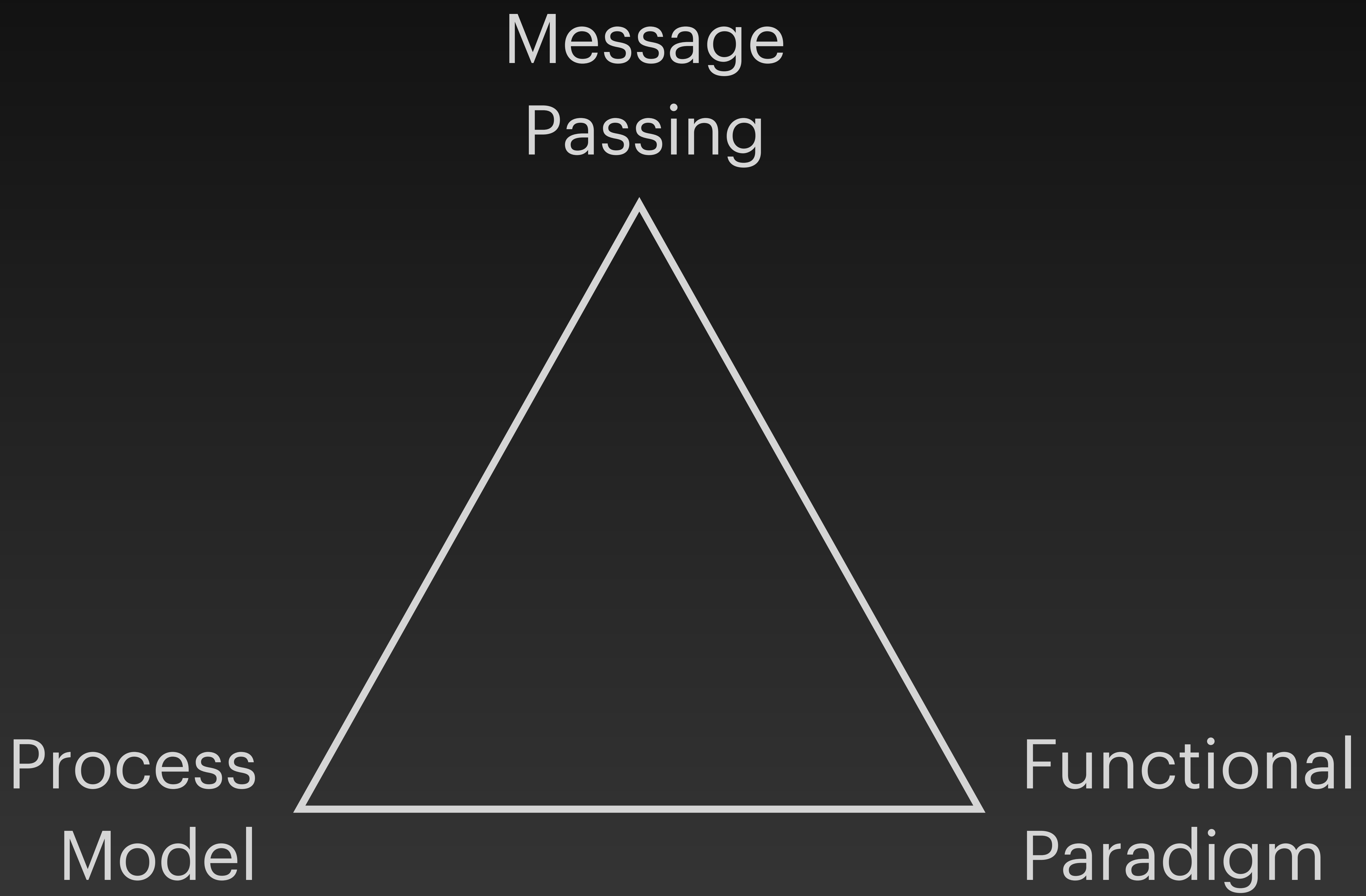
Back to basics with processes

Mat Trudel // mat@geeky.net // github.com/mtrudel/talks

The Soul of Elixir

(with apologies to Saša Jurić)



