheit,N} when N >= 68,
      N =< 113 -> 'good in the US';
    _ -> 'avoid beach'
  end.
```

Type conversions and type guards

- Erlang provides built-in functions (BIFs) for type conversions and guards
  - In module **erlang**

```
list_to_integer(X)
integer_to_list(X)
list_to_float(X)
…
is_integer(X)
is_float(X)
is_atom(X)
…
```

```
1> list_to_integer("54").
2> integer_to_list(54).
3> list_to_integer("54.32").      % error
4> list_to_float("54.32").
5> atom_to_list(true).
6> list_to_bitstring("hi there").
7> bitstring_to_list(<<"hi there">>).
8> is_integer(54).
9> is_float("3.5").
10> is_atom(true).
```

Loops and recursion

- Erlang relies on “recursion” for loops
  - Base case and function that calls itself
- Some classical examples
  - Factorial computation
  - Length of a list
  - Reversing a list
- For better efficiency, one can use “tail recursion”
  - Space usage is constant

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

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

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

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

tail_fac(0, Acc) -> Acc;
tail_fac(N, Acc) when N > 0 ->
  tail_fac(N-1, N*Acc).
```

Higher order functions

- Erlang provides support for lambda calculus
  - Pass functions as parameter, anonymous functions, …

```
1> L = [1,2,3,4,5].
2> hhfuns:increment(L).
3> hhfuns:decrement(L).
4> hhfuns:map(fun hhfuns:incr/1, L).
5> hhfuns:map(fun hhfuns:decr/1, L).
6> Fn = fun() -> a end.
7> Fn().
8> hhfuns:map(fun(X) -> X + 1 end, L).
9> hhfuns:map(fun(X) -> X - 1 end, L).
```

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

increment([]) -> [];
increment([H|T]) -> [H+1|increment(T)].

decrement([]) -> [];
decrement([H|T]) -> [H-1|decrement(T)].

map(_, []) -> [];
map(F, [H|T]) -> [F(H)|map(F,T)].

incr(X) -> X + 1.
decr(X) -> X - 1.
```

Simple input and output

- Function **io:read** reads any term from terminal
- Function **io:format** prints formatted output
  - The **~** character denotes a token (some built in, some to format data in arguments)

    ~n: line break

    ~s: string

    ~f: float

    ~w: any term (standard syntax)

    ~p: any term ("pretty-printed")

    …

```
1> io:read("Enter term> ").
Enter term> atom.
{ok,atom}
2> io:read("Enter term> ").
Enter term> {2,tue,{mon,"weds"}}.
{ok,{2,tue,{mon,"weds"}}}
```

