Advanced Functional Programming, 1DL450 2012

Lecture 3, 2012-11-05

Cons T Åhs

Receiving messages

```
foobar() ->
  F = fun(Msg) ->
    {message_queue_len, L} = process_info(self(), message_queue_len),
    io:format("Msg: ~p (~p)~n", [Msg, L])
  end,

  receive M0=foo -> F(M0) end,
  receive M1=bar -> F(M1) end,
  foobar().
```

- A `receive` will wait until a message matching a specified pattern is in the queue.
- Messages are processed in an order specified by the `receives` in the process
- Messages are thus not necessarily processed in the order they arrive
- The code
  - reports queue length when acting on a message
  - messages are processed in the sequence `foo, bar, foo, bar, ..`
  - Note use of binding pattern in `receive`
  - Why can't we have the same variable in both `receives`

Receiving messages

- What if you don't want to wait?
  - add an `after` *`time`* clause which will be executed after *`time`* (in ms) has passed
  - use exit(PID, kill) when you get tired of the impatient process

```
impatient() ->
  receive M ->
      io:format("Msg: ~p~n", [M])
  after 5000 ->
      io:format("Hello..~n", [])
end,
impatient().
```

Linking processes

- Send a message (with `Pid ! Message`) returns the message.
  - This happens even if the process has died
  - No delivery receipt
  - if `process_info(Pid) == undefined` the process is not alive
  - querying the process status is impractical
- A process will run until it
  - terminates normally
  - is killed by someone else
  - is killed by an accident
- A system with several processes will not work if one process ceases to exist
  - default is that process death is ignored - no one cares
  - The rest of system needs to know about the death of other processes
  - Possible actions
    - take down other processes
    - restart dead process
    - restart several other processes

Linking processes

- Processes can be tied together with *links*
- Two (of several) ways to create links
  - `link(Pid)` - link current process with `Pid`
  - `spawn_link(Fun)` - create new process and link it with current process
- Linking processes means linking their destiny
  - Links are bidirectional
  - Without additional considerations in place, a process P_0 linked to P_1 will terminate if P_1 terminates (and vice versa)
    - This is (slightly) better since we'll have no silent sending of messages to dead processes.
- A process that dies/exits will send a signal to linked processes and they will react by dying as well.

Linking processes

```
failing() ->
  receive
    X ->
      io:format("failing, msg: ~p~n", [X]),
      X=elrang,
      failing()
    end.
```

```
124> f(P), P = spawn(fun() -> linking:failing() end).
<0.300.0>
125> P!foo.
failing, msg: foo
foo

=ERROR REPORT==== 4-Nov-2012::09:57:41 ===
Error in process <0.300.0> with exit value:
{{badmatch,elrang},[{linking,failing,0}]}
```

Linking processes

```
parent() ->
  Child = spawn_link(fun() -> failing() end),
  receive
    M ->
      io:format("Parent, msg: ~p~n", [M]),
      Child ! M,
      parent()
  end.

f(P), P = spawn(fun() -> linking:parent() end).
<0.314.0>
132> P!bar.
Parent, msg: bar
bar
failing, msg: bar
133>
=ERROR REPORT==== 4-Nov-2012::10:03:17 ===
Error in process <0.315.0> with exit value:
{{badmatch,elrang},[{linking,failing,0}]}

P!hello.
hello
134>
```

Linking processes

- Much better is to be made aware of a linked process being in trouble
- Catch the signal, convert it to a message and act upon it.
- This is the base for building robust systems that act upon failures

```
responsible_parent() ->
  process_flag(trap_exit, true),
  care_for().

care_for() ->
  Child = spawn_link(fun() -> failing() end),
  care_for(Child).

care_for(Child) ->
  receive
    {'EXIT', Child, Why} ->
       io:format("child died (reason: ~pn), restart it~n", [Why]),
       care_for();
    M ->
       io:format("Parent, msg: ~p~n", [M]),
       Child ! M,
       care_for(Child)
  end.
```

