Process Basics

SimAlchemy, Part 1

The Plan

How we will spend our time today

Schedule

8:30AM—10:00AM	Process Basics
10:30AM—12:00PM	GenServer and Friends
1:30PM—3:00PM	Supervisors
3:30PM—5:00PM	OTP Strategy

I Lead a Discussion, Then We Code

For each topic

Stop me with questions!

I am
James Edward Gray II

- I’m a long time programmer
  - My day job is mostly Web development
  - But I play with everything
- I’ve done a lot in the Ruby community
- Lately I’ve been learning Elixir
  - I’m going to share what I’ve learned with you

My Layers of Elixir Learning

Elixir Syntax
OTP (Open “Telecom” Platform)
Erlang
Elixir Macros
Phoenix, Nerves, etc.

Photo from: http://www.trademarksandbrandsonline.com/article/sharing-knowledge-lawfully

Today's (Mostly) About the OTP

The most significant layer for me

What is the OTP?

- Part application development framework
  - gen_server
  - Supervisors
- Part philosophy
  - “Let it crash!”

Other Players

- The BEAM
  - Processes
  - Share nothing architecture
- Elixir’s own twists
  - GenServer
  - Agent
  - Task

Performant, Self-healing Applications

Our goal

Process Basics

Processes: Three Things

- pid = spawn_link(module, fun, args)
  - Make new processes
- send(pid_name_etc, message)
  - Send a message to a process
- receive do message_pattern -> code() end
  - Receive a message

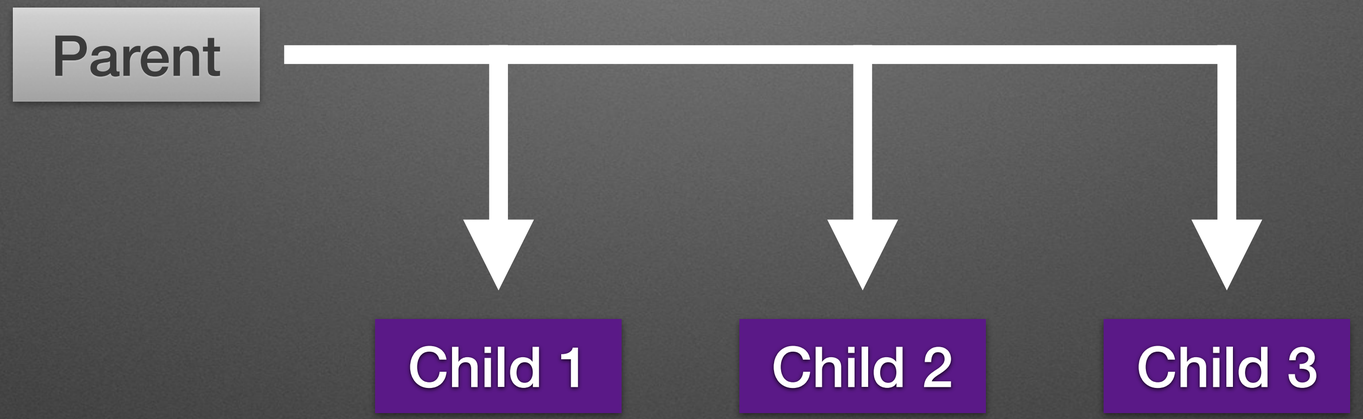
Spawning Processes

- Any Elixir code can spawn_link() processes as needed
- Processes are super lightweight
  - The BEAM laughs at thousands of processes
- Processes are always concurrent and often parallel
- Processes share nothing