```
1> io:format("hello, world!~n").
hello, world!
ok
2> List = [2,3,4].
[2,3,4]
3> io:format("sum(~w)=~w~n",
    [List,lists:sum(List)]).
sum([2,3,4])=9
ok
```

Built-in functions

- BIFs are implemented in C in the runtime and auto-imported

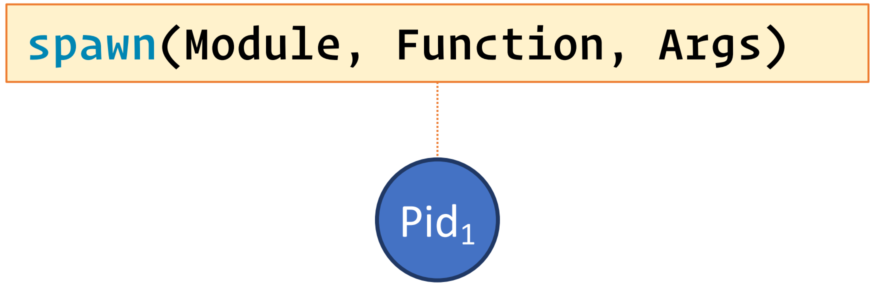
```
abs/1 adler32/1 adler32/2 adler32_combine/3 append_element/2 apply/2 apply/3 atom_to_binary/1 atom_to_binary/2 atom_to_list/1
binary_part/2 binary_part/3 binary_to_atom/1 binary_to_atom/2 binary_to_existing_atom/1 binary_to_existing_atom/2 binary_to_float/1
binary_to_integer/1 binary_to_integer/2 binary_to_list/1 binary_to_list/3 binary_to_term/1 binary_to_term/2 bit_size/1
bitstring_to_list/1 bump_reductions/1 byte_size/1 cancel_timer/1 cancel_timer/2 ceil/1 check_old_code/1 check_process_code/2
check_process_code/3 convert_time_unit/3 crc32/1 crc32/2 crc32_combine/3 date/0 decode_packet/3 delete_element/2 delete_module/1
demonitor/1 demonitor/2 disconnect_node/1 display/1 dist_ctrl_get_data/1 dist_ctrl_get_data_notification/1 dist_ctrl_get_opt/2
dist_ctrl_input_handler/2 dist_ctrl_put_data/2 dist_ctrl_set_opt/3 element/2 erase/0 erase/1 error/1 error/2 exit/1 exit/2
external_size/1 external_size/2 float/1 float_to_binary/1 float_to_binary/2 float_to_list/1 float_to_list/2 floor/1 fun_info/1
fun_info/2 fun_to_list/1 function_exported/3 garbage_collect/0 garbage_collect/1 garbage_collect/2 get/0 get/1 get_cookie/0 get_keys/0
get_keys/1 get_stacktrace/0 group_leader/0 group_leader/2 halt/0 halt/1 halt/2 hd/1 hibernate/3 insert_element/3 integer_to_binary/1
integer_to_binary/2 integer_to_list/1 integer_to_list/2 iolist_size/1 iolist_to_binary/1 iolist_to_iovec/1 is_alive/0 is_atom/1
is_binary/1 is_bitstring/1 is_boolean/1 is_builtin/3 is_float/1 is_function/1 is_function/2 is_integer/1 is_list/1 is_map/1
is_map_key/2 is_number/1 is_pid/1 is_port/1 is_process_alive/1 is_record/2 is_record/3 is_reference/1 is_tuple/1 length/1 link/1
list_to_atom/1 list_to_binary/1 list_to_bitstring/1 list_to_existing_atom/1 list_to_float/1 list_to_integer/1 list_to_integer/2
list_to_pid/1 list_to_port/1 list_to_ref/1 list_to_tuple/1 load_module/2 load_nif/2 loaded/0 localtime/0 localtime_to_universaltime/1
localtime_to_universaltime/2 make_ref/0 make_tuple/2 make_tuple/3 map_get/2 map_size/1 match_spec_test/3 max/2 md5/1 md5_final/1
md5_init/0 md5_update/2 memory/0 memory/1 min/2 module_loaded/1 monitor/2 monitor_node/2 monitor_node/3 monotonic_time/0
monotonic_time/1 nif_error/1 nif_error/2 node/0 node/1 nodes/0 nodes/1 now/0 open_port/2 phash/2 phash2/1 phash2/2 pid_to_list/1
port_call/3 port_close/1 port_command/2 port_command/3 port_connect/2 port_control/3 port_info/1 port_info/2 port_to_list/1 ports/0
pre_loaded/0 process_display/2 process_flag/2 process_flag/3 process_info/1 process_info/2 processes/0 purge_module/1 put/2 raise/3
read_timer/1 read_timer/2 ref_to_list/1 register/2 registered/0 resume_process/1 round/1 self/0 send/2 send/3 send_after/3 send_after/4
send_nosuspend/2 send_nosuspend/3 set_cookie/2 setelement/3 size/1 spawn/1 spawn/2 spawn/3 spawn/4 spawn_link/1 spawn_link/2
spawn_link/3 spawn_link/4 spawn_monitor/1 spawn_monitor/2 spawn_monitor/3 spawn_monitor/4 spawn_opt/2 spawn_opt/3 spawn_opt/4
spawn_opt/5 spawn_request/1 spawn_request/2 spawn_request/3 spawn_request/4 spawn_request/5 spawn_request_abandon/1 split_binary/2
start_timer/3 start_timer/4 statistics/1 suspend_process/1 suspend_process/2 system_flag/2 system_info/1 system_monitor/0
system_monitor/1 system_monitor/2 system_profile/0 system_profile/2 system_time/0 system_time/1 term_to_binary/1 term_to_binary/2
term_to_iovec/1 term_to_iovec/2 throw/1 time/0 time_offset/0 time_offset/1 timestamp/0 tl/1 trace/3 trace_delivered/1 trace_info/2
trace_pattern/2 trace_pattern/3 trunc/1 tuple_size/1 tuple_to_list/1 unique_integer/0 unique_integer/1 universaltime/0
universaltime_to_localtime/1 unlink/1 unregister/1 whereis/1 yield/0
```

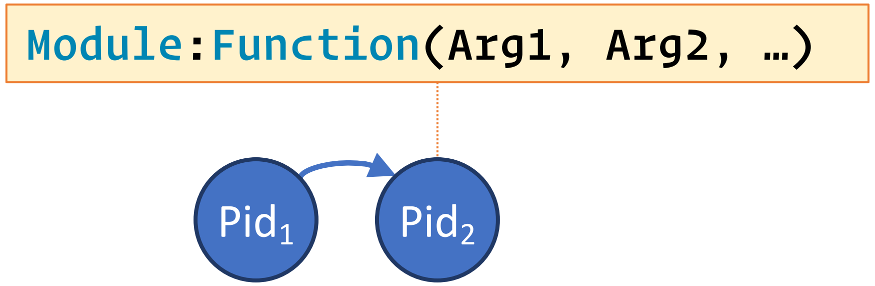
Creating processes

- Erlang is designed for massive concurrency
  - Lightweight processes (grow/shrink dynamically), small memory footprint, fast to create and terminate, can easily exchange messages
- Processes are created using