Behaviours

- A *behaviour* in Erlang specifies the *interface* of a module
  - A module *must* implement the functions specified by the behaviour
  - It can implement and export more functions
  - A module that implements a behaviour can then be passed to a generic module expecting that behaviour
  - This can also rather easily be implemented using higher order functions

Behaviours

- The actual behaviour is specified by the function `behaviour_info/1`
- It should return a list of tuples `{functionname, arity}`
- The actual implementation making use of the implementation can be in the same module defining the behaviour or in another module.
- There is no checking that the module supplied actually implements the behaviour - this is discovered at runtime.
- Example: implement a generic module for caching the values of a (pure) function call. Since the actual computation might take a long time, we want to avoid computing the function several times.
- General idea:
  - Receive a “function call”
  - Check the cache if we already have computed the value
    - If so, return the value (no change in the cache)
    - If not, compute the value, add it to the cache and return the value

```
-module(cachefun).

-export([init/1 , behaviour_info/1]).

behaviour_info(callbacks) -> [{compute, 1}];
behaviour_info(_) -> undefined.

init(Module) ->
  Cache = dict:new(),
  Pid = spawn(fun() -> loop(Cache, Module) end),
  fun(X) ->
      Pid ! {self(), X},
      receive V -> V end
  end.

loop(Cache, Module) ->
  receive {Pid, Arg} ->
   case dict:find(Arg, Cache) of
     {ok, Value} ->
       NewCache = Cache;
     error ->
       Value = Module:compute(Arg),
       NewCache = dict:store(Arg, Value, Cache)
   end,
      Pid ! Value,
      loop(NewCache, Module)
  end.
```

Behaviours

- `fibfun()` returns a function
- `?MODULE` is a macro returning the module name

```
-module(fibcache).

-behaviour(cachefun).

-export([compute/1, fibfun/0]).

fibfun() -> cachefun:init(?MODULE).

compute(N) -> fib(N).

fib(0) -> 0;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).


3> F= fibcache:fibfun().
#Fun<cachefun.1.45378360>
4> F(40).
```

Standard behaviours

- `gen_server` - implements a generic server, supporting
  - request/response (synchronous calls)
  - commands (requests without response, or asynchronous calls)
  - code upgrade
  - You implement the specific details for handling state and responding to the calls, the generic server takes care of the rest
- `supervisor` - implements generic functions for supervising processes, i.e., how the different processes should react when process die etc.
- `gen_fsm` - finite state machine; you code the states, events and transistions and the generic machine takes care of the rest.

Code loading

- One core feature of Erlang is the ability to load new code during runtime
- To cater for scenarios where you "long" running processes Erlang actually supports holding two versions (current and old) of a module at a given time.
- When a new version is loaded the old is thrown away, the (previously) current becomes the old and newly loaded becomes the current.
- This works for external calls, i.e., a module calls another using a module prefix.
- For an internal call a name always refers to the code version in the module
  - a process holding a reference to an old module might fail due to the code being unloaded and thrown away
- This is "solved" by always calling with the module prefix, but it also means that the function has to be exported.
  - the current (newest) version is always called

```
-module(server).

-export([loop/1]).

loop(State) ->
  <wait for messages and compute new state>,
  server:loop(NewState).
```

Binaries

- The telecom world is full of protocols, often at a very low level, i.e., 3 bits for this, followed by 7 bits for that etc.
- Erlang makes it very easy to manipulate bit strings, treating them in a very nice abstract manner.
- External syntax <<. .>> where .. is a sequence of bit field specifiers
- A binary is a datatype in the same way as numbers, terms, lists etc
  - integers must be converted to and from binaries
- Instead of masking and shifting one can extract bitfields through matching
- Similarly, one can construct a binary the same way.

```
decode_parts(<<T:1, F:3, U:2, S:2>>) ->
  {T==1, F, U, S}.

encode_parts({Flag, F, U, S}) ->
  T = if Flag -> 1;
         true -> 0
      end,
  <<T:1, F:3, U:2, S:2>>.
```

