The Actor Model

CSCI 5828: Foundations of Software Engineering
Lecture 13 — 10/04/2016

Goals

- Introduce the Actor Model of Concurrency
  - isolation, message passing, message patterns
- Present examples from our textbook as well as from
  - "Seven Concurrency Models in Seven Weeks" by Paul Butcher

Elixir: Types related to the Actor Model

- Elixir provides a wide range of types
  - Value Types: integers, floats, atoms (like symbols in Ruby; keywords in Clojure); ranges (5..15), regular expressions and strings (aka binaries)
  - Boolean values: true, false, nil
    - In boolean contexts, only false and nil evaluate to false; everything else evaluates to true
- But ***system types*** are related to the Actor model:
    - **pids**: a “process id”; not a Unix process, an Elixir process
      - the function `self` will return the pid of the current process
    - **refs**: a globally unique id

Collection Types

- Elixir has the following collection types
  - **Tuples**: an ordered collection of values
    - { 1, :ok, “hello” } — you can use tuples in pattern matching
      - We will use tuples to pass messages between actors
  - **Lists** — a linked data structure with a head and a tail
    - the head contains a value; the tail is another list; a list can be empty
  - **Maps** — a collection of key-value pairs
    - %{ key => value, key => value }

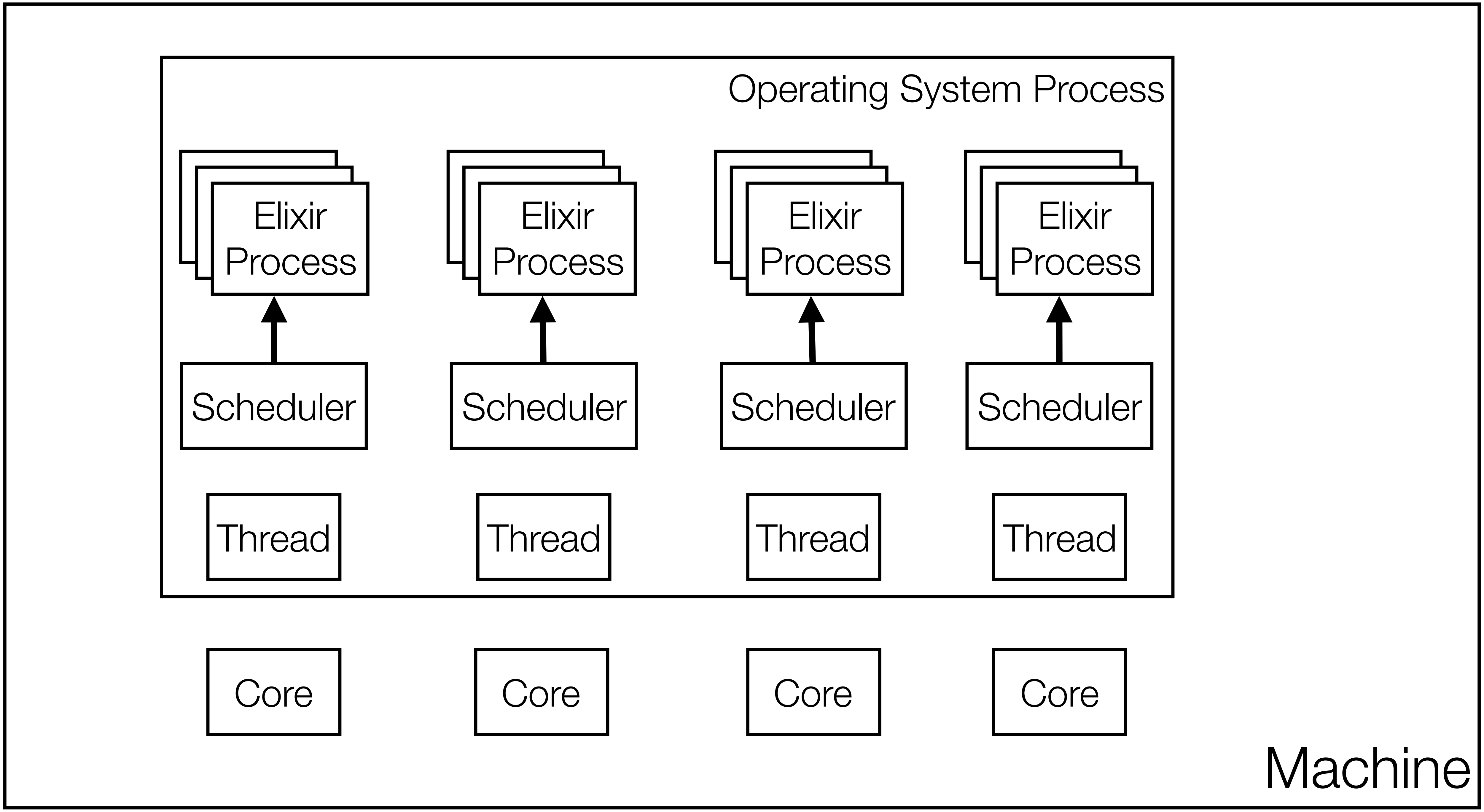
Actors

- Elixir makes use of a novel approach to concurrency, pioneered by Erlang, called the Actor model
    - In this model, actors are independent entities that run in parallel
    - Actors **encapsulate state that can change over time**
        - but that state **is not shared** with any other actor
        - As a result, there can be no race conditions
    - Actors **communicate by sending messages** to one another
        - An actor will process its messages ***sequentially***
    - Concurrency happens because many actors can run in parallel
        - but each actor is itself a sequential program
            - an abstraction with which developers are comfortable