```
Pid = spawn(m, f, [a])
```

  - BIF returns process identifier



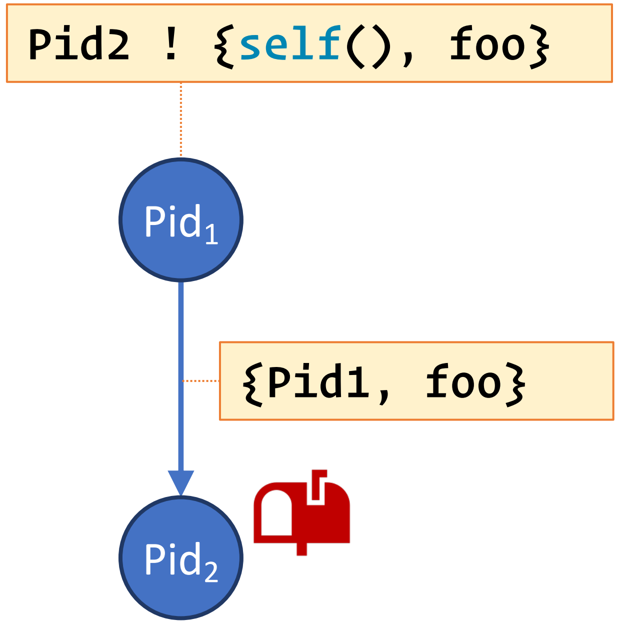


Sending messages

- Messages are sent using

  `Pid ! Message`

  - "Pid": valid process identifier
  - "Message": value of any data type
- Each process has a mailbox for incoming messages (stored in FIFO order)
  - Upon send, a message is copied from sending process into the recipient's mailbox
  - Message are delivered in sending order
  - Sending a message *never fails*
  - Message passing is *asynchronous*



```
1> Pid = self().
<0.80.0>
2> Pid ! hello.
hello
3> flush().
Shell got hello
```

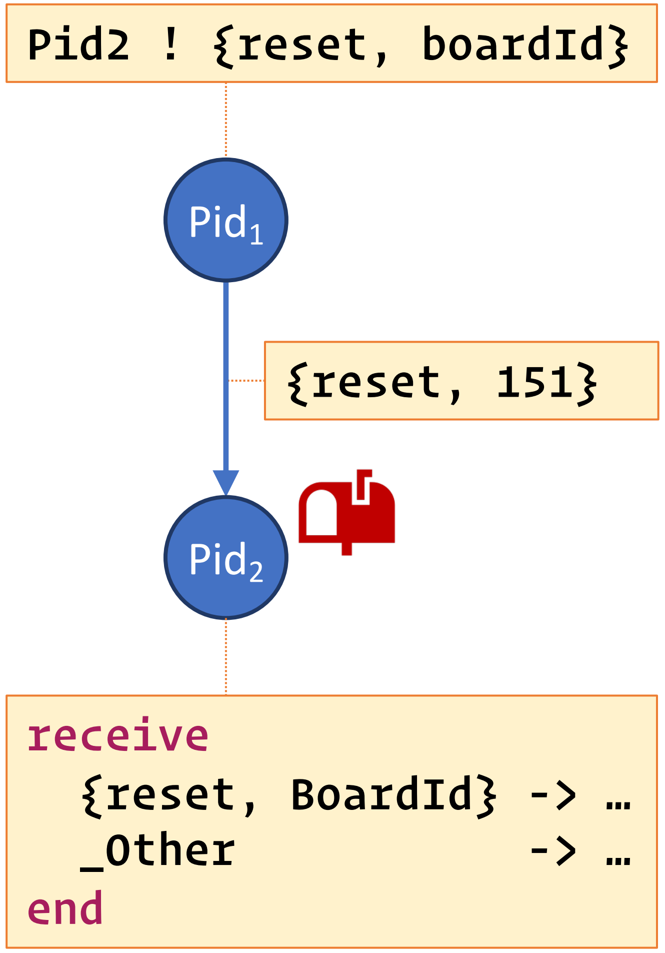
Spawning and sending: more examples