The Process Model

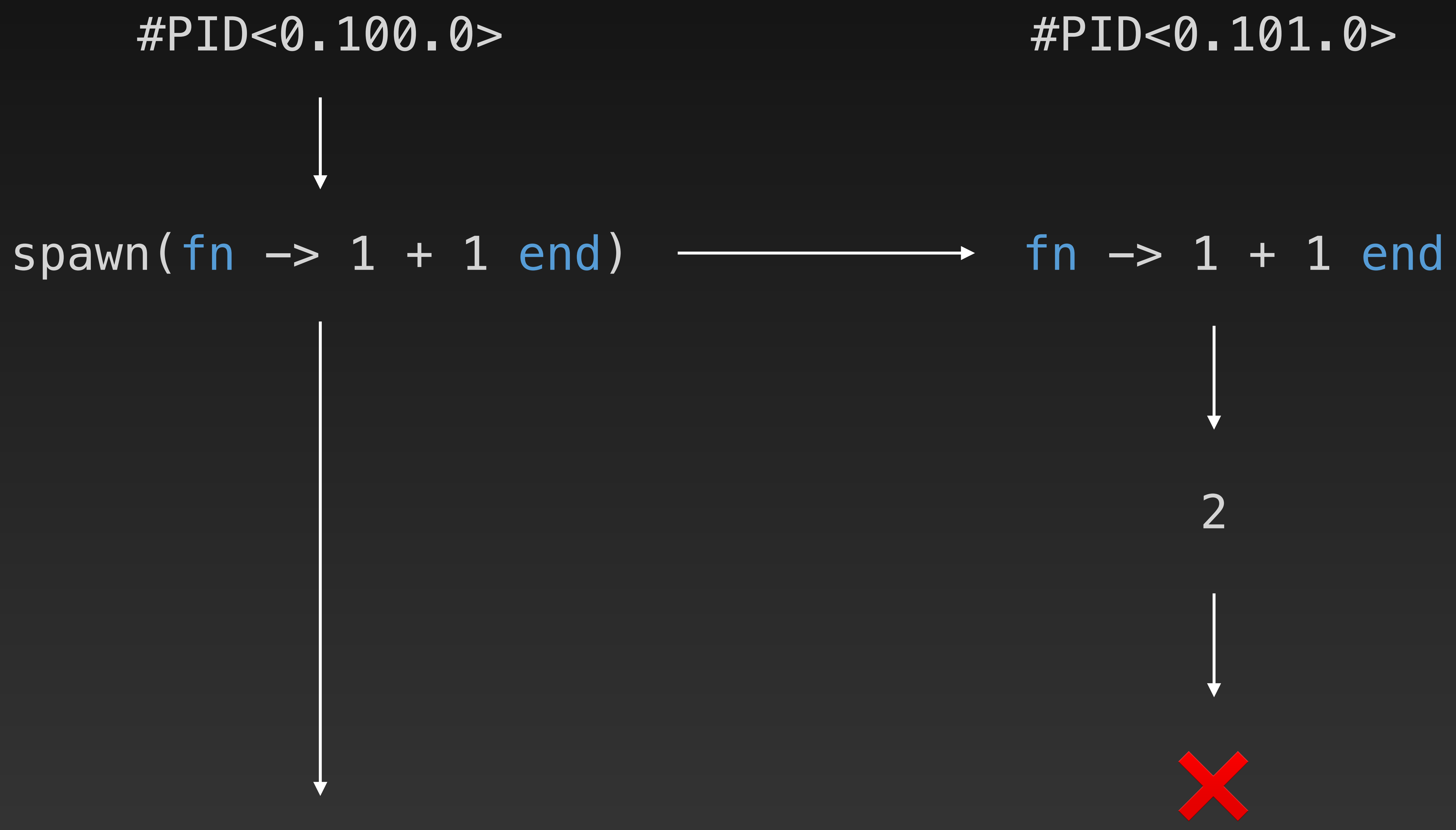
The simplest thing

Kernel.spawn/1

```
spawn(fn -> 1 + 1 end)
```

The simplest thing

Kernel.spawn/1



The Process Model

- No shared state
- Lifetime tied to initial function call; duration arbitrary
- Processes always spawned from one another
- Also *linking* and *monitoring* (not important for now)

