Concurrency in Erlang: an overview

Modelling and Verification of Reactive Systems

Luca Di Stefano

Gran Sasso Science Institute, L'Aquila, Italy

luca.distefano@gssi.it

Erlang

- Developed at Ericsson in 1986; open source since 1998
  - Immutable data types
  - Pattern matching constructs
  - Dynamic typing
  - Message passing concurrency
- "Soft real-time" systems

Some notable use cases[1]:

- Facebook (chat backend)
- Whatsapp
- Networking hardware vendors
  - Ericcson
  - RAD
  - Ubiquiti

[1] *Frequently Asked Questions about Erlang*, section *Who uses Erlang for product development?*. *Link*

Basic concepts

- *Functions* organized in *modules*
- *Processes* execute functions, run on a virtual machine (BEAM)
- Each process has a unique *Pid*

The easiest way to interact with the VM is via the Eshell:

```
$ erl
Erlang/OTP 19 [erts-8.2] [source] [64-bit] [smp:4:4]
[async-threads:10] [hipe] [kernel-poll:false] [dtrace]

Eshell V8.2  (abort with ^G)
1> io:format("Hello world!~n").
Hello world!
ok
2>
```

Example (1)

A simple echo server: will print any message it receives... with feeling.

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

echo() ->
    register(echo_server, self()),
    recv_loop().
recv_loop() ->
    receive
        {happy, Msg} ->
            io:format("Echo: ~p :)~n", [Msg]);
        {sad, Msg} ->
            io:format("Echo: ~p :(~n ", [Msg]);
        Msg ->
            io:format("Echo: ~p~n", [Msg])
    end,
    recv_loop().
```

Example (2)

Compile the module and **spawn** the server in a new process:

```
1> c(echo).
{ok,echo}
2> Pid = spawn(echo, echo, []).
<0.64.0>
```

Send messages either by using the Pid or the registered process name:

```
3> Pid ! {happy, "Hello!"}.
Echo: "Hello!" :)
{happy,"Hello!"}
4> echo_server ! {sad, "Hello!"}.
Echo: "Hello!" :(
{sad,"Hello!"}
```

Processes

Erlang processes are **not** OS processes/threads!

Data structures managed by the VM

- Lightweight (~300 words at creation time)
- No shared memory with other processes
- `spawn` is *fast* (~3 μs)[2]
- Context switch is fast

Each process is garbage-collected separately.

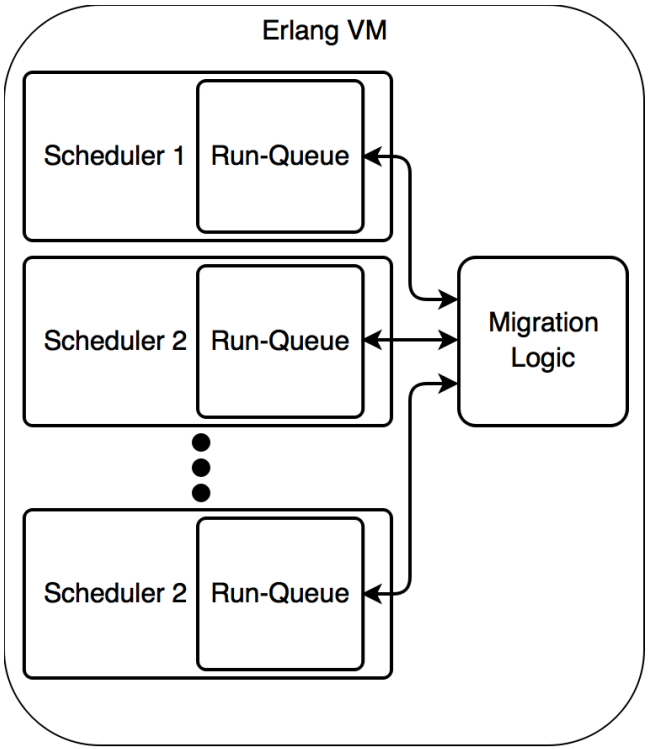
[2] J. Armstrong, *Concurrency Oriented Programming in Erlang*, 2003. *Link*

Scheduling

- One scheduler per core
- Each has its own run-queue
- **Migration logic** redistributes processes[3]

A process is released when:

- `receive`, but no pattern matches
- It calls `erlang:yield()`
- Waits for IO
- More than 2000 *reductions*:
  - Function calls
  - Sending/receiving messages
  - Garbage collection...



[3] K. Lundin, *About Erlang/OTP and Multi-core performance in particular*, Erlang Factory, 2009. *Link*.
H. Soleimani, *Erlang Scheduler details and why it matters*, 2016. *Link*

Message passing

Asynchronous, direct asymmetric, built into the language:

```
Pid ! Message          receive ... end
```

Each process has a message queue; sending a message is atomic (the VM must lock the receiver's queue)[4]. Message order is preserved on a **per-process basis**.

Example:

P1 sends A to P3	P2 sends X to P3
P1 sends B to P3	P2 sends Y to P3

Possible message queues for P3:

[A, B, X, Y]	[A, X, B, Y]
[X, Y, A, B]	[X, A, Y, B]

[4] D. Lytovchenko, *Processes ELI5,* 2016. *Link*

The actor model

First proposed by Carl Hewitt, 1973

- Actors are a concurrency primitive that share no resources between each other
- Communication is only via message passing
- Actors have a *mailbox* and a *behavior*
- An actor reads one message at a time: for each message, it can
  - Create new actors
  - Send messages
  - "Turn into" another actor (i.e. change its own behavior)

Atomicity and race conditions

How can we have race conditions if there is no shared state?

Trivial example[5]: `register/2`.

```
proc_reg(Name) ->
  ...
    case whereis(Name) of
      undefined ->
        Pid = spawn(...),
        register(Name,Pid);
      Pid ->  % already
        true  % registered
  end,
  ...
```

Suppose that two processes try to run `proc_reg/1` concurrently...

All BIFs (built-in functions) are atomic, but there is no construct to enforce atomicity of a code block!

[5] M. Christakis and K. Sagonas, *Static Detection of Deadlocks in Erlang*, 2010. *Link*

Simple mutex

```
mutex(N) ->
  io:format("MUTEX: ~B~n", [N]),
  receive
    {From, p} when N > 0 ->
      From ! ok,
      mutex(N-1);
    {From, p} ->
      From ! wait,
      mutex(N);
    {From, v} ->
      From ! ok,
      mutex(N+1)
  end.
```

```
worker(N, Mutex) ->
  timer:sleep(rand:uniform(50)),
  Mutex ! {self(), p},
  receive
    ok ->
      io:format("~B acquired mutex.~n", [N]),
      io:format("Critical section...~n"),
      Mutex ! {self(), v};
    wait ->
      io:format("~B waiting.~n", [N]),
      worker(N, Mutex)
  end.
```

- "Random queue"-like behavior
- Emulate synchronous communication via a simple messaging *protocol*

Slightly less simple mutex

```
mutex(N, Queue) ->
  io:format("MUTEX --> ~B~n", [N]),
  case (N>0) and
  not queue:is_empty(Queue) of
    false -> ok;
    true ->
      {{value, Pid}, NewQueue} =
        queue:out(Queue),
      Pid ! ok,
      mutex(N-1, NewQueue)
  end,
  receive
    {From, p} when N > 0 ->
      From ! ok,
      mutex(N-1, Queue);
    {From, p} ->
      From ! wait,
      mutex(N, queue:in(From, Queue));
    {From, v} ->
      From ! ok,
      mutex(N+1, Queue)
  end.
```