```
1> F = fun() -> 2 + 2 end.
#Fun<erl_eval.45.97283095>
2> spawn(F).
<0.83.0>
3> spawn(fun() -> io:format("~p~n",[2 + 2]) end).
4
<0.85.0>
4> G = fun(X) -> timer:sleep(10), io:format("~p~n", [X]) end.
#Fun<erl_eval.44.97283095>
5> [spawn(fun() -> G(X) end) || X <- lists:seq(1,5)].
[<0.88.0>,<0.89.0>,<0.90.0>,<0.91.0>,<0.92.0>]
2
1
5
3
4
```

```
6> self() ! hello.
hello
7> self() ! self() ! 'hi'.
'hi'
8> flush().
Shell got hello
Shell got 'hi'
Shell got 'hi'
ok
```

Receiving messages

- Upon **receive**, the oldest message in the mailbox is pattern-matched against each pattern from the receive in turn
  - If a clause matches, the message is retrieved from the mailbox, variables in the pattern are bound to the matching parts of the message, body of the clause is executed
  - If none of the clauses matches, the other messages are pattern-matched one by one against all the clauses until one match is found (or none is found)



```
receive
  {reset, BoardId} -> …
  _Other           -> …
end
```

Selective vs. non-selective receive

- Selective receive retrieves only the messages matching certain criteria, leaving the others in the mailbox
  - One can impose an order on messages received
  - Receives can have guards (like "case… of")
- Non-selective receive retrieves messages regardless of content

```
receive ->    % selective (w/ binding)
  {celsius, T} -> convert(T);
end

receive ->      % selective (no order)
  {celsius, T} when T >= 20 -> true;
  foo -> true;
end

receive ->      % selective (in order)
  foo -> true
end,
receive ->
  bar -> true
end

receive ->             % non-selective
  Msg -> true
end
```

Registering processes

- One can *register* processes with an *alias*
  - Can be searched and listed

```
1> c(echo).
{ok,echo}
2> echo:go().
hello
ok
3> whereis(echo).
<0.87.0>
4> regs().
** Registered procs on node … **
Name …Pid      …Initial Call …Reds …Msgs
…
echo  <0.87.0> echo:loop/0      11      0
```

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

go() ->
  register(echo, spawn(echo, loop, [])),
  echo ! {self(), hello},
  receive
    {_Pid, Msg} ->
      io:format("~w~n", [Msg])
  end.

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

Distributed processes

- A node is an executing Erlang runtime that has been given a name using a command-line flag
- Format is an atom "name@host"
  - The name is given by the user
  - The host is either the fully qualified server name or just its local name

    `-name` (long names, e.g., `foo@host.doma.in`)

    `-sname` (short names, e.g., `foo` or `foo@host`)
- The `node()` function returns the name of the node

```
% erl -name pf@uni.ne.ch
❯ erl
Erlang/OTP 23 …
(pf@uni.ne.ch)1> node().
'pf@uni.ne.ch'
(pf@uni.ne.ch)2>
```

```
% erl -sname pf
❯ erl
Erlang/OTP 23 …
(pf@uni)1> node().
'pf@uni'
(pf@uni)2>
```

Security

- Access controlled is managed via authentication
  - Specifies which nodes can communicate with other nodes
  - Each node has its own magic cookie, which is an Erlang atom
  - Nodes that communicate together must have an identical cookie
- By default, the magic cookie is in a file readable only by user

  `.erlang.cookie` (or `$HOME/.erlang.cookie`)
  - If missing, the cookie is randomly generated when the file is first created, then it is reused for subsequent nodes
- Nodes can also be started with an explicit cookie value

```
erl -sname foo -setcookie bar
```

Remote communication

- Remote messages are (almost) like local ones
  - Must add remote node name
  - Code must be available there