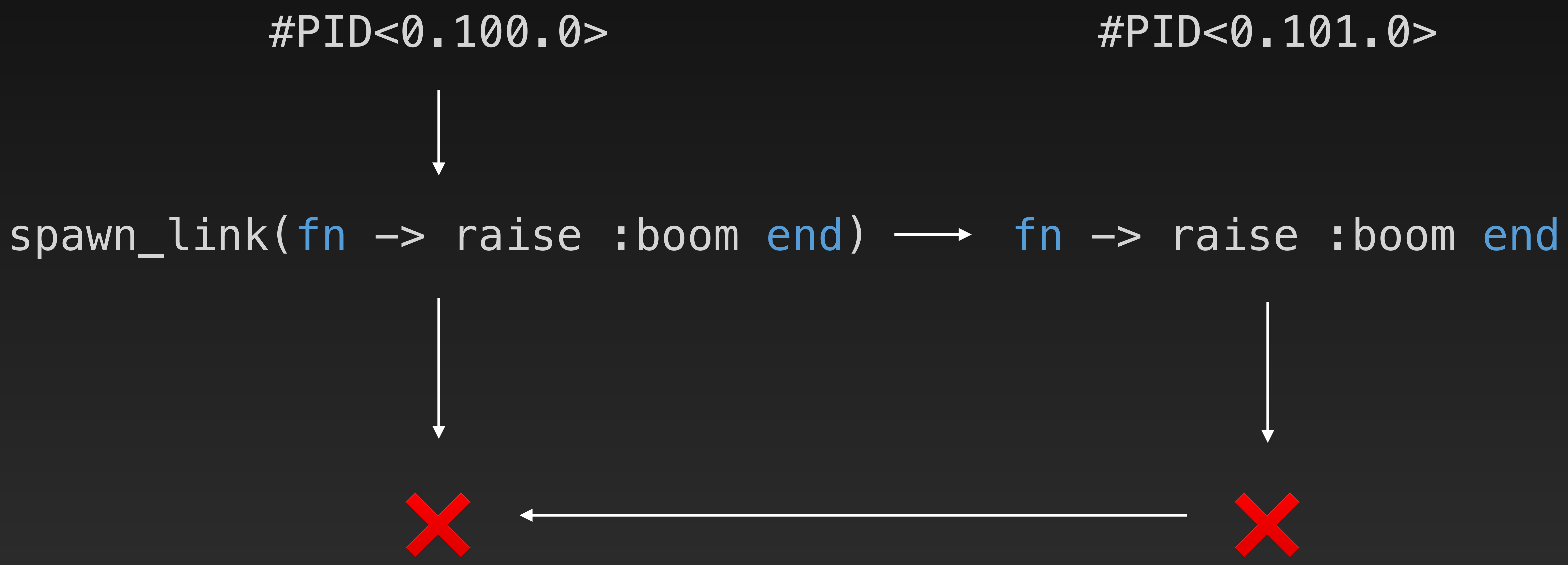
The *next* simplest thing

spawn_link/1

```
spawn_link(fn -> 1 + 1 end)
```

The *next* simplest thing

spawn_link



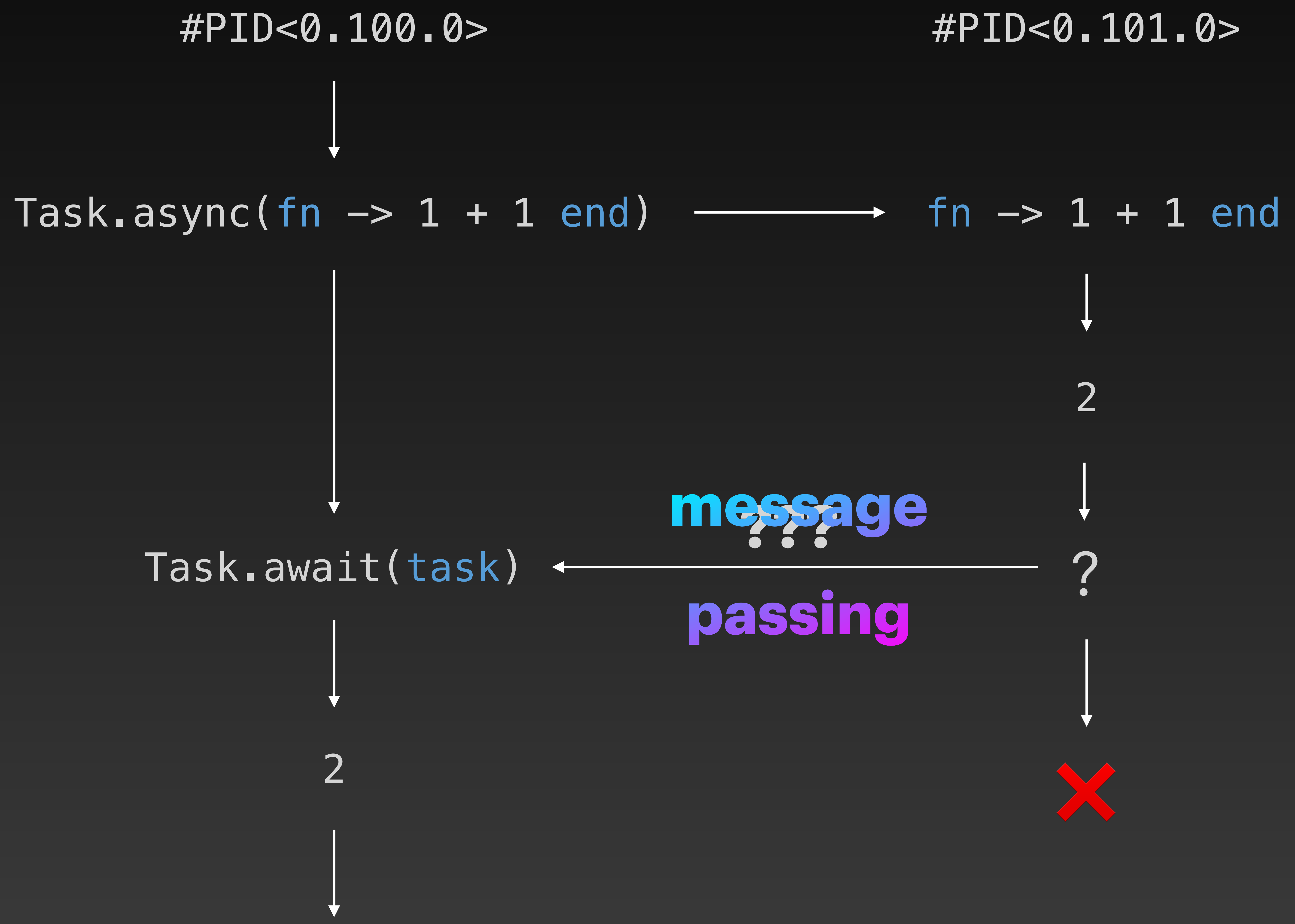
```
#PID<0.100.0>
#PID<0.101.0>
spawn_link(fn -> raise :boom end)
fn -> raise :boom end
```

The *next next* simplest thing?

The Task API

```elixir
task = Task.async(fn -> 1 + 1 end)
Task.await(task)
#=> 2
```

The *next next* simplest thing?



Message Passing

Message Passing

- Unidirectional
- Asynchronous (no delivery guarantee)
- Receiver sees an ordered queue ('mailbox')
- Receiver can receive selectively & at their leisure
- Several addressing options (pid, name, registry)

Sending messages

Kernel.send/2

```
send(#PID<0.101.0>, "Hello")
```

Receiving messages

Kernel.SpecialForms.receive/1

```elixir
receive do
  msg -> IO.puts("Received #{inspect(msg)}")
end
```

What about GenServers?

The GenServer behaviour

- Provides richer messaging primitives (call, cast, & info)
- Useful conventions for state management & lifecycle events
- Not a whole lot of interesting process concerns otherwise

A simple GenServer

```elixir
defmodule HelloWorld do
  use GenServer

  def init(state), do: {:ok, state}

  def handle_call(msg, _from, state) do
    IO.puts "Got call with #{inspect(msg)}"
    {:reply, :ok, state}
  end

  def handle_cast(msg, state) do
    IO.puts "Got cast with #{inspect(msg)}"
    {:noreply, state}
  end

  def handle_info(msg, state) do
    IO.puts "Got info with #{inspect(msg)}"
    {:noreply, state}
  end
end
```

Messaging a GenServer

```elixir
reply = GenServer.call(pid, msg)

GenServer.cast(pid, msg) # No reply

send(pid, msg) # Also no reply
```

GenServer call

$$\begin{tikzcd}
\texttt{caller} \arrow[d] & \texttt{GenServer} \arrow[d] \\
\texttt{GenServer.call(pid, "ping")} \arrow[r] \arrow[d] & \texttt{handle\_call("ping", from, state)} \arrow[d] \\
\texttt{"pong"} \arrow[d] & \texttt{\{:reply, "pong", state\}} \arrow[l] \arrow[d] \\
{} & {}
\end{tikzcd}$$