Binaries

- Decoding an IP (V4) datagram

```
ip_datagram(Dgram) ->
  Size = byte_size(Dgram),
  case Dgram of
    <<?IP_VERSION:4, HLen:4, SrvcType:8, TotLen:16,
      ID:16, Flgs:3, FragOff:13,
      TTL:8, Proto:8, HdrChkSum:16,
      SrcIP:32,
      DestIP:32, RestDgram/binary>> when HLen>=5, 4*HLen=<Size ->
      OptsLen = 4*(HLen - ?IP_MIN_HDR_LEN),
      <<Opts:OptsLen/binary,Data/binary>> = RestDgram,
      ...
  end.
```

Storage and Persistence

- Any real life application will have the need to handle larger amounts of data
  - in memory (with pragmatic access)
  - persistently (still there after a restart)
  - efficient access (constant)
  - distributed
- Erlang provide several options
  - process dictionary - “global storage” for a process (limited use)
  - ets - erlang term storage, table based, in memory, belongs to a process
  - dets - disk based ets, persistent (similar to ets in operations, but slower)
  - mnesia - database built on which support transactions and distribution

Macros

- Erlang the possibility to write macros, i.e., expressions expanded at compile time
- Use for abstraction and avoiding DRY violations
- Defined as one would expect
- Can take argument
- Use with leading ?

```
-define(HDR_LEN, 4).
-define(PROTO_VERSION, 3).

-record(person, {name, age, length}).

-define(person_name(P), (P#person.name)).

extract_header(<<Hdr:?HDR_LEN, _/bitstring>>) -> Hdr.

greeting(Person) ->
  "Hello, " ++ ?person_name(Person) ++ "!".
```

Macros

- You can do clever things like define a correct if, but you have to be careful.
- One of these definitions work, the other does not - why?

```
-define(IFA(C, E1, E2),
        _E1=fun() -> E1 end,
        _E2=fun() -> E2 end,
        case C of
          true  -> _E1();
          false -> _E2()
        end).


-define(IFB(C, X, Y),
        (fun(T, TE, FE) ->
           case T of
             true -> TE();
             false -> FE()
           end
         end)(C, fun() -> X end, fun() -> Y end)).
```

Macros

- There is no safe way to introduce new variable names in a macro, so you have to be careful to protect any free variables from capture.

```
-define(convert(Expr), case Expr of
                         {ok, Value}  -> Value;
                         {error, Rsn} -> throw(Rsn)
                       end).
```

```
f1() ->
  Value = 666,
  ?convert(f(Value)).
%%% expands to
f1() ->
    Value = 666,
    case f(Value) of
        {ok,Value}  -> Value;
        {error,Rsn} -> throw(Rsn)
    end.
```

Macros

```
-define(Q, ?).
-define(P(X), {X}).
-define(R, baz).
-define(lp, ().
-define(rp, )).
-define(c, ,).
-define(foo, ?bar).
-define(bar, f).

q() -> ?Q foo ?lp 1 ?c 2 ?rp .

a() -> ?Q Q Q Q R.

b() -> ? Q P (foo).
```

- Macros are text replacement at compile time so you can do horrible things..
- Macros are *only* text replacement at compile time so there are a lot of things that you cannot do
- We will return to macros when talking about Common Lisp

Erlang Summary

- Untyped language with a functional core.
- Evolved rather designed.
- Designed for fault tolerance, distribution and robustness.
- Excellent handling of processes.
- Not an excellent language for abstraction and “normal” software engineering.
- Not so well designed in terms of syntax and some semantics.
- Some rather horrible constructions.

- Uncovered topics
  - most of the standard libraries (otp)
  - tools surrounding development and releases
  - lots of details