Episode 13: Processes

Processes

- Spawning processes
- Messaging
- Linking
- Monitoring

Spawning processes

- Don't share memory; completely isolated from each other
- Beam process != O.S. process: doesn't have the same downsides with threads; not computationally expensive.

Using the `spawn` function:

```
# spawn/1
spawn fn ->
  IO.puts "This will run in a separate process"
end

# spawn/3
spawn SomeModule, :some_function, [arg1, arg2]
```

> *"Typically developers do not use the spawn functions, instead they use abstractions such as Task, GenServer and Agent, built on top of spawn, that spawns processes with more conveniences in terms of introspection and debugging." source*

Messaging

Spawning a process returns a *Process Identifier*, or `pid` (a core Elixir data type):

```
pid = spawn(fn -> ... end)

# Get the current process's pid with self()
self()
```

```
# Send a message to a process
send pid, :message
```

Receiving messages

- Messages stay in mailbox until handled; add a default case to prevent the mailbox from filling up and crashing the process.

Using the `receive` macro:

```
receive do
  message -> # do something with the message
end
```

Gets the first available message, or if there are none, blocks process until a message is available.

`receive` supports pattern matching:

```
receive do
  {:say, msg} ->
    IO.puts(msg)
  {:think, msg} ->
    Logger.debug(msg)
  _other ->
    # default case
end
```

Wait only a specified time (in ms) with `after`:

```
receive do
  message -> process(message)
after 500 ->
  # do something
end
```

Example

Normally, `receive` will exit after processing a message because there is no work left to do. However, in this example we tell it to call itself recursively after any message is read.

> This is safe because recall that Elixir has tail-call optimization!

```
defmodule Speaker do
  def speak(prefix \\ "") when is_binary(prefix) do
    receive do
      {:say, msg} when is_binary(msg) ->
        IO.puts(prefix <> msg)
        speak(prefix)
      _other ->
        speak(prefix)
      end
    end
end

pid = spawn(Speaker, :speak, ["Says: "])
send(pid, {:say, "some message!"})
# => Says: some message!
```

Killing a process

See `Process.exit/2`.

```
pid = spawn(fn -> ... end)
Process.exit(pid, :kill)
```

By default, if a spawned process dies (either by some error, a full mailbox or being manually killed), the spawner will not be notified. It is completely isolated from the rest of the system.

Linking processes together

Tie two processes together with `spawn_link`. If one process dies, the other will die as well.

```
romeo = self
juliet = spawn_link(fn -> ... end)

# Causes the current process to also exit, because it is linked to the
↪  process we are killing.
Process.exit(juliet, :kill)
```

Process death with the `:trap_exit` flag. The current process will receive an `:EXIT` message when the linked process dies (probably a good idea).

```
Process.flag(:trap_exit, true)
juliet = spawn_link(fn -> ... end)

receive do
  {:EXIT, pid, reason} ->
    # Revive Juliet?
end
```

Spawn monitor

Monitor spawned processes with `spawn_monitor`. It returns `{pid, ref_to_monitor}`. When the process dies, it sends a `:DOWN` message rather than an `:EXIT` message.

```
{juliet, _ref} = spawn_monitor(fn -> ... end)

receive do
  {:DOWN, _ref, :process, pid} ->
    # Revive Juliet?
end
```