GenServer innards

Sender-side

```
defmodule GenServer do
  def call(pid, msg, timeout \\ 5000) do
    ref = make_ref()

    send(pid, {:"$gen_call", {ref, self()}, msg})

    receive do
      {^ref, reply} -> reply
    after
      timeout -> raise :timeout
    end
  end
end
```

github.com/erlang/otp/blob/master/lib/stdlib/src/gen_server.erl#L1030

GenServer innards

```elixir
defmodule GenServer do
  def start_link(arg) do
    start_link(__MODULE__, :gen_server_loop, state)
  end

  def gen_server_loop(state) do
    state = receive do
      {:"$gen_call", {ref, caller} = from, msg} ->
        {:reply, response, state} = handle_call(msg, from, state)
        send(caller, {ref, response})
        state

      {:"$gen_cast", msg} ->
        {:noreply, state} = handle_cast(msg, state)
        state

      msg ->
        {:noreply, state} = handle_info(msg, state)
        state

    end

    gen_server_loop(state)
  end
end
```

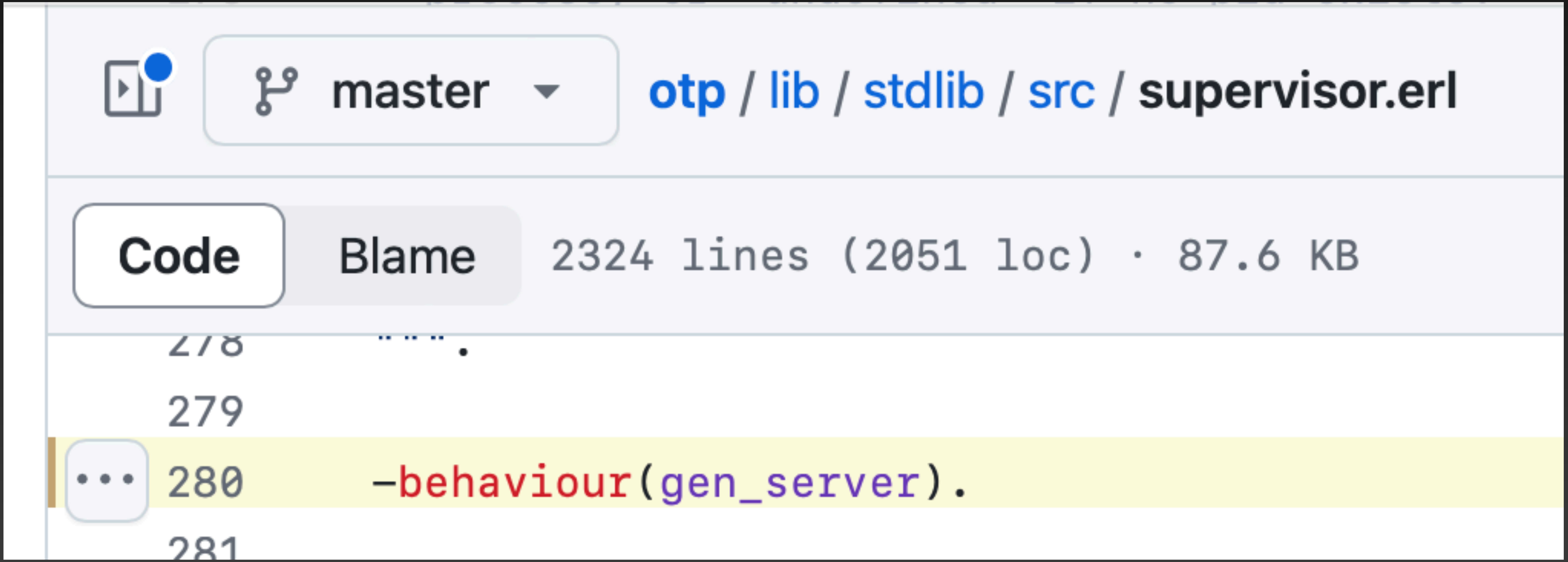
github.com/erlang/otp/blob/master/lib/stdlib/src/gen_server.erl#L2068

GenServers are not special

Just a regular process looping around receive/1

What about Supervisors?

