

# Process state and hot code swapping

## Module 8 - Erlang tutorial 6

---

Operating systems and process oriented programming 2018

1DT096

# Computation

What do we mean by computation?

How is computation performed in imperative programming languages?

How is computation performed in functional programming languages?

# Computation

**Imperative** programming defines computation as statements that changes the **mutable** program **state**.

**Functional** programming treats computation as the **evaluation** of **mathematical functions** and avoids changing state and mutable data.

# **Erlang system initialisation**

In Erlang systems it is common for a module to provide a start function that initialises the system, for example by spawning a new process.

# The Erlang process loop

In Erlang, the recursive function used by a process to handle messages is called the process loop.

# **Erlang process state**

The state of an Erlang process is defined by the values of the arguments to the recursive process receive loop function.

**An example  
of a **stateful****

**Erlang  
process**

In Erlang systems it is common for a module to provide a start function that initialises the system, for example by spawning a new process. The recursive function used by a process to handle messages is often named loop.

```
-module(state).  
-export([start/0]).  
  
start() ->  
    spawn(fun() -> loop(0) end).
```

The state of the process executing the **loop/1** function is defined by the value of the argument to the **loop/1** function. In this example the initial state is **0**.



```
loop(N) when is_integer(N) ->
    receive
        inc ->
            %% Update the process state by using a
            %% new value for the argument to the
            %% proces loop.
            loop(N+1);
        dec ->
            %% Update the process state by using a
            %% new value for the argument to the
            %% proces loop.
            loop(N-1);
        show ->
            io:format("N = ~w~n", [N]),
            %% Don't update the process state.
            loop(N)
    end.
end.
```

```
-module(state).  
-export([start/0]).
```

A complete example module  
implementing a stateful process

```
start() ->  
    spawn(fun() -> loop(0) end).  
  
loop(N) when is_integer(N) ->  
    receive  
        inc ->  
            loop(N+1);  
        dec ->  
            loop(N-1);  
        show ->  
            io:format("N = ~w~n", [N]),  
            loop(N)  
    end.
```

```
erlang> c(state).  
{ok,state}  
erlang> C = state:start().  
<0.156.0>  
erlang> C ! show.  
N = 0  
show  
erlang> C ! inc.  
inc  
erlang> C ! inc.  
inc  
erlang> C ! show.  
N = 2  
show  
erlang> C ! dec.  
dec  
erlang> C ! show.  
N = 1  
show  
erlang>
```

Compile the module.

Start a new counter process and save the Pid of the process in variable **C**.

Initially the counter state **N = 0**.

The counter state is changed by sending messages to the process.

- ▶ The **inc** message increases **N**.
- ▶ The **dec** message decreases **N**.
- ▶ The show message prints the value of **N**.

# Ask for the current counter value

Let's make it possible to ask the counter process for the current counter value.

- ▶ Assume the counter process got pid **Counter**.
- ▶ To ask for the counter value we send the message **Counter ! {get, self()}** to the counter process.
- ▶ When receiving a **{get, Pid}** message the counter process will respond with the message **Pid ! {value, N}**.

loop(N) when is\_integer(N) ->

receive

inc ->

loop(N+1);

dec ->

loop(N-1);

show ->

io:format("N = ~w~n", [N]),

loop(N);


{get, Pid} ->

Pid ! {value, N},

loop(N);

end.

Make it possible to  
ask for the current  
counter value.



```
erlang> f().
ok
erlang> c(state).
{ok,state}
erlang> self().
<0.156.0>
erlang> C = state:start().
<0.156.0>
erlang> C ! inc, C ! inc.
inc
erlang> C ! {get, self()}.
{get,<0.45.0>}
erlang> flush().
Shell got {value,2}
ok
erlang>
```

Use **f()** to free all bound variables in the Erlang shell.

Compile the state module.

Use **self()** to get the pid of the Erlang shell.

Start a new counter process with initial counter state **N = 0** and store the pid in variable **C**

Increase the counter twice.

Ask the counter for the counter value.

Use **flush()** to print all messages in the shell mailbox.

# Interface to a stateful process

It is common to provide a set of functions that hide the underlying message passing.

- ▶ These functions becomes the interface to the stateful process .
- ▶ The underlying message passing becomes part of the implementation hidden from the user of the interface.

Lets add a **start/1** function that makes it possible to set the initial counter value.

