Programming Paradigms

Unit 17 — Erlang Processes

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE

Outline

Outline

Processes

- Processes are the fundamental units of concurrency in Erlang
- They communicate with each other via messages
- Processes are also the basic container for program state in Erlang
  - No shared state, which makes it easier to proof the correctness of programs

Basic Primitives for Processes

- There are 3 basic primitives for processes
  - Creating a new process (with `spawn`)
  - Sending a message (with `!`, pronounced "bang")
  - Receiving a message (with `receive`)
- That's enough to get us started
- Let's write a simple translation process that gets a word in Spanish and replies with an English translation
  - The process should run in a loop, waiting for words to translate

Translation Example/1

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

loop() ->
    receive
        "casa" ->
            io:format("house~n"),
            loop();
        "blanca" ->
            io:format("white~n"),
            loop();
        _ ->
            io:format("I don't understand~n"),
            loop()
    end.
```

Translation Example/2

- The first two lines define the module `translate` and the export of function `loop/0`
- The next block defines the function `loop()`

```
loop() ->
        ...
    end.
```

- `loop` is called recursively in the body of the function
- This is OK, since it is tail-recursion, and Erlang is optimized for tail-recursion
- `loop()` is essentially an empty function that loops forever

Translation Example/3

- The next block is the function `receive`

```
receive ->
    ...
```

- This function will receive a message from another process
- `receive` works similar to the other pattern matching constructs, such as the `case` statement and function definitions
- It tries to match a received message to one of the matching clauses

```
"casa" ->
    io:format("house~n"),
    loop();
```

- If statements in a matching clause span more than one line, they are separated by commas

Running a Process

- After compiling the module `translate`, we can create a process running the function `loop`

```
>c(translate).
{ok,translate}
```

```
> Pid = spawn(fun translate:loop/0).
<0.130.0>
```

- To spawn a process, function `spawn` is used
  - It takes a function as argument, starts this function in a new process, and returns the process ID
- Now we have the process (with ID `0.130.0`) up and running
  - Variable `Pid` stores the process ID
- So far, the process is not doing much: it's just sitting there waiting for messages

Sending Messages

- Let us send some messages to the process

```
> Pid !  "casa".
house
"casa"
> Pid !  "blanca".
white
"blanca"
> Pid !  "loco".
?
"loco"
```

- The ! operator (pronounced "bang") is used to send messages and has the following general form: `Pid ! message`
  - `Pid` is any process ID
  - `message` can be any value, e.g., primitive values, lists, tuples

Variations on Processes

- You can also spawn processes on a remote machine using a slightly different syntax

```
Pid = spawn(node@server,function).
```

- Similar, it is possible to send messages to processes running on other nodes

```
{Pid, node@server} !  message
```

Outline

Messaging

- What we've just implemented is called asynchronous messaging
- In asynchronous messaging, the sender sends a message, but does not wait actively for a reply
  - E-mails and SMS text messages are asynchronous
- In synchronous messaging, the sender sends a message and actively waits for the response
  - Phone calls and loading a web page are synchronous

Synchronous Messaging

- To change the message model to synchronous messaging we need to do the following steps:
  1. Each `receive` clause will also have to match the process ID of the requesting sender (in addition to the "original" message, e.g., the word)
  2. Each `receive` clause has to send a response to the sender (instead of, e.g., just printing the result)
  3. On the sender side, instead of using `!`, we'll write a simple function that sends a request and waits for the response

Synchronous Messaging – Receiver

- First, we rewrite the `receive` clause

```
receive
    ...
    {Pid,"casa"} ->
        Pid !  "house",
        loop();
    ...
```

- Instead of just matching a word, we match a tuple consisting of the process ID of the sender and a word
- Instead of printing the result, we send it back to the requesting process

Synchronous Messaging – Sender/1

- Starting a process with the new modified `loop` function and sending something to it is not enough

```
> Trans = spawn(fun translate2:loop/0).
<0.144.0>

> Trans !  {self(),"casa"}.
{<0.61.0>,"casa"}
```

- We send the correct tuple to the process, but we don't pick up its answer
- Function `self()` returns the own process ID
- Next we'll write a function that will send a message and wait for the reply

Synchronous Messaging – Sender/2

- In the synchronous model, the sender must send a message and then immediately wait for a response
- Given a process ID in `Receiver`, a general sender looks as follows