Processes

- Actors are also called “processes”
    - In most programming languages/operating systems
        - processes are ***heavyweight*** entities
    - In Elixir, a process is very ***lightweight*** in terms of resource consumption and start-up costs; lighter weight even than threads
- Elixir programs might launch ***thousands of processes all running concurrently***
    - and without the programmer having to create thread pools or manage concurrency explicitly (the Erlang virtual machine does that for you)
- Instead, Elixir programs make sure the right processes get started and then work is performed by passing messages to/between them

Actor Architecture in Elixir



Messages and Mailboxes

- **Messages** in Elixir are ***asynchronous***
  - When you send a message to an actor, the message is placed instantly (actually ***copied***) in the actor's mailbox; the calling actor **does not block**
- **Mailboxes** in Elixir are ***queues***
  - Actors perform work in response to messages
  - When an actor is ready, it pulls a message from its mailbox
    - and responds to it, *possibly sending other messages in response*
  - It then processes the next message, until the mailbox is empty
    - at that point, it blocks waiting for a new message to arrive

Actor Creation: spawn and spawn_link

- An actor is created by using the `spawn` function or the `spawn_link` function
  - We will discuss `spawn_link` later in this lecture
- `spawn` takes a function and returns a “process identifier”, aka a `pid`
  - The function passed to `spawn` takes no arguments and
    - its structure is expected to be an infinite loop
    - at the start of the loop, a `receive` statement is specified
      - this causes the actor to block until a message arrives in its mailbox
    - The body of the receive statement specifies the messages that the actor responds to
      - once a message is handled, the actor loops, executing the receive statement again, thus blocking until the next message arrives

Simple Example (1)

