

Processes

What are they?

- A “basic unit of concurrency”
 - actor model/job/task/thread
- Runs in the BEAM virtual machine
- Not an OS process
- No shared state
- Lightweight: 1 to 2kb
- Limited to 268 million (theoretical) by VM
- communicates by message passing

Message Passing

Alan Kay - “The notion of object oriented programming is completely misunderstood. It's not about objects and classes, it's all about messages”



B ! {self(),foo}

receive

{From,Msg} ->
Actions

end

How to create one

- 'spawn/1'
 - takes a function with no arguments and “spawns” it
 - returns the “pid”

```
iex> spawn(fn -> IO.puts "pong" end)
```

```
pong
```

```
#PID<0.66.0>
```

Argument Passing

```
iex> job = fn(what) -> spawn(fn -> IO.puts "do: #{what}" end) end
#Function<6.90072148/1 in :erl_eval.expr/5>

iex> job("pong")
do: pong
#PID<0.69.0>
```

send

- 'send(dest, mesg)'
 - send a mesg back to the dest pid

```
iex(> echo = fn(mesg) -> spawn(fn -> send(pid, mesg) end) end
#Function<6.90072148/1 in :erl_eval.expr/5>
iex> echo("Hiyas!")
#PID<0.75.0>
iex> flush
"Hiyas!"
:ok
```

What's wrong with this?

```
iex> echo("zoom")
```

```
#PID<0.78.0>
```

```
iex> flush
```

```
"zoom"
```

```
:ok
```


What's wrong with this?

- different process each time -- called spawn
 - function falls off the end
 - doesn't stay around/not a “daemon” process or persistent actor
 - can't handle different types of messages
- solved by
 - recursion
 - receive

receive

```
receive do
```

```
  pat1 -> "something"
```

```
  pat2 -> "something else"
```

```
  _ -> :fallthrough
```

```
end
```

```
iex(16)> self = self()
```

```
#PID<0.56.0>
```

```
iex(17)> send(self, {:echo, "some job"})
```

```
{:echo, "some job"}
```

```
iex(18)> receive do
```

```
...(18)> {:echo, mesg} -> IO.puts "got: #{mesg}"
```

```
...(18)> _ -> IO.puts "fallthrough"
```

```
...(18)> end
```

```
got: some job
```

To the Files!

```
defmodule Procpresz do
  def pong() do
    receive do
      {:echo, pid, mesg} -> send(pid, mesg)
                          pong()
      {:ping, pid} -> send(pid, "pong")
                     pong()
      _ -> IO.puts "Unauthorized message"
         pong()
    end
  end
end
```

Run it

```
iex(1)> pong_pid = spawn(Procpres, :pong, [])
```

```
#PID<0.85.0>
```

```
iex(2)> self = self()
```

```
#PID<0.83.0>
```

```
iex(3)> send(pong_pid, {:echo, self, "hi"})
```

```
{:echo, #PID<0.83.0>, "hi"}
```

```
iex(4)> flush
```

```
"hi"
```

```
:ok
```

```
iex(5)> send(pong_pid, {:ping, self})
```

```
{:ping, #PID<0.83.0>}
```

```
iex(6)> flush
```

```
"pong"
```

```
:ok
```

```
iex(7)> send(pong_pid, {:foo})
```

```
Unauthorized message
```

```
{:foo}
```

```
iex(8)> flush
```

```
:ok
```

Keeping state

```
defmodule Doubler do
  def loop(num) do
    receive do
      {:inc, pid} -> loop(num + num)
    end
  end
end

def start() do
  spawn(Doubler, :loop, [1])
end
end
```

Doubler in Action

```
iex(1)> p = Doubler.start()  
#PID<0.94.0>  
iex(2)> send(p, { :inc, self() })  
{ :inc, #PID<0.92.0> }  
iex(3)> send(p, { :inc, self() })  
{ :inc, #PID<0.92.0> }  
iex(4)> send(p, { :inc, self() })  
{ :inc, #PID<0.92.0> }  
iex(5)> send(p, { :inc, self() })  
{ :inc, #PID<0.92.0> }
```


Where's my count?

```
def loop(num) do
  receive do
    { :inc, pid } -> loop(num + num)
    { :count, pid } -> send(pid, num)
                    loop(num)
  end
end
```

Count in Action

```
iex(6)> send(p, {:count, self()})
```

```
{:count, #PID<0.92.0>}
```

```
iex(7)> flush
```

```
16
```

```
:ok
```

Keypoints

- Have a start method that kicks off the process and passes the initial state in
- last arguments to spawn is a list of arguments
- during send - always pass in the pid so the process has someone to respond to
- recursion to keep state
- use `_` -> or `garbage` -> as a catchall

Exercises

1. Code pong from memory
2. Create another file with its own spawned process and have it send 3 different types of messages to pong
3. send back "Unauthorized message" to the pid that called it
4. Create another process called HitCounter: everytime 'pong' gets a message, have it send a message to increment the hit counter from a :ping or :echo message, but not any other type of message
5. Have the hit counter send back a count of the current hits pong has received

Addendum

- “I know this stuff already” - Speed code it & no looking at docs or write it on paper, better yet - code it in Erlang
 - Help people who need help
- “I’m lost, I need HELP!” - Experiment, Pair Program, ask questions, use StackOverflow, Cheat
- Answers are in branches at:
 - <https://github.com/MonkeyIsNull/Procprez>