```
Receiver ! {self(),"message"},
    receive
        Message -> do_something_with(Message)
    end.
```

- The sender uses a `receive` function to wait for the response

Synchronous Messaging – Sender/3

- Since we are using the translation service frequently, we encapsulate the request for a translation into a new function

```
translate(To,Word) ->
    To !  {self(),Word},
    receive
        Translation -> "The translation is:" ++ Translation
    end.
```

- The complete module is shown on the next slide

Synchronous Messaging – Sender/4

```
-module(translate2).
-export([loop/0]).
-export([translate/2]).

loop() ->
    receive
        {Pid, "casa"} ->
            Pid !  "house",
            loop();
        {Pid, "blanca"} ->
            Pid !  "white",
            loop();
        {Pid, _} ->
            Pid !  "???",
            loop()
    end.

translate(To,Word) ->
    To !  {self(),Word},
    receive
        Translation -> "The translation is:  " ++ Translation
    end.
```

Synchronous Messaging – Sender/5

- After compiling it, we can spawn a process running `loop` and then send messages using `translate`

```
> Trans = spawn(fun translate2:loop/0).
<0.39.0>

> translate2:translate(Trans,"blanca").
"The translation is:  white"

> translate2:translate(Trans,"xxxx").
"???"
```

- The new version of `translate` sends a message and then waits for the reply

Outline

Adding Reliability

- Erlang has exception handling for catching errors in a piece of code
  - This is very similar to what Java offers, so we are not covering it here
- In addition to this Erlang provides process links
  - This is a system for handling process failures
  - We are going to have a closer look at process links

Process Links

- Whenever an Erlang process dies unexpectedly, an exit signal is generated
- All processes linked to the dying process receive that signal and can react accordingly
- By default, the receiver will exit as well (sending another exit signal)
  - So you can have a whole cascade of exiting processes
- However, you can overwrite the default behaviour and react in an appropriate way

- What is the advantage of this?
  - Allows you to have a group of processes behave as a single application
  - You don't have to worry about leftover processes still running

Supervision

- You don't always want to shut down a process when receiving an exit signal
- Someone needs to be there to restart parts of the system when receiving exit signals
- These so-called supervisor processes need to be able to overwrite the default exiting behavior
  - This can be done by trapping an exit signal, i.e., you get informed, but don't exit yourself
- Non-trapping processes are usually called worker processes
- Let's look at an example

Russian Roulette/1

- First, let's build a process that can be killed deliberately

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

loop() ->
    receive
        3 ->
            io:format("bang!~n"),
            exit({roulette,die,at,erlang:time()});
        _ ->
            io:format("click.~n"),
            loop()
    end.
```

- The code is essentially a message loop:
  - Matching 3 kills the process by calling function `exit`
  - Anything else prints a message and goes back to the top of the loop

Russian Roulette/2

- Let's start the process and try it out

```
> Gun = spawn(fun roulette:loop/0).
<0.39.0>
> Gun ! 1.
click.
1
> erlang:is_process_alive(Gun).
true
> Gun ! 3.
bang!
3
> erlang:is_process_alive(Gun).
false
```

- Function `erlang:is_process_alive(PID)` checks whether process PID is running

Monitoring Processes/1

- Now let's build a monitor process that tells us whether a process dies by trapping exit signals

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

loop() ->
    process_flag(trap_exit,true),
    receive
        {monitor,Process} ->
            link(Process),
            io:format("Monitoring process.~n"),
            loop();
        {'EXIT',From,Reason} ->
            io:format("~p died:  ~p~n",[From,Reason]),
            io:format("Please start another one.~n"),
            loop();
        _ ->
            io:format("Unexpected message received.~n"),
            loop()
    end.
```

Monitoring Processes/2

- The first step in the loop is to register the process as one that will trap exit signals:

  ```
  process_flag(trap_exit,true)
  ```

- otherwise exit signals are not received

Monitoring Processes/3

- The `receive` gets two types of tuples
  - Tuples beginning with atom `monitor`