```
-module(dist).      % distributed module
-export([s/0]).
s() ->
  register(srv, self()), loop().
loop() ->
  receive {M, Pid} -> Pid ! M end,
  loop().
```

```
% erl -sname …
> erl
Erlang/OTP 23 …
…1> spawn('pf@uni', dist, s, []).    % 2
<8768.90.0>
…2> {srv, 'pf@uni'} ! {hi, self()}. % 4
{hi,<0.86.0>}
…3> flush().                         % 5
Shell got hi
ok
```

```
% erl -sname pf
> erl
Erlang/OTP 23 …
(pf@uni)1> node().                        % 1
'pf@uni'
(pf@uni)2> regs().                        % 3
** Registered procs on node … **
Name …Pid      …Initial Call …Reds …Msgs
…
srv  <0.87.0> dist:s/0          11     0
```

Example: a simple file server

```
-module(afile_client).
-export([ls/1, get_file/2]).

ls(Server) ->
  Server ! {self(), list_dir},
  receive
    {Server, FileList} ->
      FileList
  end.

get_file(Server, File) ->
  Server ! {self(), {get_file, File}},
  receive
    {Server, Content} ->
      Content
  end.
```

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

start(Dir) ->
  spawn(afile_server, loop, [Dir]).

loop(Dir) ->
  receive
    {Client, list_dir} ->
      Client ! {self(),
                file:list_dir(Dir)};
    {Client, {get_file, File}} ->
      Full = filename:join(Dir, File),
      Client ! {self(),
                file:read_file(Full)}
  end, loop(Dir).
```

Example: a simple file server (cont'd)

- Compare with C or Java (10s vs. 100s SLOC)
  - Trivial to distribute, simple error handling, "symmetric" code…

```
1> c(afile_server).
{ok,afile_server}
2> c(afile_client).
{ok,afile_client}
3> FS = afile_server:start(".").
<0.92.0>
4> afile_client:ls(FS).
{ok,["afile_client.erl","afile_client.beam",
     "afile_server.erl","afile_server.beam"]}
5> afile_client:get_file(FS, "missing").
{error,enoent}
6> afile_client:get_file(FS, "afile_server.erl").
{ok,<<"%% ---\n%%  Excerpted from \"Programming Erlang, Second
Edition\",\n%%  published by The Pragmatic Bookshelf.\n%%"...>>}
```

Some useful tricks and programming patterns

- There is no way to track “state” in Erlang by changing values of global (or process-local) variables
    - The preferred way of doing it is by recursively calling a function and changing its parameters
- List comprehension is the Erlang way of doing “for loops”
    - For creating elements, sending/receiving multiple messages, etc.
    - Complementary to recursion
- Code reuse can often be elegantly done by sending a message to oneself
    - Upon reception, trigger execution of the target code block

Conclusion

- Erlang is well adapted to concurrent and distributed programming
  - Lightweight processes
  - Massive concurrency
  - Powerful pattern matching
  - Asynchronous message passing
  - Similar mechanisms for local and remote communication
  - Built-in fault tolerance

Appendix

Why functional programming?

Why functional programming?

"Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. In other words, functional programming promotes code with no side effects, no change of value in variables. It opposes imperative programming, which emphasizes change of state."

[Wikipedia]

- This means…
  - No mutable data (no side effect)
  - No state (no implicit, hidden state)

Once assigned (value binding), a variable (a symbol)
does not change its value

Is state really that bad?

- No… only "hidden" implicit state is bad
- Functional programming does not eliminate state, it just makes it visible and explicit (when programmers want it)
  - Functions are pure functions in the mathematical sense: their output depend only on their inputs, there is no "environment"
  - Functions return the same result when called with the same inputs
- Advantages?
  - Cleaner code: "variables" are not modified once defined, no need to track their changes to understand code
  - Referential transparency: expressions can be replaced by their values (thanks to functions being idempotent)

Advantages of referential transparency

- Memoization: cache results from previous function calls
- Idempotence: same results for multiple identical calls
- Modularization: no state pervades the whole code, promotes bottom-up programming with black-box components
- Ease of debugging: functions are isolated and only depend on their input and output, hence very easy to debug
- Parallelization: function calls are independent and can be parallelized in different process/CPUs/computers…
- Simpler concurrency: no semaphores, no monitors, no locks $\Rightarrow$ no race conditions, no deadlocks…