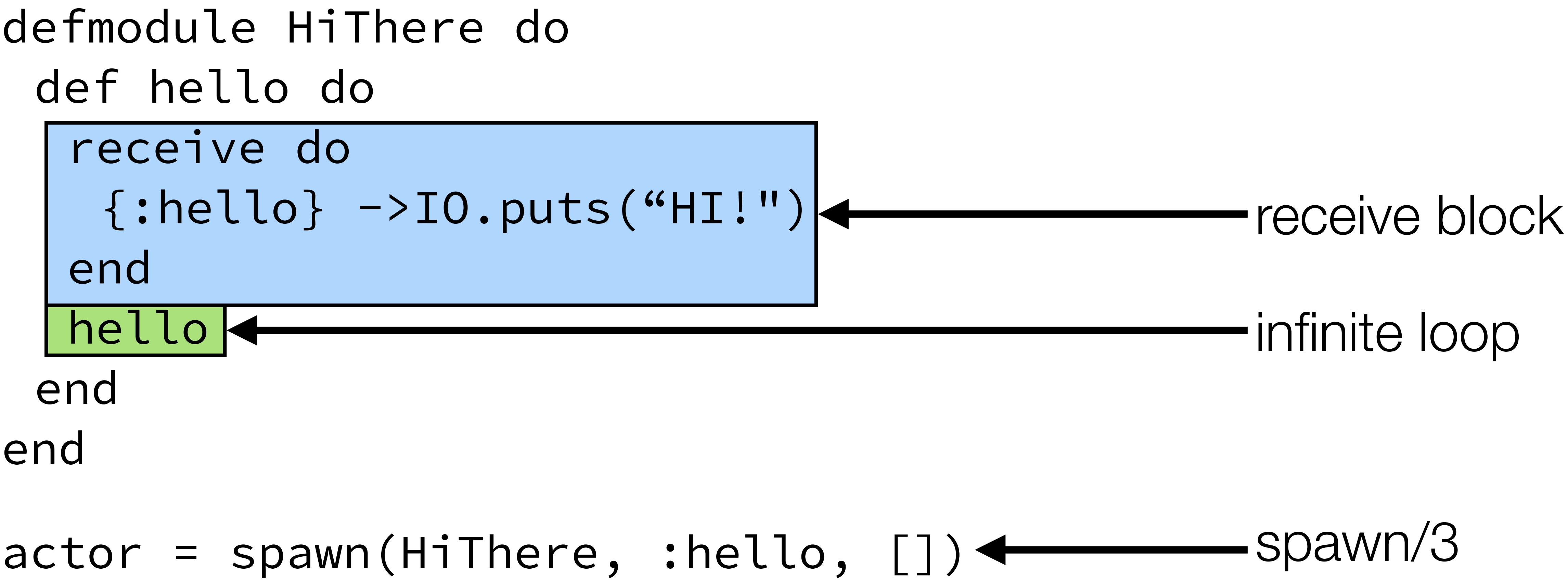
```
one_message = fn () ->
 receive do
  {:hello} -> IO.puts(“HI!”)
 end
end
actor = spawn(one_message)
send(actor, {:hello})
```

DEMO: simple1.exs and simple2.exs

- This example creates an actor that can only respond to a single message. That message MUST be the tuple `{:hello}`. Any other message is ignored
    - When the message `{:hello}` arrives, the actor prints out “HI!” and then the function of the actor returns. That is interpreted as a “normal” exit, similar to having the `run()` method of a Java thread return.
        - Note: you can still send messages to the pid that was returned, those messages are simply ignored

Simple Example (2)

- To create a version of our actor that stays alive and can always respond to {:hello} messages, we need to use a named function inside of a module



```
defmodule HiThere do
 def hello do
  receive do
   {:hello} ->IO.puts(“HI!")
  end
  hello
 end
end

actor = spawn(HiThere, :hello, [])
send(actor, {:hello}) => “HI!”
send(actor, {:hello}) => “HI!”
…
```

DEMO: simple3.exs

© Kenneth M. Anderson, 2016

Lots of Processes

- We mentioned that Elixir processes are lightweight
  - What does that mean in practice?
  - It means you can create LOTS of Elixir processes and it will NOT tax your machine; for instance, on my machine, this code creates 10,000 Elixir processes in 0.4 seconds!