```
{monitor,Process} ->
    link(Process),
    io:format("Monitoring process.~n"),
    loop();
```

    - Links the `coroner` process (i.e., the `loop/0` function that implements the monitor) to the process `Process`
    - Hence, if `Process` dies, the `coroner` process gets a message
  - Tuples beginning with atom `'EXIT'`

```
{'EXIT',From,Reason} ->
    io:format("~p died:  ~p~n",[From,Reason]),
    io:format("Please start another one.~n"),
    loop();
```

    - PID of dying process is printed together with the reason
    - The user is asked to start another process

Monitoring Processes/4

- After compiling the modules `coroner` and `roulette` we create processes and ask process `Coroner` to monitor process `Gun`

```
> Coroner = spawn(fun coroner:loop/0).
<0.44.0>

> Gun = spawn(fun roulette:loop/0).
<0.46.0>

> Coroner ! {monitor,Gun}.
Monitoring process.
{monitor,<0.46.0>}

> Gun ! 3.
bang!
3
<0.46.0> died: {roulette,die,at,{14,42,57}}
Please start another one.
```

Coroner

- The module `coroner` does not do much at this point
- It only notices that the `roulette` process died
- We are going to improve the module by
  - moving the creation of a new `roulette` process into this new process
  - automatically respawning a new `roulette` process if it gets killed
  - registering the `roulette` process ID with an atom called `gun`
    - So a user does not have to remember the PID to play

Meet the Doctor/1

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

loop() ->
    process_flag(trap_exit,true),
    receive
        new ->
            io:format("Creating and monitoring new process.~n"),
            register(gun,spawn_link(fun roulette:loop/0)),
            loop();
        {'EXIT',From,Reason} ->
            io:format("~p died:  ~p~n",[From,Reason]),
            io:format("Restarting.~n"),
            self() !  new,
            loop();
        _ ->
            io:format("Unexpected message received.~n"),
            loop()
    end.
```

Meet the Doctor/2

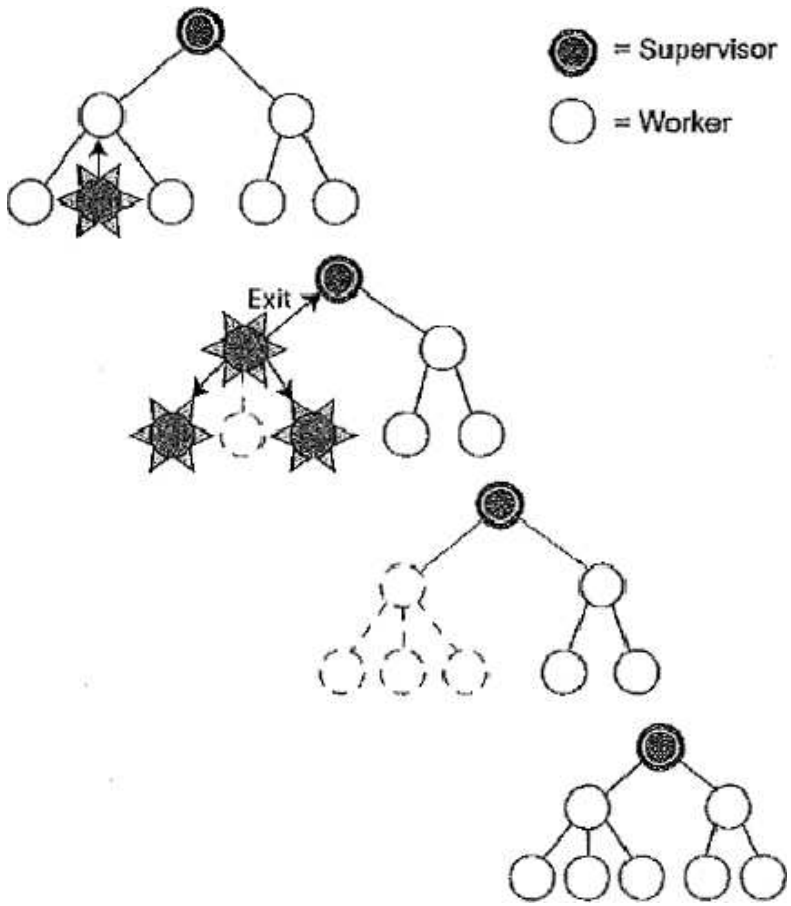
- `spawn_link` creates a new process and links it to the calling process
  - Hence, `doctor` will be notified whenever a `roulette` process dies
- `register(gun,...)` binds the PID returned by `spawn_link` to the atom `gun`
  - Users can now send messages to this process by using `gun ! message`
- For restarting a `roulette` process, the `doctor` process just sends the message `new` to itself
- Now let's have a look

