# Erlang – concurrency

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

# Process creation

- An Erlang process is a self-contained entity; own heap and stack;
- Processes do not share memory;
- All comunication through message passing;
- Processes created by `spawn`:
  - passing module, function and list of arguments:

```
Pid = spawn(Module, Function, Args)
```

  - passing a closure:

```
F = fun() -> ... end,
Pid = spawn(F)
```

- Process terminates when function returns; return value ignored;

# Message sending

- Erlang follows the Actor model:
  - each process has a mailbox – message queue;
  - messages are sent specifying destination process – no other concept, such as channel, is used.
- Message sent is performed with the CSP inspired syntax:

```
Pid ! Msg
```

  - Pid is the identifier of the destination process;
  - Msg is an arbitrary Erlang term (including closures);
- Sending is asynchronous and does not return errors:
  - does not wait for message to reach destination or be delivered;
  - if the destination process has already terminated the message is lost, but the sender is not notified.

# Message receiving

- Messages are kept in the mailbox by arrival order;
- A message may be removed from the mailbox with:

```
receive
  Pattern1 [when GuardSeq1] ->
    Seq1;
  ...
  PatternN [when GuardSeqN] ->
    SeqN
end
```

- Pattern matching is used:
  - the first message from the mailbox that matches any pattern (whose optional guard succeeds) is removed;
  - in such case, the corresponding sequence is executed;
  - otherwise, the process blocks until such message arrives;

# Order in the mailbox and guard order

- The choice of which message from the mailbox is selected:
  - first considers the order of messages in the mailbox;
  - for each message, the order of guards is then used;
- Example:
  - if mailbox contains messages:

    ```
    [msg1, msg2, msg4]
    ```

    and we execute:

    ```
    receive
      msg4 ->
        ...
      msg2 ->
        ...
    end
    ```

    the message selected is msg2, even though it is the last case;
- Conclusion: the order of patterns cannot be used to implement a message priority scheme;

# Waiting for messages from a given process

- There is no special mechanism to send replies or choose the origin of a message;
- Pattern matching can be used, together with sending pids in messages;
- A process can send its own pid in a message:

```
Pid ! {self(), ... }          % self() - current process pid
```

- Another process can decide to wait for a message from that process, using a variable with that process pid in a receive:

```
From = ...
receive
  {From, Msg} ->
    doSomething(Msg)
end
```

# Timeouts

- A timeout can be used in a receive:

```
receive
  ...
  after TimeoutExpr ->
    Seq
end
```

- The timeout value is in miliseconds;
- If no message is selected, after this time `Seq` is executed;
- There are two special cases for the timeout value:
  - `infinity` means wait forever;
  - 0 means imediate return if no message in the mailbox is selected;

# Some uses for timeouts

- A sleep function, to block the current process T miliseconds:

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

- A function to empty the mailbox, discarding messages:

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

# Uses for timeouts – a simple message priority scheme

- Suppose when want to treat different kinds of messages in the mailbox with different priority;
- To give more priority to msg1, then msg2, . . . :

```
receive
  msg1 ->
    Seq1
  after 0 ->
    receive
      msg2 ->
        Seq2
      after 0 ->
        ...
    end
end
```

# Registered processes

- Many times it can be convenient to be able to contact some process without having to know its pid;
- E.g., having some main singleton processes of an application, without having to be forced to disseminate their pids everywhere;
- Erlang has, in each node, a simple name service that maps names (atoms) to pids;
- It can be used with:
  - `register(Name, Pid)` – associates the atom Name to Pid;
  - `unregister(Name)` – removes the association;
  - `whereis(Name)` – returns corresponding pid or `undefined`;
  - `registered()` – returns list of registered names;
- Registered names can be used in the send primitive:

```
Pid = spawn(...),
register(server, Pid),
Msg = ...,
server ! Msg
```

# Example: counter

- Implementing a counter abstraction, managed by a process;
- This process should handle two kinds of messages:
  - increment the counte;
  - know the current value;
- Function counter defines the process behaviour:

```
counter(Val) ->
  receive
    increment ->
      counter(Val + 1);
    {value, From} ->
      From ! Val,
      counter(Val)
  end.
```

- The parameter Val keeps the current value;
- Tail recursion is used to simulate looping, with the new state;
- To know the counter value, a process sends its pid in message, so that the counter manager can send a reply;

# Example: counter – a client

- A function to ask the counter to be incremented several times;
- Two parameters:
  - which counter (which manager process);
  - how many times to increment the counter;

```
client(_, 0) ->
  true;
client(Counter, N) ->
  Counter ! increment,
  client(Counter, N - 1).
```

# Example: counter – another client

- Function printVals asks the counter value, in infinite loop;
- Sends its own pid, so that a reply can be sent;

```
printVals(Counter) ->     % infinite loop
  Counter ! {value, self()},
  receive
    Value ->
      io:format("Counter value:~w~n", [Value])
  end,
  printVals(Counter).
```