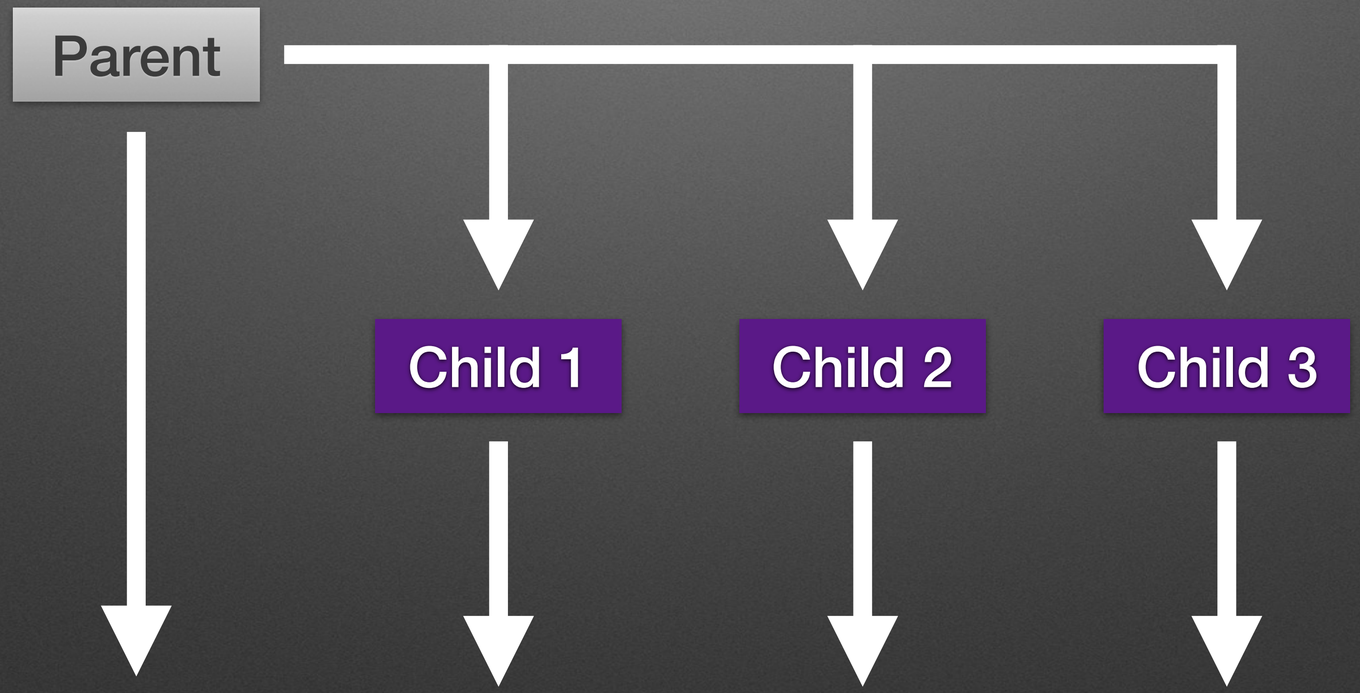
Concurrency and Parallelism

Parent

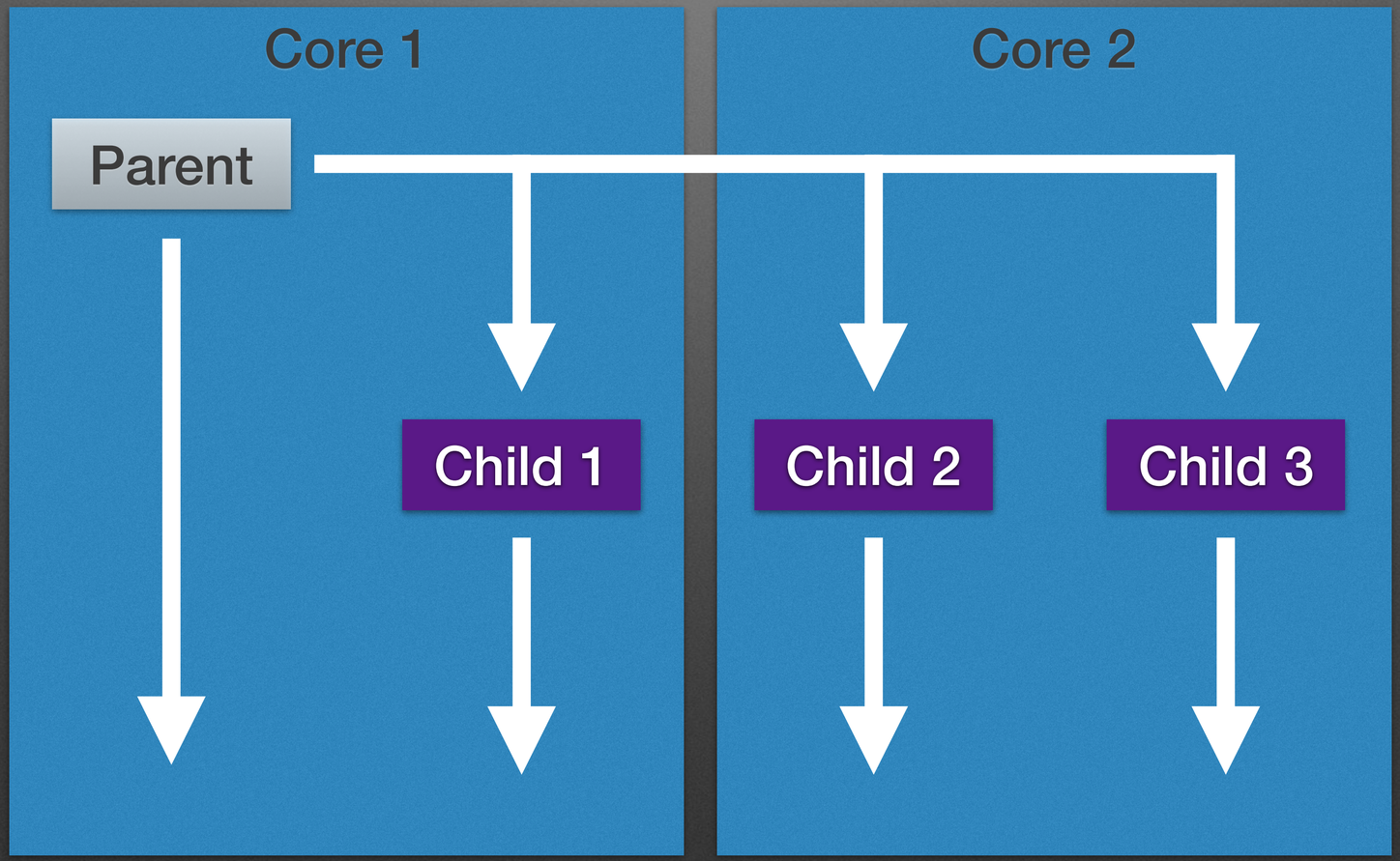
Concurrency and Parallelism



Concurrency and Parallelism



Concurrency and Parallelism



Favor spawn_link()
Over spawn()

We'll discuss the differences later

Sending Messages

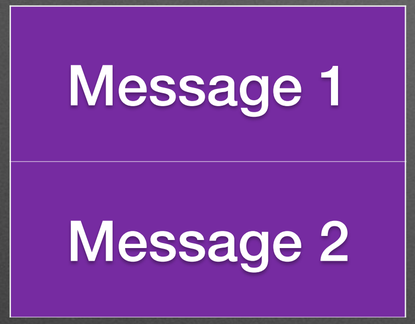
- If a process wants to interact with another process, it needs to send messages
- send() allows any process to send a message to another process it knows of
- Messages are copied to the new process
- Messages are placed in a queue-like mailbox