Supervisors are actually GenServers



otp / lib / stdlib / src / supervisor.erl

master

Code | Blame | 2324 lines (2051 loc) · 87.6 KB

```erlang
-behaviour(gen_server).
```

Supervisor is a GenServer

```elixir
defmodule Supervisor do
  @behaviour GenServer

  def start_child(sup_pid, child_spec) do
    GenServer.call(sup_pid, {:start_child, child_spec})
  end

  def handle_call({:start_child, child_spec}, state) do
    {:ok, pid} = start_link(child_spec)
    # Add 'pid' to our set of children in state
    {:reply, {:ok, pid}, updated_state}
  end

  # ...implement other Supervisor functions similarly

  def handle_info({"EXIT", pid, reason}, state) do
    # Look up pid in state, figure out how to restart
    {:noreply, state}
  end
end
```

https://github.com/erlang/otp/blob/master/lib/stdlib/src/supervisor.erl

Supervisors are not special

still just a regular process looping around receive/1

(and also trapping exits)

Some other things

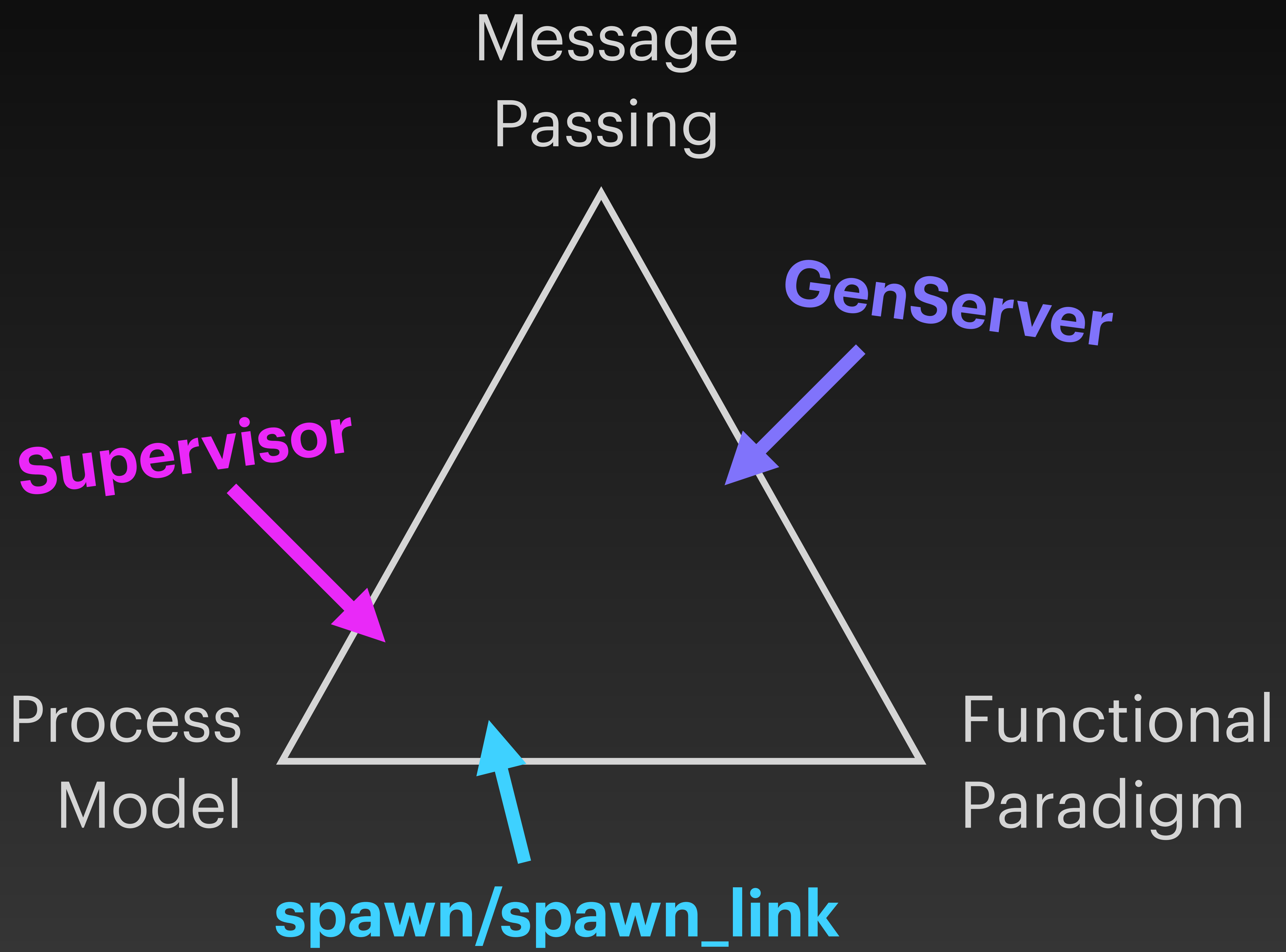
Important details, but still just details

- Process links / monitors
- Trapping exits
- The process dictionary

Read you some Erlang!

