Slide 1

DISTRIBUTED SYSTEMS (COMP9243)

Lecture 2: Erlang

① Introduction
② Basics: Sequential programming
③ Concurrent programming
④ More Details & Resources

Slide 2

INTRODUCTION TO ERLANG

Erlang: Functional language with built in concurrency support

OTP: A large collection of libraries for Erlang

Features:

- ➜ Concurrency and asynchronous message passing
- ➜ Lightweight processes. Fast context switches
- ➜ Virtual machine
- ✗ Not suitable for low-level system software

History:

- ➜ Named after mathematician Agner Erlang
- ➜ Originated from Ericsson (maybe Erlang actually stands for ERicsson LANGuage?)
- ➜ Used for a lot of telecoms applications: e.g. switches
- ➜ Open sourced in 1998

Slide 3

THE ERLANG ENVIRONMENT

```
unix% erl
1> 1 + 2.
3
2> c(demo).
{ok,demo}
3> demo:double(25).
50
4> date().
{2004,2,24}
5> halt().
unix% cat demo.erl
-module(demo).
-export([double/1]).
double(X) -> 2 * X.
unix%
```

Slide 4

BASICS: SEQUENTIAL PROGRAMMING

- ➜ Numbers: Integers (`1`, `-10`), Floats (`3.1415`, `-0.23`)
  - Hex: `16#AB123` Binary: `2#100110`
  - ASCII: `$A` (65), `$z` (122), etc.
- ➜ Atoms: `hello`, `how_are_you`, `'I am fine'`
- ➜ Variable: `Counter`, `Good_server`, `BadServer`
  - Only bound once. Value cannot be changed once bound!!!
- ➜ Operators: `+`, `-`, `*`, `/`, `>`, `>=`, `<`, `=<`, `==`, `=/=`

Slide 5

Data Structures:

- Tuples: `{123, hello, 'Good Morning', {super, 456}},{}`
- Lists: `[123, hello, 'Welcome'],[],"abcdefg",""`
- Combinations: `[{123, house}, guest, {friends, family}], {123, [1,2,3,4], "building"}`
- Others (dict, process dictionary, etc.): see documentation

Slide 6

Pattern Matching:

Binding variables to values

- ☑ `A = 10`
- ☑ `{B, C, D} = {10, foo, bar}`
- ☑ `{A, A, B} = {abc, abc, foo}`
- ☒ `{A, A, B} = {abc, def, 123}`
- ☑ `[A,B,C] = [1,2,3]`
- ☒ `[A,B,C,D] = [1,2,3]`
- ☑ `[A,B|C] = [1,2,3,4,5,6,7]`
- ☑ `[A|B] = [abc]`
- ☒ `[A|B] = []`
- ☑ `{A,_, B} = {123, 456, 789}`

Slide 7

Functions:

Function definition (in a module)

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

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

Function use

```
2> math:factorial(5).
120
```

Slide 8

Function Evaluation Rules:

- Clauses scanned until a match is found
- All variables in function head are bound
- Variables are local to each clause
- Body evaluated sequentially

Built In Functions:

- In module `erlang`.
- Do what you cannot (easily) do in Erlang
- See documentation
- See documentation (`http://www.erlang.org/documentation/doc-6.4/erts-6.4/doc/html/erlang.html`)

Anonymous Functions:

Slide 9

```
F = fun(X) -> X*2 end.
F(2).
```

Punctuation:

Easiest way to think about it:

- ➜ , is AND
- ➜ ; is OR
- ➜ . is END

Slide 10

Example:

```
factorial(0) ->
    1; % OR
factorial(N) ->
    io:format("factorial ~w~n", [N]), % AND
    N * factorial(N-1). % END
```


CONCURRENT PROGRAMMING

Processes:

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

Slide 11

Creates a new process that evaluates the given function with the given arguments

```
Pid = spawn(math, factorial, [12]).
```

With anonymous functions (most useful):

```
F = fun() -> io:format("Hello!") end.
Pid = spawn(F).
```

Message Passing:

A does:

```
B ! {self(), hello, you}
```

This sends a message `{A, hello, you}` to process B

In order to receive the message B does:

Slide 12

```
receive
    {From, Msg1, Msg2} -> ...
end
```

Processing messages:

- ➜ queue messages in arrival order
- ➜ test each message against all receive clauses – until match
- ➜ wait for more messages if no match

Slide 13

Selective Message Reception:

A: `C!foo`

B: `C!bar`

C:

```
receive
    foo -> true
end,
receive
    bar -> true
end
```

➜ `foo` is received before `bar` no matter what order they were sent in (or how they were queued).

Slide 14

Timeouts:

Wait a given amount of time (milliseconds)

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

Wait forever

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


Slide 15

0 is special

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

0 means:

➜ Check message buffer

➜ If empty execute the given code (`true`)

Slide 16

CLOSURES (VERY USEFUL)

Values of bound variables are passed along in messages

```
-module(closures).
-export([do_send/4, do_receive/0]).
do_send(Dest, A, B, C) ->
  Dest ! {msg, fun(D) ->
  io:format("A: ~s, B: ~s, C: ~s, D: ~s~n", [A, B, C, D]) end}.
do_receive() ->
  receive
      {msg, F} -> F("woohoo")
  end.
```

```
1> B = spawn(fun() -> closures:do_receive() end).
2> closures:do_send(B, "hello", "there", "friend")
```

```
A: hello, B: there, C: friend, D: woohoo
```

Slide 17

Why is Erlang Good for Distributed Systems?