DEMO: lots.exs

```
defmodule Lots do
  def loop do
    receive do
      {:hello} -> "HI!"
    end
    loop
  end
end
pids = Enum.map(1..10_000, &(spawn(Lots, :loop, [])))
```

© Kenneth M. Anderson, 2016

More Advanced Example *(pg. 191-192 of textbook)*

```
defmodule Chain do
  def counter(next_pid) do
    receive do
      n ->
        send next_pid, n + 1
    end
  end

  def create_processes(n) do
    last = Enum.reduce 1..n, self,
             fn (_,send_to) ->
               spawn(Chain, :counter, [send_to])
             end

    send last, 0     # start the count by sending a zero to the last process

    receive do       # and wait for the result to come back to us
      final_answer when is_integer(final_answer) ->
        "Result is #{inspect(final_answer)}"
    end
  end

  def run(n) do
    IO.puts inspect :timer.tc(Chain, :create_processes, [n])
  end
end
```

DEMO: chain.exs

More Advanced Message Passing

```
defmodule Talker do
  def loop do
    receive do
      {:greet, name} -> IO.puts("Hello #{name}")
      {:praise, name} -> IO.puts("#{name}, you're amazing!")
      {:celebrate, name, age} -> IO.puts(“HB #{name}. #{age} years old!")
    end
    loop
  end
end
```

DEMO: talker.exs

```
pid = spawn(Talker, :loop, [])
send(pid, {:greet, "Ken"})
send(pid, {:praise, "Lilja"})
send(pid, {:celebrate, "Miles", 42})
:timer.sleep(1000) # allow responses to be generated
```

Discussion (I)

- The actor specifies what messages it can process with `receive`
  - Each message uses pattern matching specifying a literal atom (:praise) and a variable that then matched whatever was sent with the rest of the message
    - {:praise, name} matches all 2-tuples that start with the :praise atom and then binds name to the second value
      - that binding can then be used in the message handler
        - `IO.puts("#{name}, you're amazing!”)`
  - The call to `receive` blocks the actor until there is a message to process
- The actor defines a single function: `loop`; `loop` is seemingly implemented as an **infinite recursive loop** because it calls `loop` after it calls `receive`
  - however, tail call elimination implements this with a goto
    - it’s a loop **not** a recursive call

Discussion (II)

- The rest of the code is used to create the actor and send messages to it
  - since the message sends are asynchronous, this code ends with a call to :timer.sleep (actually an Erlang function) to allow time for the messages to be received
- The call to `spawn`, returns a process id that allows us to send messages to the actor with the function `send`. `send` takes a pid and a tuple, adds the tuple to the actor’s mailbox and returns immediately

Linking Actors

- We can establish better interactions with our actors if we link them
  - Linked actors get **notified** if one of them ***goes down***
    - by either exiting normally or crashing
  - To receive this notification, we have to tell the system to "trap the exit" of an actor; it then sends us a message in the form: {:EXIT, pid, reason} when an actor goes down but ONLY if we start the process using `spawn_link`
- We can modify our previous example to more cleanly shutdown by implementing another message
  - {:shutdown} -> exit(:normal)
- We then call Process.flag(:trap_exit, true) in our main program, change it to send the shutdown message, and then wait for the system generated notification that the Talker actor shutdown.

DEMO: talker2.exs

Maintaining State

- To maintain state in an actor, we can use pattern matching and recursion
    - defmodule Counter do
        - def loop(count) do
            - receive do
                - {:next} ->
                    - IO.puts(“Current count: #{count}”)
                    - loop(count + 1)
            - end
        - end
    - end
- counter = spawn(Counter, :loop, [1])
- send(counter, {:next}) => Current count: 1
- send(counter, {:next}) => Current count: 2

DEMO: counter1.exs

Hiding Messages

- You can add functions to your actor to hide the message passing from the calling code

- `def start(count) do`
  - `spawn(Counter, :loop, [count])`
- `end`
- `def next(counter) do`
  - `send(counter, {:next})`
- `end`

- These functions can then be called instead

  - `counter = Counter.start(23)`
  - `Counter.next(counter) => Current count: 23`
  - `Counter.next(counter) => Current count: 24`