```
worker(N, Mutex) ->
  PrintMsg = "~B acquired mutex.~n",
  Mutex ! {self(), p},
  receive
    ok ->
      io:format(PrintMsg, [N]);
    wait ->
      io:format("~B waiting.~n", [N]),
      receive
        ok -> io:format(PrintMsg, [N])
      end
  end,
  io:format("Critical section...~n"),
  Mutex ! {self(), v}.
```

- FIFO-queued processes
- Empty the queue, then process new messages
- Notice the nested `receive`

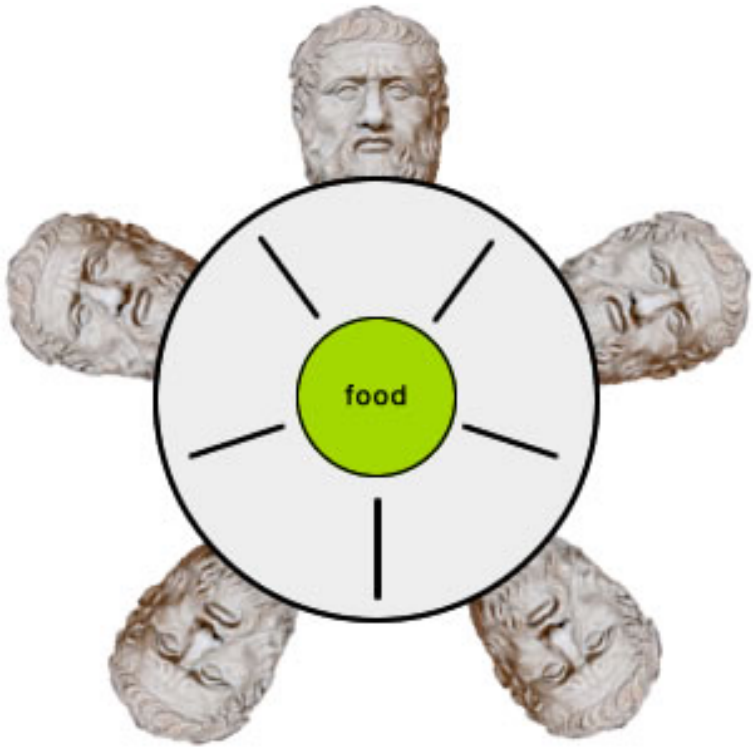
Dining Philosophers (1)

Five philosophers at a Chinese restaurant.

There are only five chopsticks on the table!

In order to eat, they agree to follow this behavior:

```
while(1) {
    Think(some_time);
    TAKE(leftChopstick);
    TAKE(rightChopstick);
    Eat(some_food);
    RELEASE(leftChopstick);
    RELEASE(rightChopstick);
}
```



Dining Philosophers (2)

```
table() ->
    table({0,0,0,0,0}).
table(Chopsticks = {C1, C2, C3, C4, C5}) ->
    receive
        {From, N, takeleft} when element(N, Chopsticks) =:= 0 ->
            From ! {self(), okleft},
            table(setelement(N, Chopsticks, N));
        {From, N, takeright} when element((N rem 5) + 1, Chopsticks) =:= 0 ->
            From ! {self(), okright},
            table(setelement((N rem 5) + 1, Chopsticks, N));
        {_, N, releaseleft} ->
            table(setelement(N, Chopsticks, 0));
        {_, N, releaseright} ->
            table(setelement((N rem 5) + 1, Chopsticks, 0));
        _ ->
            table(Chopsticks)
    end.
```

- Again, we enforce synchronous behavior by building a protocol on top of the language constructs.
- Notice the *abstence of mutable state*

Dining Philosophers (3)

```
phil(N, Table) ->
    phil(N, Table, 0, takeleft).
phil(N, Table, Meals, Msg) ->
    case Msg of
        takeleft -> timer:sleep(rand:uniform(50)); % Think(some_time)
        _ -> ok
    end,
    Table ! {self(), N, Msg},
    receive
        {_, okleft} ->
            phil(N,Table, Meals, takeright);
        {_, okright} ->
            io:format("~B is eating.~n", [N]),
            Table ! {self(), N, releaseright},
            Table ! {self(), N, releaseleft},
            phil(N,Table, Meals+1, takeleft)
    end.
```

- Deadlock-prone (can we implement a deadlock-free solution?)
- Philosophers can't starve: processes are scheduled round-robin[6]

[6] http://erlang.org/doc/man/erlang.html#process_flag-2

Barrier synchronization

```
barrier(N) ->
    barrier(N, 0, []).

barrier(N, ProcsLen, Procs) ->
    case ProcsLen == N of
        true ->
            send_continue(Procs);
        false ->
            receive
                {From, done} ->
                    barrier(N, ProcsLen+1, [From | Procs])
            end
    end.

send_continue([]) ->
    ok;
send_continue([Head | Tail]) ->
    Head ! continue,
    send_continue(Tail).
```

Types (or lack thereof)

```
receive
  X when is_integer(X) -> ...
  X when is_float(X) -> ...
end.
```

> *You might be wondering why there is no function just giving the type of the term being evaluated (something akin to* `type_of(X) -> Type`*). The answer is pretty simple. Erlang is about programming for the right cases: you only program for what you know will happen and what you expect. Everything else should cause errors as soon as possible.*[7]

[7] F. Hebert, *Learn you some Erlang for great good!*, Section "Types (or lack thereof)". *Link*