Receiving Messages

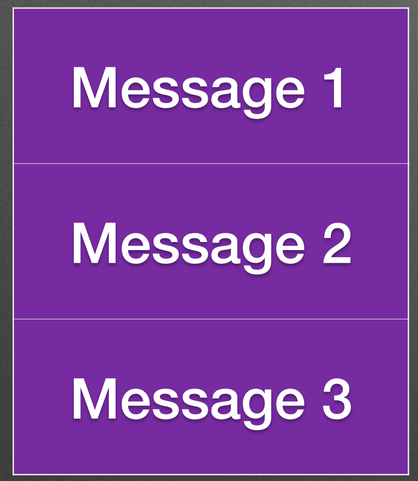
Receiving Messages

Message 1

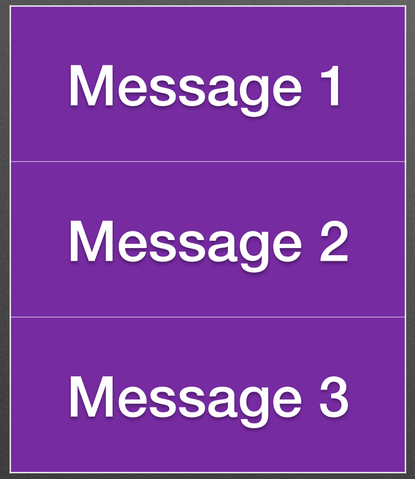
Receiving Messages



Receiving Messages

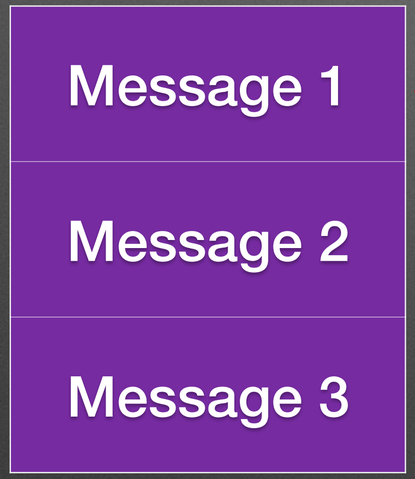


Receiving Messages



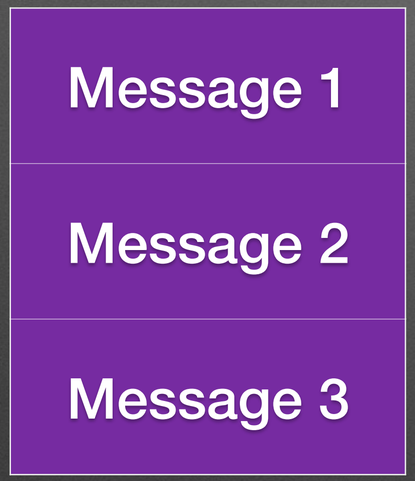

```
receive do
  {:a, payload} -> handle_a(payload)
  {:b, payload} -> handle_b(payload)
  {:c, payload} -> handle_c(payload)
end
```

Receiving Messages



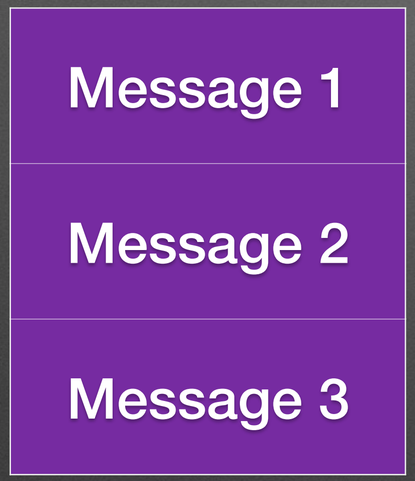

```
receive do
  {:a, payload} -> handle_a(payload)
  {:b, payload} -> handle_b(payload)
  {:c, payload} -> handle_c(payload)
end
```

Receiving Messages



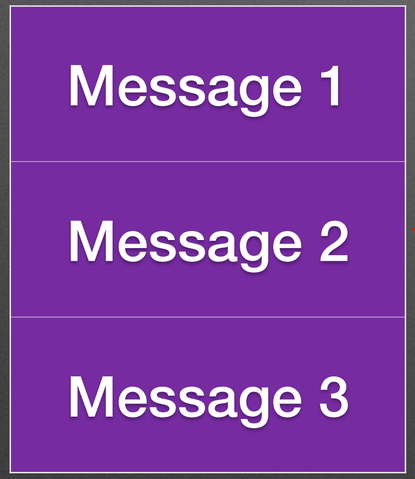

```
receive do
  {:a, payload} -> handle_a(payload)
  {:b, payload} -> handle_b(payload)
  {:c, payload} -> handle_c(payload)
end
```

Receiving Messages




```
receive do
  {:a, payload} -> handle_a(payload)
  {:b, payload} -> handle_b(payload)
  {:c, payload} -> handle_c(payload)
end
```

