# 10 - Processos

André Rauber Du Bois
dubois@inf.ufpel.edu.br
Computação - CDTec - UFPel

# Processos

- Um **processo** é uma unidade de código que executa de forma **concorrente (ao mesmo tempo)** que outros processos
- A máquina virtual do Elixir/Erlang possuí **escalonadores** que distribuem os processos entre os **cores** disponíveis na máquina
- **Processos** não compartilham memória, ou seja, cada processo possui a sua própria área de memória
- Processos se comunicam através de **troca de mensagens**

**Erlang/OTP 25 [erts-13.1.3] [source] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:1] [jit:ns]**

**[smp:8:8]: Minha máquina possui 8 cores e existem 8 escalonadores ativos, um para cada core**

# Processos

- Para pedir para um processo executar algo, por exemplo, temos que enviar uma **mensagem** para esse processo
- Cada processo possui uma **caixa postal** onde as mensagens recebidas são colocadas
- Processos podem retirar mensagens da **caixa postal** através de **casamento de padrões**
- Se sabemos o nome (**pid**) de um processo, podemos enviar mensagens para esse processo
- **Tudo em Elixir executa dentro de processos!!!**

# Spawn

- **spawn/1**: recebe uma função e executa essa função em um novo processo. Devolve como resposta um **PID (Process Identifier)**

**iex(1)>  spawn (fn () -> IO.puts "Alo do Processo!" end)**

**Alo do Processo!**

**#PID<0.110.0>**

```
iex(1)> pid = spawn(fn -> IO.puts "Alo mundo!" end)
Alo mundo!
#PID<0.110.0>
iex(2)> Process.alive?(pid) // verifica se o processo continua rodando
false                       // este processo não está vivo pois terminou
iex(3)> Process.alive?(self()) // sef() retorna o pid do processo corrente
true                           // o processo corrente obviamente está vivo
```

# Troca de mensagens: send/receive

- **send/2:** serve para enviar uma mensagem para um processo. Recebe como argumentos o **pid** do processo e a **mensagem. Qualquer valor do Elixir pode ser enviado em uma mensagem**
- **receive/0:** usada por um processo para retirar uma mensagem de sua **caixa postal**. Permite o uso de **guardas/casamento de padrões** (assim como o **cond** e o **case**). Retira da caixa postal a primeira mensagem que se encaixa no padrão. O **receive é bloqueante:** caso não exista mensagem na caixa postal, o processo espera até que uma mensagem chegue

# Exemplo de script send/receive

```
defmodule Aula10 do
 def my_process() do
   receive do
     msg -> IO.puts "Mensagem recebida: #{msg}"
   end
 end
end

pid = spawn(&Aula10.my_process/0)

send(pid,"Alo!!!")
```

# Comunicação entre processos diferentes **(script 1/2)**

```
defmodule Aula10 do
 def resp_process() do
   receive do
     {:soma, n1,n2,pid} -> send(pid,n1+n2)
                           resp_process()
     {:mult, n1,n2,pid} -> send(pid,n1*n2)
                           resp_process()
     :die                -> IO.puts("Tchau!")
   end
 end
end
```

# Comunicação entre processos diferentes **(script 2/2)**

```
pid = spawn_link(&Aula10.resp_process/0)

send(pid,{:mult, 10, 20, self()})
receive do
 mult -> IO.puts("10 * 20 = #{mult}")
end
send(pid,{:soma, 50 , 25, self()})
receive do
 soma -> IO.puts("50 + 25 = #{soma}")
end
send(pid,:die)
```

# Processos que mantém um estado

- Algumas vezes queremos que o processo **mantenha um estado**
- Nesse caso as mensagens **modificam esse estado**
- **Exemplo:** o processo mantém um banco de dados (ex: lista) com os itens de uma loja e as mensagens são usadas para inserir ou retirar itens da loja
- Em Elixir, os **processos não acessam uma memória compartilhada**
- Para resolver esse problema, a função com **o código do processo deve receber como argumento o estado**
- Após atender uma mensagem, o processo faz uma **chamada recursiva para o seu código passando como argumento a nova versão do estado**

# Processo que mantém um estado **(script 1/2)**

```
defmodule Aula10 do
 def proc_conta_mensagens(n) do
   receive do
     {:die, pid} -> send(pid, {:total, n})
     _msg -> proc_conta_mensagens(n+1)
   end
 end
end
```

# Processo que mantém um estado **(script 2/2)**

```
pid = spawn_link(fn -> Aula10.proc_conta_mensagens(0) end)

send(pid, "mensagem1")
send(pid, "mensagem2")
send(pid, "mensagem3")
send(pid, "mensagem4")
send(pid, "mensagem5")
send(pid,{:die,self()})

receive do
 {:total,n} -> IO.puts("O processo recebeu #{n} mensagens")
end
```

# Dando nomes aos processos

- Podemos registrar um nome para um processo:

```
pid = spawn(&Aula10.proc/0)
Process.register(pid, :meu_processo)
```

- Após o registro, podemos enviar mensagens usando o **nome ao invés do pid:**

```
send(:meu_processo,"Alo!!!!")
```