1. Built-in support for message passing
2. Light-weight processes
3. Functional language:
   - ➜ no global state ➜ no concurrent access of global state
   - ➜ Note: it's possible to have global state, but *avoid* this!
4. Error handling

Slide 18

More Details

Output:

```
io:format(FormatString, ArgList)
```

Slide 19

Examples

```
1> io:format("Hello world!~n", []).
Hello world!
ok
2> io:format("arg1:~w, arg2:~w, arg3:~w", [1,2,5]).
arg1:1, arg2:2, arg3:5ok
3>
```

Guarded Function Clauses:

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

Slide 20

Examples

- `is_number(X)` - `X` is a number
- `is_atom(X)` - `X` is an atom
- `is_tuple(X)` - `X` is a tuple
- `is_list(X)` - `X` is a list
- See documentation for more (`http://www.erlang.org/documentation/doc-6.4/doc/index.html`)

Case and If:

```
case X of
     {yes, _} -> ...;
     {no, _} -> ...;
     _Else -> ...
end,
...
```

Slide 21

```
if
     is_integer(X) -> ...;
     is_tuple(X) -> ...;
     true -> ...
end,
...
```

Recursion and List Traversal:

Common patterns

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

Slide 22

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

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



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

What happens:

```
double_list([1,2,3]).
```

Slide 23

```
double_list([1,2,3]) => [2|double_list([2,3])]
double_list([2,3]) => [4|double_list([3])]
double_list([3]) => [6|double_list([])]

[2,4,6]
```

List Comprehensions:

```
List = [ X || X <- L, Filter ]
```

Slide 24

Example:

```
Y = [ 1/X || X <- List, X > 0].
```

Slide 25

Useful functions for lists:

```
lists:filter(fun(E) -> E rem 2 == 0 end, List).

lists:map(fun(E) -> E * 2 end, List).

lists:flatten([[1,2,3],[4,5,6],[[7,8], 9, [10]]]).

lists:unzip([{1,a}, {2,b}, {3,c}]). -> {[1,2,3],[a,b,c]}

lists:zip([1,2,3],[a,b,c]). -> [{1,a},{2,b},{3,c}]
```

Slide 26

Some Useful Libraries

stdlib:

`http://www.erlang.org/documentation/doc-6.4/lib/stdlib-2.4/doc/html/index.html`

- `io`: read, write, format, etc.
- `lists`: append, concat, flatten, reverse, sort, member, etc.
- `string`: len, equal, concat, substr, strip, etc.
- `dict`: new, find, store, fetch, update, etc.
- `math`: sin, cos, tan, exp, log, pow, sqrt, etc.

Slide 27

Error Handling

Try - Catch:

```
catch_error(N) ->
  try error_func(N) of
    {ok, Ret} -> io:format("SUCCES: ~w~n", [Ret])
  catch
    throw:Err -> io:format("THROW: ~w~n", [Err]);
    exit:Err -> io:format("EXIT: ~w~n", [Err]);
    error:Err -> io:format("ERROR: ~w~n", [Err])
  after
    io:format("All Done~n")
  end.

error_func(1) -> throw(woops);
error_func(2) -> exit(woops);
error_func(3) -> erlang:error(woops);
error_func(N) -> {ok, N}.
```

Slide 28

Trap Exit:

```
trapper(N) ->
  process_flag(trap_exit, true),
  Pid = spawn(fun() -> exiter(N) end),
  link(Pid),
  receive
    {'EXIT', Pid, Why} -> io:format("~w exited with ~w~n", [Pid, Why])
  end.

exiter(1) -> exit(1);
exiter(2) -> 1/0;
exiter(N) -> true.
```

Dynamic Code Loading

```
-module(dyn).
-export([start/0]).
start() -> spawn(fun() -> dyn_loop() end).
dyn_loop() -> io:format("a = ~w~n",[dyn_a:a()]), sleep(), dyn_loop().
sleep() -> receive after 3000 -> true end.
```

Slide 29

```
-module(dyn_a).
-export([a/0]).
a() -> 1.
```

```
3> dyn:start().
a = 1
a = 1
% change dyn_a.erl to return 2
4> c(dyn_a).
{ok,dyn_a}
a = 2
```

Erlang Resources

http://www.erlang.org

Documentation http://www.erlang.org/doc.html

Introductory Course (Do This!)
http://www.erlang.org/course/course.html

Slide 30

Man pages http://www.erlang.org/documentation/doc-6.4/doc/man_index.html

Erlang Books http://learnyousomeerlang.com

Programming Rules and Conventions
http://www.erlang.se/doc/programming_rules.shtml

Homework

Client-Server in Erlang:

- ➜ Simple address database server and client
- ➜ See Exercises: Client server exercise (Erlang), Part A.

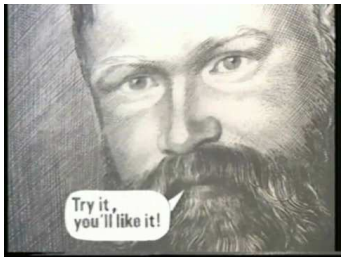
Slide 31

Hacker's edition: Performance of Erlang:

- ➜ Evaluate how long to takes to create processes in Erlang
  - ➜ How about processes on another machine?
- ➜ Evaluate how long it takes to send messages in Erlang
  - ➜ Local: same core? different cores?
  - ➜ Remote: same cluster, same LAN? over WAN?

Watch the Movie!

Slide 32



http://www.youtube.com/watch?v=uKfKtXYLG78