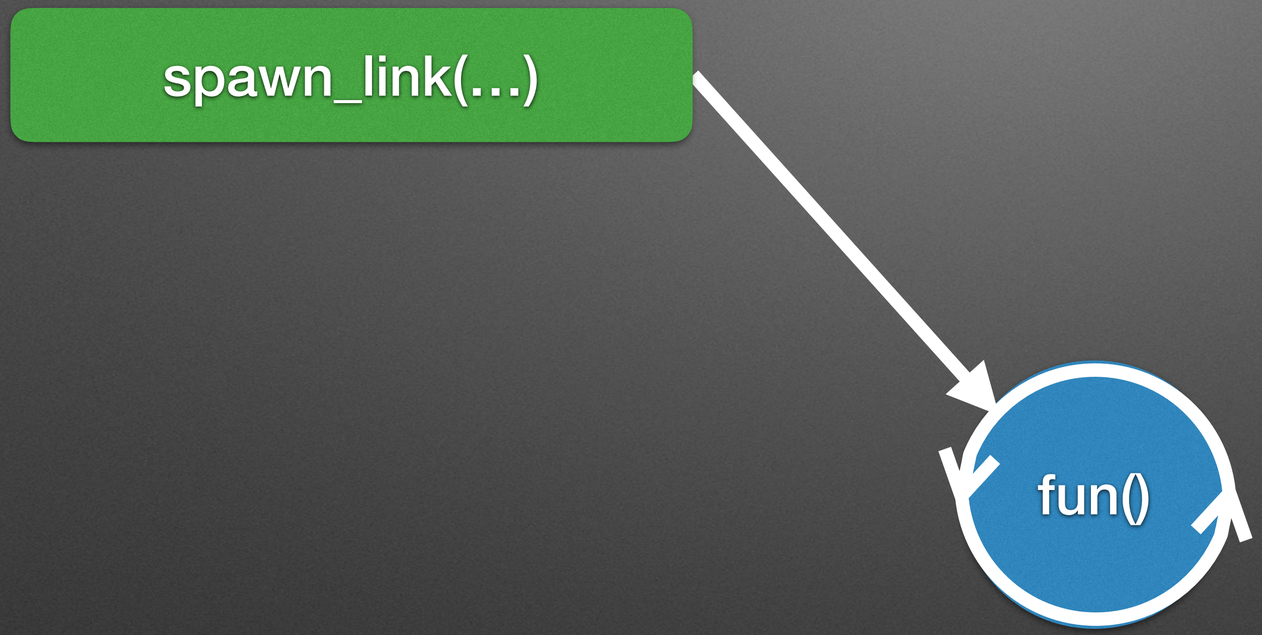
Receiving Messages



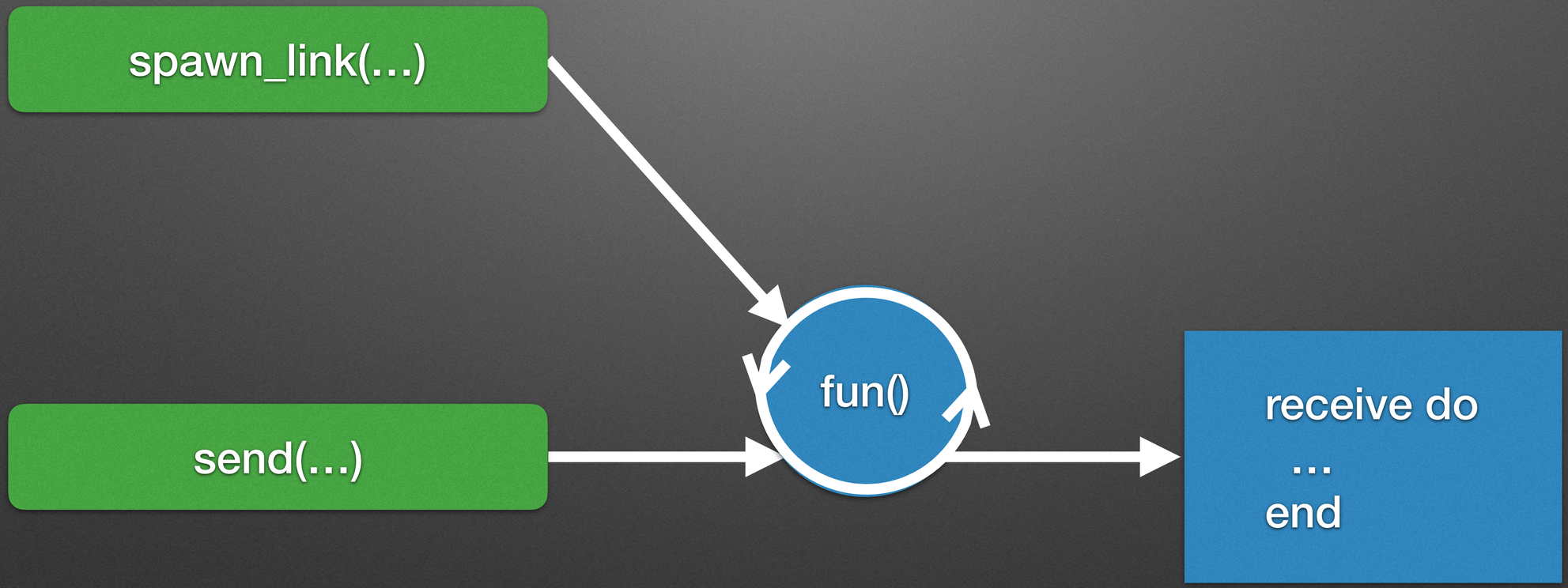

```
receive do
  {:a, payload} -> handle_a(payload)
  {:b, payload} -> handle_b(payload)
  {:c, payload} -> handle_c(payload)
end
```

