Elixir Concurrency

Programming II - Elixir Version

Johan Montelius

Spring Term 2018

Concurrency

Elixir was designed for concurrent programming. You will quickly learn how to divide your program into communicating processes and thereby give it far better structure. Try the following:

```
defmodule Wait do

    def hello do
    receive do
        x -> IO.puts("aaa! surprise, a message: #{x}")
    end
    end

end
```

The `IO.puts` procedure will output the string to the stdout and insert the `x` value by means of string interpolation. Compile and load the above module in the Elixir interactive shell `iex` (the returned PID number may be different):

```
iex(1)> c("wait.ex")
[Wait]

iex(2)> p = spawn(Wait, :hello, [])
#PID<0.92.0>
```

The variable `p` is now bound to the *process identifier* of spawned process. The process was created and called the procedure `hello/0` (this is how we name a function with zero arguments). It is now suspended waiting for incoming messages. In the same Elixir `iex` shell execute the command:

```
iex(3)> send p, "hello"
...
```

Now register the process identifier under the name `:foo` after having started a new process (the one above died after having received the message):

```
iex(4)> p = spawn(Wait, :hello, [])
#PID<0.99.0>

iex(5)> Process.register(p, :foo)
true

iex(6)> send :foo, "hello"
...
```

1 Tic-Tac-Toe

In the example above the only thing we sent was a string but we can send arbitrary complex data structures. The `receive` statement can have several clauses that try to match incoming messages. Only if a match is found will a clause be used. Try this:

```
defmodule Tic do

    def first do
      receive do
        {:tic, x} ->
          IO.puts("tic: #{x}")
          second()
      end
    end

    defp second do
      receive do
        {:tac, x} ->
          IO.puts("tac: #{x}")
          last()
        {:toe, x} ->
          IO.puts("toe: #{x}")
          last()
      end
    end

    defp last do
      receive do
        {t, x} ->
          IO.puts("#{t}: #{x}")
      end
    end

end
```

Then in the `iex` shell execute the following commands:

```
iex(1)> c("tic.ex")
[Tic]

iex(2)> p = spawn(Tic, :first, [])
#PID<0.103.0>

iex(3)> send p, {:toe, :bar}
...

iex(4)> send p, {:tac, :gurka}
...

iex(5)> send p, {:tic, :foo}
...
```

In what order where they received by the process? Note how messages are queued and how the process selects in what order to process them.