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

```
start(N) ->  
    spawn(fun() -> loop(N) end).
```



```
inc(Counter) ->  
  Counter ! inc,  
  ok.
```

```
dec(Counter) ->  
  Counter ! dec,  
  ok.
```

```
show(Counter) ->  
  Counter ! show,  
  ok.
```

Hide the  
message  
passing  
and return  
the atom  
**ok.**

Here we hide the message passing and return the current counter value.

```
get(Counter) ->  
  Counter ! {get, self()},  
  receive  
    {value, Value} ->  
      Value  
  end.
```

Export all interface functions.

```
-module(state).  
-export([start/0,  
        start/1,  
        inc/1,  
        dec/1,  
        show/1,  
        get/1]).
```

```
erlang> c(state), f(), C = state:start(7).
```

```
<0.126.0>
```

```
erlang> state:inc(C).
```

```
ok
```

```
erlang> state:get(C).
```

```
8
```

```
erlang> state:dec(C), state:dec(C).
```

```
ok
```

```
erlang> state:get(C).
```

```
6
```

```
erlang>
```

# Hot swapping

Hot swapping is replacing or adding components without stopping or shutting down the system.

# Hot code swapping in Erlang

Erlang supports change of code  
in a running system.

Code replacement is done on the  
module level.

# Current and old module version

The code of a module can exist in two variants in a system: **current** and **old**.

- ▶ When a module is loaded into the system for the first time, the code becomes **current**.
- ▶ If then a new instance of the module is loaded, the code of the previous instance becomes **old** and the new instance becomes **current**.

# Fully qualified function calls

When including the module name when calling a function, the function is said to be called using the fully qualified function name.

The following is an example of a fully qualified function call to the `bar/1` function in the `foo` module.

```
foo:bar(77).
```

**Note** that a function must be exported from a module in order to allow for fully qualified function calls.



# ?MODULE

The **?MODULE** macro returns the name of the current module.

To make a fully qualified function call to the **bar/1** function in the current module the **?MODULE** macro can be used as follows.

```
?MODULE:bar(77).
```

Using the **?MODULE** macro makes the code valid even if the name of the module is changed.

# **A fully qualified function call triggers hot code swapping**

Both the old and the current version of a module code is valid, and can be evaluated concurrently.

- ▶ Fully qualified function calls always refer to current code.
- ▶ Old code can still be evaluated because of processes lingering in the old code.

# **An example of hot code swapping in Erlang**

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

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

```
result(X) -> 2*X.
```

```
loop() ->  
    receive
```

```
        swap ->
```

```
            io:format("Hot code swapping!~n"),
```

```
            ?MODULE:loop();
```

```
X when is_integer(X) ->
```

```
    io:format("Result = ~w~n", [result(X)]),
```

```
    loop();
```

```
X ->
```

```
    io:format("Unhandled message: ~p~n", [X]),
```

```
    loop()
```

```
end.
```

After receiving the atom **swap**, a fully qualified recursive function call **?MODULE:loop()** is made to the process loop.

```
erlang> c(swap).  
{ok, swap}  
erlang> P1 = swap:start().  
<0.189.0>  
erlang> P2 = swap:start().  
<0.191.0>  
erlang> P1 ! 7.  
Result = 14  
7  
erlang> P2 ! 33.  
Result = 66  
33  
erlang>
```

Compile the module.

Create two processes,  
both running the  
**swap:loop/0** function.

When receiving an integer  
**X**, both processes  
calculates **2\*X**.

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

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

```
result(X) -> 10*X.
```

```
loop() ->  
    receive  
        swap ->  
            io:format("Hot code swapping!~n"),  
            ?MODULE:loop();  
        X when is_integer(X) ->  
            io:format("Result = ~w~n", [result(X)]),  
            loop();  
        X ->  
            io:format("Unhandled message: ~p~n", [X]),  
            loop()  
    end.
```

Now, change **result/1** from returning **2\*X** to returning **10\*X** and save the file.

```
erlang> c(swap).  
{ok, swap}  
erlang> P1 ! swap.  
Hot code swapping!  
swap  
erlang> P1 ! 7.  
Result = 70  
7  
erlang> P2 ! 33.  
Result = 66  
33
```

Compile the module.

Send the atom **swap** to the **P1** process.

- ▶ When receiving an integer **X**, **P1** now calculates **10\*X**, i.e., **P1** runs the new version of the module.
- ▶ When receiving an integer **X**, **P2** continues to calculate **2\*X**, i.e., **P2** runs the old version of the module.

# **The Erlang reference manual**

You can read more about hot code swapping  
in the Erlang Reference Manual.

[http://erlang.org/doc/  
reference\\_manual/  
code\\_loading.html](http://erlang.org/doc/reference_manual/code_loading.html)