Process Flow

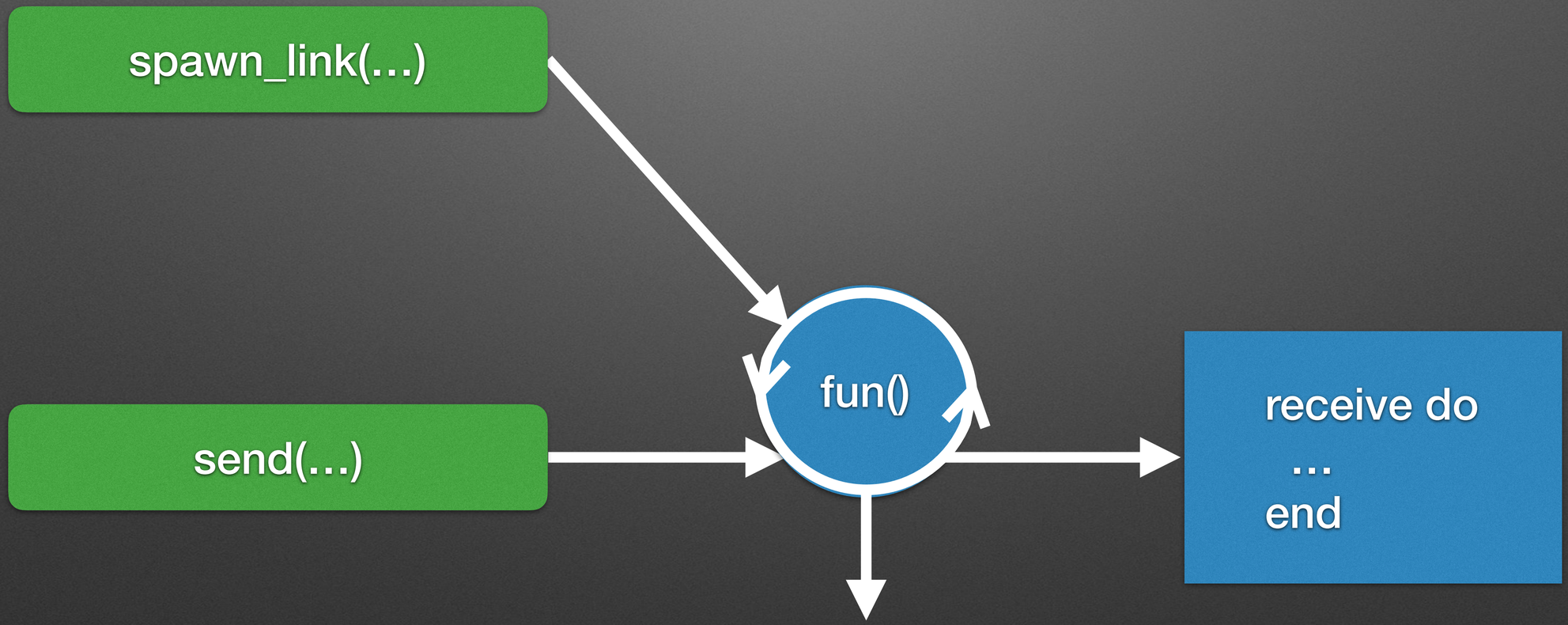
Process Flow



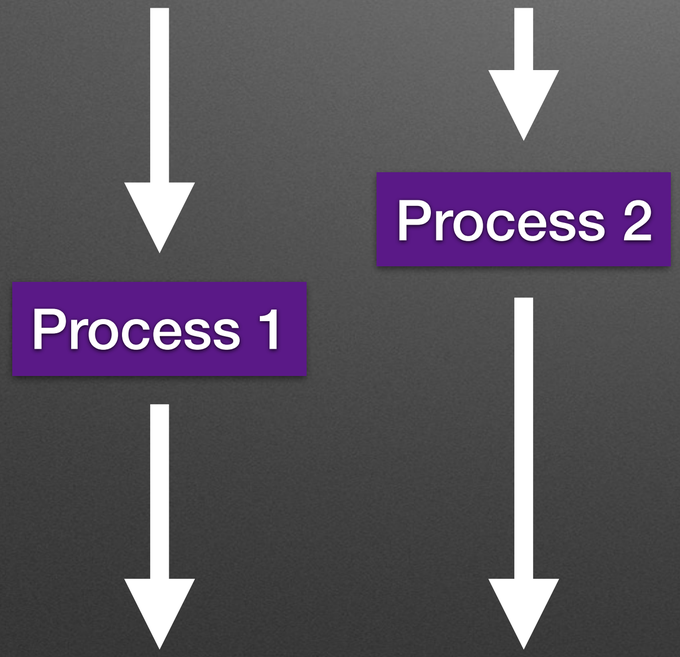
Process Flow



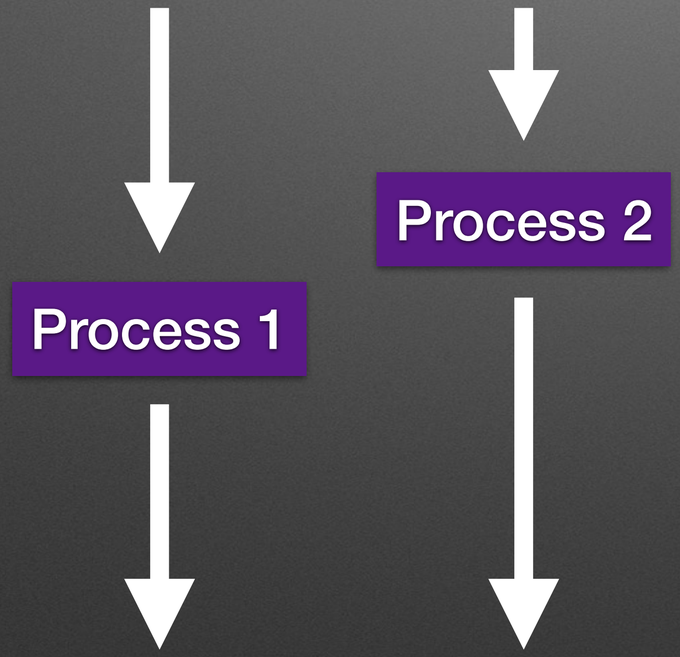
Process Flow

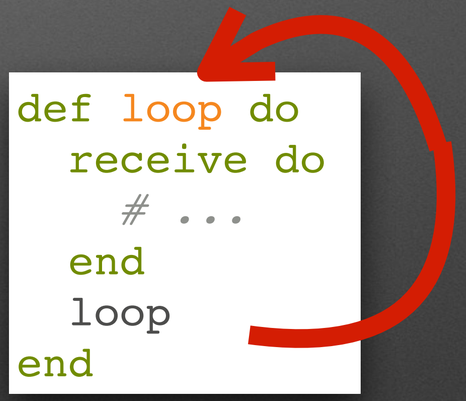


Asynchronous Outside, Synchronous Inside



