Programming Distributed Systems

Concurrency in Elixir - Exercise

Philipp Lersch

RPTU Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

Overview

- Concurrency in Elixir
- Concepts
- Modules

Processes

- Contain logic and state while alive
- Have mailboxes for data exchange
- Can be linked and supervised
- Provide concurrency
- Can exist on different nodes
- Identifiable by PID or registered alias

Working with processes

Create a new process with $spawn/1$

```
> pid = spawn(fn -> IO.puts("I'm alive!") end)
#PID<0.113.0>
```

Create a new process with $spawn\_link/1$

```
> pid = spawn_link(fn -> IO.puts("I'm alive!") end)
#PID<0.113.0>
```

Check state of processes with $alive?/1$

```
> Process.alive?(pid)
false
```

Query your own PID with $self/0$

```
> pid = self()
#PID<0.105.0>
```

You can print PIDs with $IO.inspect()$

Checkout https://hexdocs.pm/elixir/processes.html

Ready, steady - go!

Elixir processes communicate via

A. shared memory.

B. message passing.

C. channels.

Message passing

To pass messages it is nessecary to know the PID of the target:

Send a message to an processes' mailbox with $send/3$

```
> send(pid, "Hi :)")
:ok
```

Check for available matching messages in the processes' mailbox with $receive\ do\ ...\ end$

```
> receive do
>   {:hi, sender, m} -> IO.puts(m)
>   ...
>   match_all -> IO.inspect(match_all)
> end
```

→ Prevent memory leaks with catch all clauses

→ The receive statement blocks until a message from the mailbox is pattern matched

What does process Q print?

process P

```
def p do
    send(q, {self(), 0})
    send(q, {self(), 2})
end
```

process Q

```
def q do
    receive do {p, N} -> IO.inspect(N+1) end,
    receive do {p, M} -> IO.inspect(M+1) end.
end
```

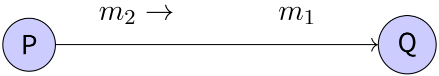
1. 0 and 2, in any order
2. 0 and then 2
3. 1 and then 3
4. 1 and 3, in any order

Message passing

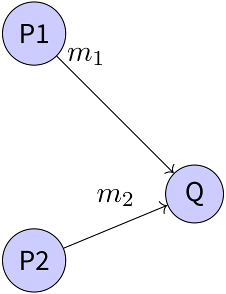
- Messages are sent **asynchronously**
- Any value can be sent as a message.
- Each process has a **message queue** (mailbox)
  - Arriving messages are placed in the queue
  - No size limit ⇒ Can turn into memory leak!
  - Message are removed from the queue when they fit a pattern on which a process receives next
  - Can take only one message at a time
- If no message in the mailbox matches, the process will wait till matching message arrives or time-out limit is reached.
- Messages to terminated processes will be discarded without warning.

Message passing

Same sender and receiver: **FIFO order**



Different senders, same receiver: **No guaranteed order**



Creating a Server

A server has state and reacts to arbitrary many incoming messages.

```
> defmodule ServerName do
>   def listen do
>     receive do
>       {:ping, source} -> send(source, :pong)
>       ...
>       match_all -> IO.inspect(match_all)
>     end
>     listen()
>   end
> end
```

→ Rerun the receive statement to process more than one message

→ This server pattern can provide features like sharing state and async computation

Use provided Abstractions

Use specialized abstractions instead of rewriting a server pattern

- Agent: Update and retrieve state
- Registry: Local, decentralized and scalable key-value storage
- Task: Await the result of async operations
- Supervisor: Supervise child processes and act if they crash
- GenServer: Keeps state and executes code asynchronously

→ All have additional features to aid debugging, fault tolerance and integrate with other advanced features

→ Checkout https://hexdocs.pm/elixir/GenServer.html

GenServer: Client-side

A GenServer is an abstraction for server processes

A client can connect to a GenServer with $start/3$ and $start\_link/2$

```
> {:ok, pid} = GenServer.start(GenImplName, "inital data")
```

A client can send messages to a GenServer with $cast/2$ (async)

```
> GenServer.cast(pid, {:push, argument})
```

A client can send AND receive messages with $call/3$ (sync)

```
> result = GenServer.call(pid, {:push, argument})
```

$\rightarrow$ Similar to a server process:

spawn $\approx$ start/3

send $\approx$ cast/2 and call/3

GenServer: Server-side I

A GenServer's functionality is defined through callbacks

These callbacks should return the resulting GenServer's state

A GenServer calls init on start-up

```
> defmodule GenImplName do
>   use GenServer
>
>   @impl true
>   def init(argument) do
>     initial_state = ...
>     {:ok, initial_state}
>   end
>   ...
> end
```

→ Receives the argument passed to $start$ and $start\_link$

→ Creates the GenServer's inital state

GenServer: Server-side II

A GenServer also calls $handle\_cast/2$ and $handle\_call/3$

```
> defmodule GenImplName do
>   use GenServer
>   ...
>   @impl true
>   def handle_cast(argument, state) do
>     ...
>     {:noreply, new_state}
>   end
>   @impl true
>   def handle_call(argument, caller, state) do
>     ...
>     {:reply, to_caller, new_state}
>   end
>   ...
> end
```

→ Hint: Pattern match e.g. arguments in the function head