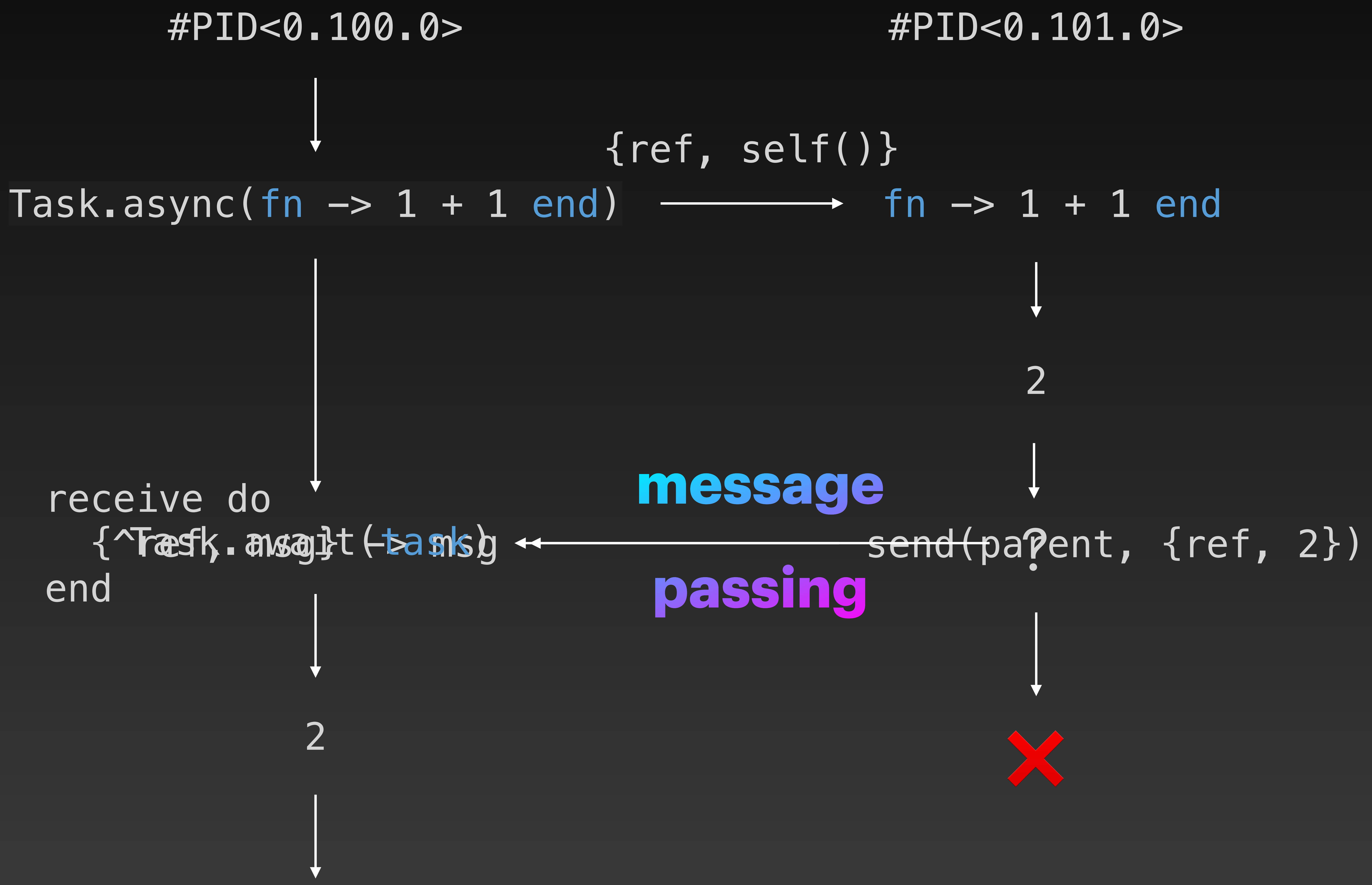
All of Elixir's soul is Erlang's as well