Meet the Doctor/3

```
> Doc = spawn(fun doctor:loop/0).
<0.44.0>
> Doc ! new.
Creating and monitoring new process.
new
> gun ! 1.
click.
1
> gun ! 3.
bang!
3
<0.48.0> died: {roulette,die,at,{15,0,32}}
Restarting.
Creating and monitoring new process.
> gun ! 1.
click.
1
```

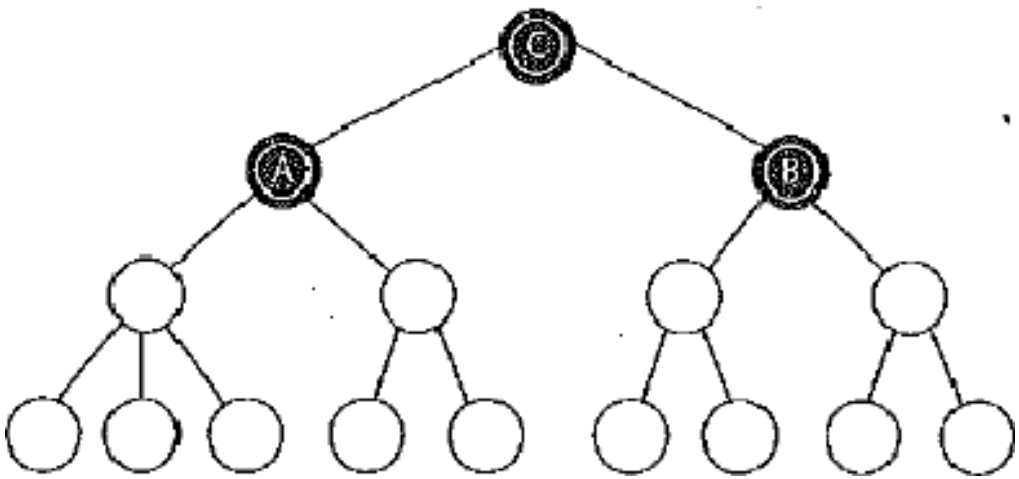
Managing Subsystems/1

- Usually a supervisor monitors more than one process
- Typically it manages different groups of processes
- These subsystems can then be cleanly restarted
- On the right hand side, one of the processes in the left subgroup crashes . . .
- . . . the whole subgroup is terminated and restarted



Managing Subsystems/2

- Usually you should build a whole supervision tree with multiple layers of supervisors
- This gives you a finer granularity in terms of "rebooting" certain parts of the system



Open Telecom Platform

- We were only able to cover a small part of Erlang
- The Open Telecom Platform (OTP) is a powerful package that helps Erlang reach its full potential
- It's not specific to telecom applications and helps you in
  - writing stable and reliable code (OTP has been thoroughly used and tested)
  - providing frameworks for applications
  - offering functionality for code upgrades

Summary – Strengths of Erlang

- The shared-nothing, message-passing process model is very powerful when it comes to implementing concurrency
  - Concurrency means any execution order (e.g., parallel or serial) without compromising the correctness of the program
- Erlang offers a lot in terms of reliability and fault tolerance
  - Controlled crash
- Erlang was developed with the aim to achieve industrial-strength high performance
- Erlang processes run on a virtual maching that automatically adapts to the underlying hardware
  - Runs on as many cores/machines as available
- Language supports some powerful features of functional and logic-oriented languages
  - e.g., pattern matching, optimized for tail-recursion
- OTP provides a lot of functionality to make it easier to implement concurrent applications

Summary – Weaknesses of Erlang

- The syntax of the language is a weird mix of Prolog with functional language constructs thrown in
- While Erlang shines when it comes to concurrency, programming simpler (serial) things tend to be harder than in other languages