Actor Model Concurrency in Erlang

Processes and their interaction

Walter Cazzola

Dipartimento di Informatica
Università degli Studi di Milano
e-mail: cazzola@di.unimi.it
twitter: @w_cazzola

Actor Model Concurrency

Traditional (Shared-State) Concurrency

Threads are the traditional way of offering concurrency

- the execution of the program is split up into concurrently running tasks;
- such tasks operate on shared memory

Several problems

- race conditions with update loss

T_1 (`withdraw(10)`)	T_2 (`withdraw(10)`)	Balance
`if (balance - amount >= 0)`		15€
	`if (balance - amount >= 0)`	15€
	`balance -= amount;`	5€
`balance -= amount;`		-5€

- deadlocks

P_1	P_2
`lock(A)`	`lock(B)`
`lock(B)`	`lock(A)`

Erlang (and also Scala via the Akka library) takes a different approach to concurrency: the Actor Model.

Actor Model Concurrency
Overview

Each object is an actor.

- it has a mailbox and a behavior;
- actors communicate through messages buffered in a mailbox.

Computation is data-driven, upon receiving a message an actor

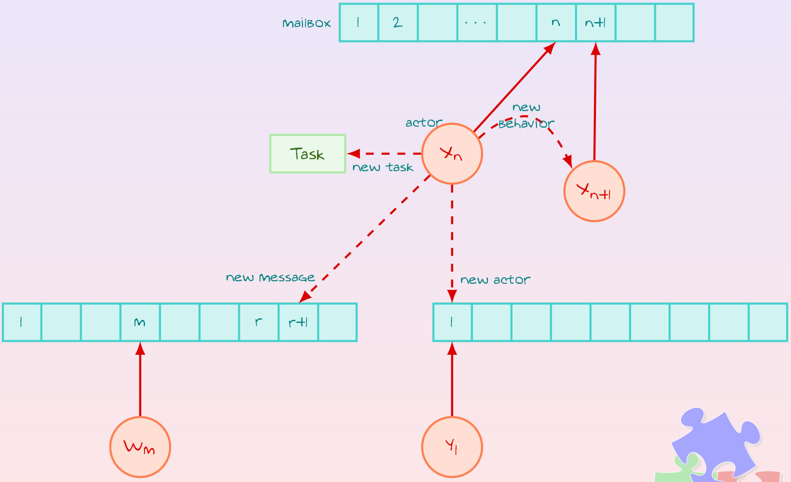
- can send a number of messages to other actors;
- can create a number of actors; and
- can assume a different behavior for dealing with the next message in its mailbox.

Note that,

- all communications are performed asynchronously;
  - the sender does not wait for a message to be received upon sending it;
  - no guarantees about the receiving order but they will eventually be delivered.
- there is no shared state between actors
  - information about internal state are requested/provided by messages;
  - also internal state manipulation happens through messages.
- actors run concurrently and are implemented as lightweight user-space threads

Actor Model Concurrency

Transaction Overview

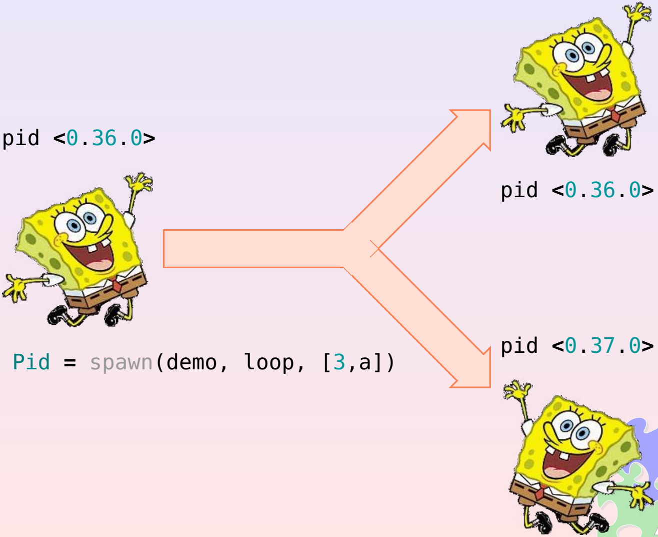


Concurrency in Erlang
Overview

Three basic elements form the foundation for concurrency

- a built-in function (spawn()) to create new actors;
- an operator (!) to send a message to another actor; and
- a mechanism to pattern-match message from the actor's mailbox.