Asynchronous Outside, Synchronous Inside





```
def loop do
  receive do
    # ...
  end
  loop
end
```

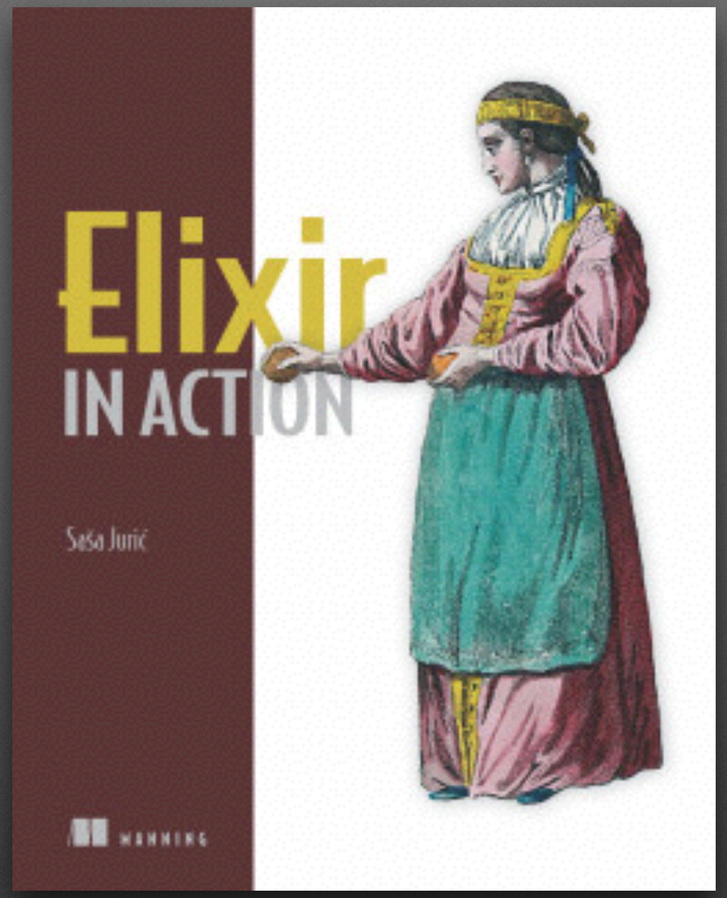
Building on the Basics: Message Replies

```
send(other_process, {:message, self})
receive do
  {:message_reply, reply} ->
    # use reply here...
end
```