- We are matching any message in the receive;
- This would not work if more than one type of message or messages from different processes could be arriving;
- An antipattern – should not be used like this;

# Example: counter – main

- Main function:
  - creates a counter process;
  - creates N client processes;
  - executes printVals, looping forever;

```
-module(counter).
-export([start/2]).

start(N, Iters) ->
  Counter = spawn(fun() -> counter(0) end),
  createClients(Counter, Iters, N),
  printVals(Counter).

createClients(_, _, 0) ->
  true;
createClients(Counter, Iters, N) ->
  spawn(fun() -> client(Counter, Iters) end),
  createClients(Counter, Iters, N - 1).
```

# Example: counter – new version with interface functions

- In the previous version, clients know the message protocol used to request operations;
- It is often useful to encapsulate the protocol, hiding it from clients, and make corresponding interface functions available;
- This allows changing the protocol without rewriting client code;
- We will make available 4 interface functions:
  - create counter: creates process and returns its pid;
  - increment counter – takes the counter pid as parameter;
  - obtain the counter value;
  - destroy the counter, ending the process;
- We will also strengthen the protocol when sending the reply to a process that wants to know the counter value;
- We will put the counter in a module, and the client in another.

# Example: counter – interface functions

```
-module(counter).
-export([create/0, increment/1, value/1, stop/1]).

create() ->
  spawn(fun() -> counter(0) end).

increment(Counter) ->
  Counter ! increment.

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

stop(Counter) ->
  Counter ! stop.
```

# Example: counter – counter process

- The new version of the counter function:

```
counter(Val) ->
  receive
    increment ->
      counter(Val + 1);
    {value, From} ->
      From ! {self(), Val},
      counter(Val);
    stop ->
      true;
    _ ->                    % discard other unexpected messages
      error_logger:error_report(...),
      counter(Val)
  end.
```

- In the reply to value() the counter pid is sent; more robust;
- A case of a stop message to end the process is added;
- A wildcard pattern to discard any other message is added;

# Example: counter – clients

- Clients are now simpler;
- The protocol is encapsulated by the interface functions;

```
start(N, Iters) ->
  Counter = counter:create(),
  createClients(Counter, Iters, N),
  printVals(Counter).

createClients(_, _, 0) -> true;
createClients(Counter, Iters, N) ->
  spawn(fun() -> client(Counter, Iters) end),
  createClients(Counter, Iters, N - 1).

client(_, 0) -> true;
client(Counter, N) ->
  counter:increment(Counter),
  client(Counter, N - 1).

printVals(Counter) ->    % infinite loop
  V = counter:value(Counter),
  io:format("Counter value:~w~n", [V]),
  printVals(Counter).
```

# Example: Read/Write Lock

- Mutual exclusion abstraction controlling two modes of access:
  - readers: want to perform read-only access to a resource;
  - writers: want to perform updates on a resource;
- Useful even in message passing systems; e.g., to access files;
- A block of access operations is surrounded by sincronization requests; there are classically 4 operations:
  - acquireRead and releaseRead to surround read access;
  - acquireWrite and releaseWrite to surround write access;
- Better to have only two operations:
  - acquire; specifying mode, either read or write;
  - release; does not make sense to specify mode;
- Safety:
  - several processes can be reading;
  - if a process is writing, no other process can be accessing, either reading or writing.
- Liveness: this problem leads easily to starvation.

# Example: Read/Write Lock – interface functions

```
create() ->
  spawn(fun released/0).

acquire(Pid, Mode) when Mode == read; Mode == write ->
  Pid ! {Mode, self()},
  receive acquired -> true end.

release(Pid) ->
  Pid ! {release, self()}.
```

# Example: Read/Write Lock – server process; write starvation

```
released() ->
  receive
    {read, Pid} -> Pid ! acquired, reading([Pid]);
    {write, Pid} -> Pid ! acquired, writing(Pid)
  end.

reading([]) -> released();
reading(Readers) ->
  receive
    {read, Pid} -> Pid ! acquired, reading([Pid | Readers]);
    {release, Pid} -> reading(Readers -- [Pid])
  end.

writing(Pid) ->
  receive
    {release, Pid} -> released()
  end.
```

# Example: Read/Write Lock – server process; no starvation

```
released() ->
  receive
    {read, Pid} -> Pid ! acquired, reading([Pid]);
    {write, Pid} -> Pid ! acquired, writing(Pid)
  end.

reading([]) -> released();
reading(Readers) ->
  receive
    {read, Pid} -> Pid ! acquired, reading([Pid | Readers]);
    {release, Pid} -> reading(Readers -- [Pid]);
    {write, Pid} -> reading(Readers, Pid)
  end.

reading([], Writer) -> Writer ! acquired, writing(Writer);
reading(Readers, Writer) ->
  receive
    {release, Pid} -> reading(Readers -- [Pid], Writer)
  end.

writing(Pid) ->
  receive
    {release, Pid} -> released()
  end.
```