Concurrency in Erlang
Spawning New Processes.



```
pid <0.36.0>

Pid = spawn(demo, loop, [3,a])

pid <0.36.0>

pid <0.37.0>
```

Concurrency in Erlang

My First Erlang Process.

```
-module(processes_demo).
-export([start/2, loop/2]).

start(N,A) -> spawn (processes_demo, loop, [N,A]).

loop(0,A) -> io:format("~p(~p) ~p~n", [A, self(), stops]);
loop(N,A) -> io:format("~p(~p) ~p~n", [A, self(), N]), loop(N-1,A).
```

```
1> processes_demo:start(7,a),processes_demo:start(5,b),processes_demo:start(3,c).
a(<0.73.0>) 7
b(<0.74.0>) 5
a(<0.73.0>) 6
c(<0.75.0>) 3
b(<0.74.0>) 4
<0.75.0>
a(<0.73.0>) 5
c(<0.75.0>) 2
b(<0.74.0>) 3
a(<0.73.0>) 4
c(<0.75.0>) 1
b(<0.74.0>) 2
a(<0.73.0>) 3
c(<0.75.0>) stops
b(<0.74.0>) 1
a(<0.73.0>) 2
b(<0.74.0>) stops
a(<0.73.0>) 1
a(<0.73.0>) stops
```

`self()` returns the PID of the process.

Concurrency in Erlang
Sending a Message.

Every actor is characterized by:

- an address which identifies the actor and
- a mailbox where the delivered messages but not cleared yet are stored;

Messages are sorted on arrival time (not on sending time).

To send a message to an actor:

- has to know the address (pid) of the target actor;
- to send its address (pid) to the target with the message if a reply is necessary; and
- to use the send (!) primitive.

Exp_1 ! Exp_2

- Exp_1 must identify an actor;
- Exp_2 any valid Erlang expression; the result of the send expression is the one of Exp_2;
- the sending never fails also when the target actor doesn't exist or is unreachable;
- the sending operation never block the sender.

Concurrency in Erlang
Receiving a Message.

The receiving operation uses pattern matching.

```
receive
  Any -> do_something(Any)
end
```

- the actor pick out of the mailbox the oldest message matching `Any`;
- it is blocked waiting for a message when the queue is empty.

```
receive
  {Pid, something} -> do_something(Pid)
end
```

- the actor tries to pick out the oldest message that matches `{Pid, something}`;
- if it fails the actor is blocked waiting for such a message

```
receive
  Pattern_1 [when GuardSeq_1] -> Body_1 ;
                  ...
  Pattern_n [when GuardSeq_n] -> Body_n
[after Expr_t -> Body_t]
end
```

- rules definition and evaluation is quite similar to the functions;
- when no pattern matches the mailbox the actor waits instead of raising an exception;
- to avoid waiting forever the clause **after** can be used, after Expr$_t$ ms the actor is woke up.

Concurrency in Erlang

Converting Some Temperatures.

```erlang
-module(converter).
-export([t_converter/0]).

t_converter() ->
 receive
   {toF, C} -> io:format("~p `°`C is ~p `°`F~n", [C, 32+C*9/5]), t_converter();
   {toC, F} -> io:format("~p `°`F is ~p `°`C~n", [F, (F-32)*5/9]), t_converter();
   {stop} -> io:format("Stopping`!`~n");
   Other -> io:format("Unknown: ~p~n", [Other]), t_converter()
 end.
```

```
1> Pid = spawn(converter, t_converter, []).
<0.39.0>
2> Pid ! {toC, 32}.
32 °F is 0.0 °C
{toC,32}
3> Pid ! {toF, 100}.
100 °C is 212.0 °F
{toF,100}
4> Pid ! {stop}.
Stopping!
{stop}
5> Pid ! {toF, 100}. % once stopped a message to such a process is silently ignored
{toF,100}
```