DEMO: counter2.exs

Bidirectional Communication

- While asynchronous messages are nice
  - there are times when we will want to ask an actor to do something and then wait for a reply from that actor to receive a value or confirmation that the work has been performed
- To do that, the calling actor (or main program) needs to
  - generate a unique reference
  - call `send` with a message that includes its pid (obtained via `self`)
  - wait for a message that includes its ref and includes the response value
- Let's look at a modified version of count that returns the actual count rather than print it out

Receiving the Message in the Actor

- We update our actor to expect the pid of the caller and the unique ref

```
• def loop(count) do
    • receive do
        • {:next, sender, ref} ->
            • send(sender, {:ok, ref, count})
            • loop(count + 1)
    • end
• end
```

- We now expect our incoming message to contain the sender's pid and a unique ref. The :next atom still provides a unique "name" for the message
    - We send the current count back to the caller and pass back its ref too

Receiving the return value in the Caller

- The caller’s code has to change as well

- `def next(counter) do`
  - `ref = make_ref()`
  - `send(counter, {:next, self, ref})`
  - `receive do`
    - `{:ok, ^ref, count} -> count`
  - `end`
- `end`

- In this function, we call make_ref() to get a unique reference. We then send the :next message to the actor. We then block on a call to receive, waiting for the response.
  - The response’s ref must match the previous value of ref (i.e. ^ref) and then binds the return value to the count variable which is then returned

DEMO: counter3.exs

Naming Actors

- You can associate names (atoms) with process ids, so you can refer to an actor symbolically
  - `Process.register(pid, :counter)`
    - this call takes a pid returned by `spawn` or `spawn_link` and associates it with the `:counter` atom
  - Now, when sending messages to that actor, you can use the atom
    - `send(:counter, {:next, self, ref})`

DEMO: counter4.exs

Reminder: Actors run in Parallel

- Here's a different implementation of Parallel.map
    - `defmodule Parallel do`
        - `def map(collection, fun) do`
            - `parent = self()`
            - `processes = Enum.map(collection, fn(e) ->`
                - `spawn_link(fn()  ->`
                    - `send(parent, {self(), fun.(e)})`
                - `end)`
            - `end)`
            - `Enum.map(processes, fn(pid) ->`
                - `receive do`
                    - `{^pid, result} -> result`
                - `end`
            - `end)`
        - `end`
    - `end`

Parallel.map in action

Take a PID of the calling process, a collection, and a function

```
parent = self()          [1, 2, 3, 4]          add_one = fn(x) -> x + 1 end;
```

Transform it into a collection of pids of actors

```
[#PID<0.57.0>, #PID<0.58.0>, #PID<0.59.0>, #PID<0.60.0>]
```

where each actor is set-up to take the original value, pass it to the function, and return it back to the calling process

send(parent, {self(), fun.(e)})

```
send(parent, {#PID<0.57.0>, add_one.(1)})
```

After the parent launches these processes, it then uses Enum.map to wait for the messages from each process

Using Parallel

- `slow_double = fn(x) -> :timer.sleep(1000); x * 2 end`
- `:timer.tc(fn() -> Enum.map([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], slow_double) end)`
- `:timer.tc(fn() -> Parallel.map([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], slow_double) end)`

- On my machine, the first call to :timer.tc returned
    - `{10010165, [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]} <= about 10 seconds`
- The second call returned
    - `{1001096, [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]} <= about 1 second`
- One process got launched per element of the input collection
    - they all waited one second, and then returned their result.
- In the first call to :timer.tc, the delay of one second occurred ten times sequentially; and so the entire call to Enum.map took 10 seconds

DEMO: parallel.exs

Summary

- We have had a brief introduction to the Actor model
    - multiple actors run in parallel
        - each has its own mailbox and processes messages sequentially
    - to perform work, actors send asynchronous messages to each other
        - if we need actors to wait for a response
            - we can do that with refs and calls to receive