The Soul of Elixir



OTP behaviours are built *on top of* Elixir's soul

Task.async revisited



```
Task.async(fn -> 1 + 1 end)
```

```
fn -> 1 + 1 end
```

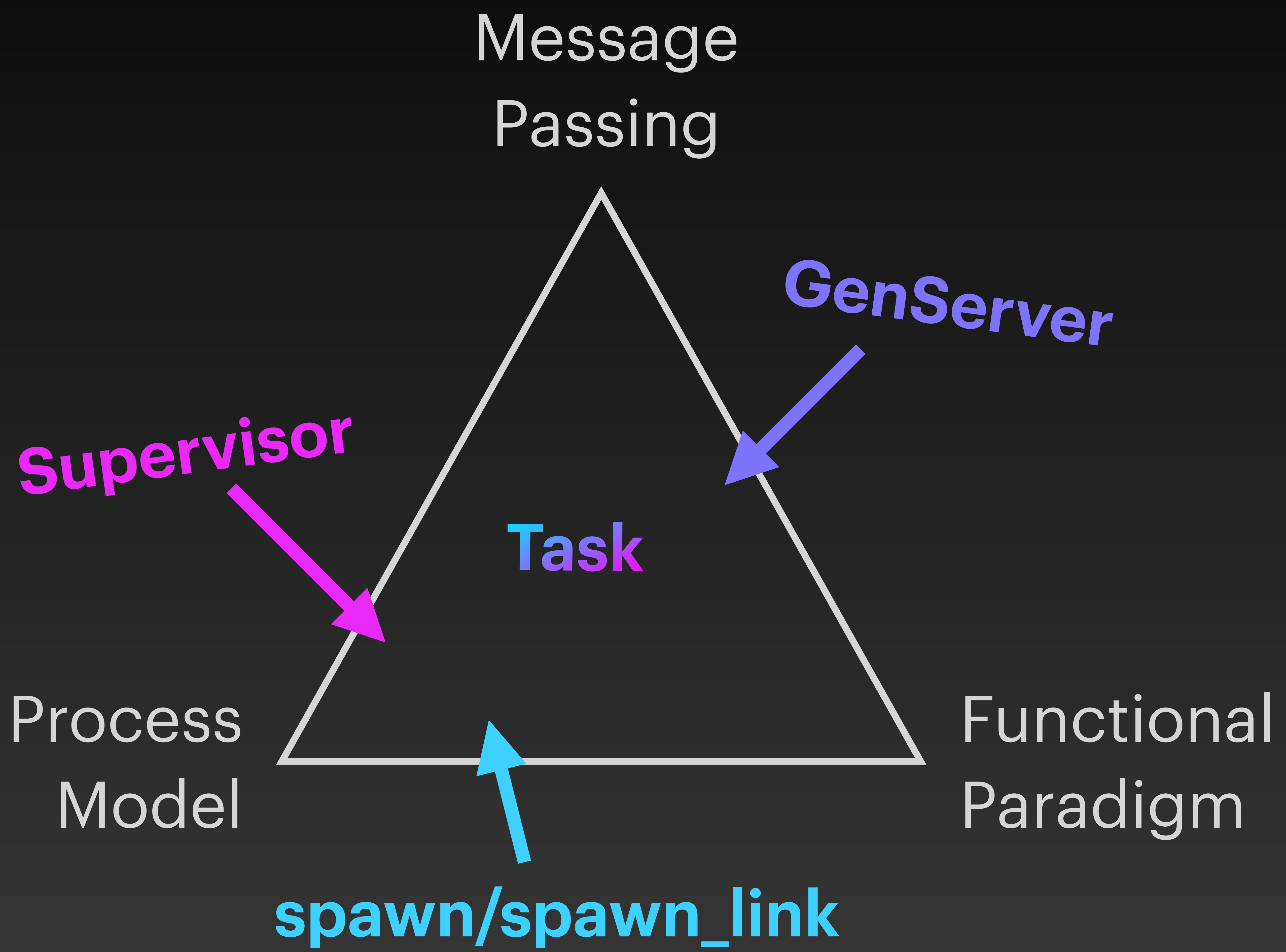
```
receive do
  {^ref, msg} -> msg
end
```

```
Task.await(task)
```

```
send(parent, {ref, 2})
```

```
?
```

Task Embodies The Soul of Elixir



fin