Concurrency in Erlang
Calculating Some Areas.

```erlang
-module(area_server).
-export([loop/0]).

loop() ->
    receive
        {rectangle, Width, Ht} ->
            io:format("Area of rectangle is ~p~n",[Width * Ht]),
            loop();
        {circle, R} ->
            io:format("Area of circle is ~p~n", [3.14159 * R * R]),
            loop();
        Other ->
            io:format("I don't know how to react to the message ~p~n",[Other]),
            loop()
    end.
```

```
1> Pid = spawn(fun area_server:loop/0).
<0.34.0>
2> Pid ! {rectangle, 30, 40}.
Area of rectangle is 1200
{rectangle,30,40}
4> Pid ! {circle, 40}.
Area of circle is 5026.544
{circle,40}
5> Pid ! {triangle,22,44}.
I don't know what the area of a {triangle,22,44} is
{triangle,22,44}
```

Concurrency in Erlang
Actor Scheduling in Erlang.

Actors are not processes and are not dealt by the operating system

- the BEAM uses a preemptive scheduler;
- when an actor run for a too long period of time or when it enters a **receive** statement with no message available, the actor is halted and placed on a scheduling queue;

Actors and the rest of the system

- OS processes and actors have different schedulers and long running Erlang applications do not interfere with the execution of the OS processes (no one will become unresponsive)
- the BEAM supports symmetric multiprocessing (SMP)
  - i.e., it can run processes in parallel on multiple CPUs
  - but it cannot run lightweight processes (actors) in parallel on multiple CPUs.

Concurrency in Erlang
Timing the Spawning Process.

```erlang
-module(processes).
-export([max/1]).

max(N) ->
  Max = erlang:system_info(process_limit),
  io:format("Maximum allowed processes:~p~n",[Max]),
  statistics(runtime), statistics(wall_clock),
  L = for(1, N, fun() -> spawn(fun() -> wait() end) end),
  {_, Time1} = statistics(runtime), {_, Time2} = statistics(wall_clock),
  lists:foreach(fun(Pid) -> Pid ! die end, L),
  U1 = Time1 * 1000 / N, U2 = Time2 * 1000 / N,
  io:format("Process spawn time = ~p (~p) microseconds~n", [U1, U2]).

wait() -> receive die -> void end.

for(N, N, F) -> [F()];
for(I, N, F) -> [F()|for(I+1, N, F)].
```

```
1> processes:max(20000).
Maximum allowed processes:32768
Process spawn time = 2.5 (3.4) microseconds
ok
2> processes:max(40000).
Maximum allowed processes:32768

=ERROR REPORT==== 8-Nov-2011::14:24:32 ===
Too many processes
...
[16:48]cazzola@surtur:~/lp/erlang>erl +P 100000
1> processes:max(50000).
Maximum allowed processes:100000
Process spawn time = 3.2 (3.74) microseconds
ok
```

Concurrency in Erlang
Giving a Name to the Actors.

Erlang provides a mechanism to render public the pid of a process to all the other processes.

- register(an_atom, Pid)
- unregister(an_atom)
- whereis(an_atom) -> Pid|undefined
- **registered**()

Once registered

- it is possible to send a message to it directly (name!msg).

```erlang
-module(clock).
-export([start/2, stop/0]).

start(Time, Fun) -> register(clock, spawn(fun() -> tick(Time, Fun) end)).
stop() -> clock ! stop.

tick(Time, Fun) ->
  receive
    stop -> void
  after
    Time -> Fun(), tick(Time, Fun)
  end.
```

```
5> clock:start(5000, fun() -> io:format("TICK ~p~n",[erlang:now()]) end).
true
TICK {1320,769016,673190}
TICK {1320,769021,678451}
TICK {1320,769026,679120}
7> clock:stop().
stop
```

References

- Gul Agha.
  Actors: A Model of Concurrent Computation in Distributed Systems.
  MIT Press, Cambridge, 1986.
- Joe Armstrong.
  Programming Erlang: Software for a Concurrent World.
  The Pragmatic Bookshelf, fifth edition, 2007.
- Francesco Cesarini and Simon J. Thompson.
  Erlang Programming: A Concurrent Approach to Software Development.
  O'Reilly, June 2009.