Remembering Things

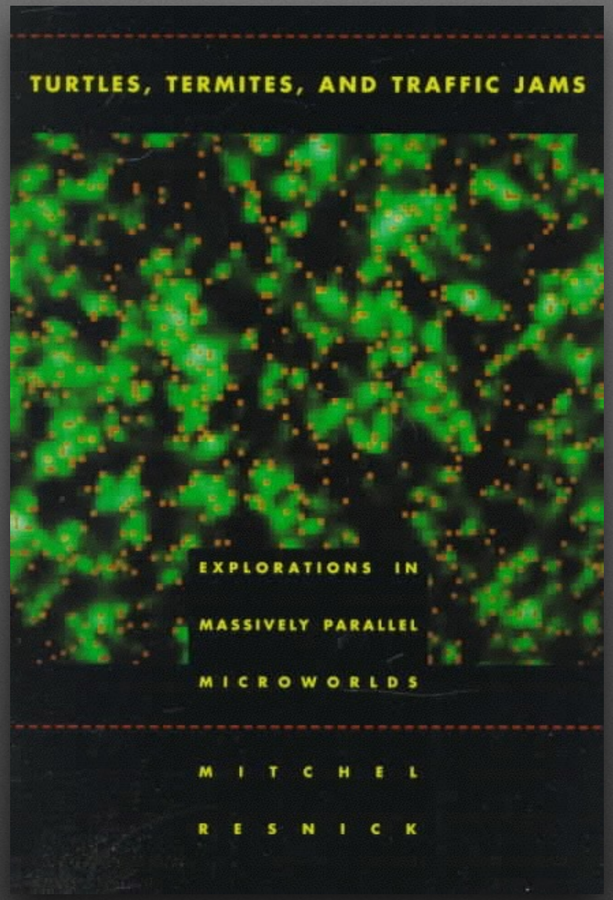
```
def remember(memory) do
  # ...
  remember(memory + 1)  # or whatever
end
```

How do we start learning this stuff?



Elixir in Action

Theory



Turtles, Termites, and Traffic Jams

Practice

Simulations FTW!

We can visualize processes

Forest Fire Simulation

Exercise 1

The Simulation

- A forest is filled with random trees
- All trees on the left edge begin on fire
- A fire burns for four turns, then dies
- While a fire burns all trees in the four cardinal directions catch fire

Building the Processes

- One process tracks the “world”
  - Has the forest data structure
  - Spawns fires as needed
- Each fire (&) is a process
  - Regularly tells the world to advance this specific fire
  - Exits as the fire dies

The Processes

Hints

- The Forest data structure is provided and documented
- You will be writing functions that spawn_link(), send(), or receive
- We wrap spawn_link() and send() so code that uses our process doesn’t need to know our message format

ForestFireSim.Forest

- ForestFireSim.Forest.get_fires(forest)
- ForestFireSim.Forest.get_location(forest, xy)
- ForestFireSim.Forest.reduce_fire(forest, xy)
- ForestFireSim.Forest.spread_fire(forest, xy)
- ForestFireSim.Forest.to_string(forest, ansi? \\ true)

Reading the Tests

```elixir
defmodule FireTest do
  use ExUnit.Case, async: true

  alias ForestFireSim.Fire

  test "advance themselves in the `world` when they receive messages" do
    world = self
    xy = {4, 2}
    intensity = 2
    fire = Fire.ignite(world, xy, intensity)
    send(fire, :advance)
    assert_receive {:advance_fire, ^xy}
  end

  # …
end
```

Let’s “Mob Program”

- We’ll have one driver
  - They translate instructions into code
- We’ll have a queue of navigators
  - I’ll rotate them periodically
  - They tell the driver what to add, but not how
- Everyone else gives the current navigator ideas

http://bit.ly/ffsimzip

See instructions in README.md