CPS2001
Programming Paradigms Assignment 4
Concurrent Paradigm

Mark Galea
mgale11@um.edu.mt

5th December 2013

1 Preliminaries

This is the last assignment for CPS2001; It carries 10% of the final mark and hence will be scored out of 10 points as follows

- **2 points** for Task A
- **8 points** for Task B

The deadline for submission of the complete assignment is at **NOON** on **2nd of January 2017** and it must be carried out **individually**.

You are reminded that we take plagiarism seriously. While you are encouraged to discuss ideas with fellow students and perform additional research you cannot copy and steal ideas. Read and follow the information found at `https://www.um.edu.mt/ict/Plagiarism` carefully. Extreme measures might be taken in case of plagiarism. ***Note that you might be randomly selected for a 10 minute interview to discuss your implementation***.

2 Tasks

2.1 Task A - Cyclic Function

The system consists of two actors; `A` and `B`. Actor `A` produces an infinite stream of numbers which Actor `B` consumes. Actor `A` outputs the sequence:

$$x, f(x), g(f(x)), f(g(f(x))), ... \tag{1}$$

where

$$f(x) = \begin{cases} 1000 & \text{if } x = 0 \\ 2x & \text{otherwise} \end{cases} \tag{2}$$

and

$$g(x) = \frac{x}{3} \tag{3}$$

Actor `B` deals with numbers three at a time. It prints each triple tabulated on the same line followed by their integer average.

The first two lines of your execution should look as follows:

Line	x	$f(x)$	$g(f(x))$	*Average*
1	0	1000	333	444

Line	$f(g(f(x)))$	$g(f(g(...)))$	$f(g(f(g(...))))$	*Average*
2	666	222	444	444

Use the following Actor definition to solve this problem and place your solution in a file named `Exercise1.scala`. Write the `main` function in the same file to start off the communication.

```
class ActorA(processB: ActorRef) extends Actor with ActorLogging {
    def receive = ???
}

class ActorB extends Actor with ActorLogging {
    def receive = ???
}
```

2.2 Task B - Logic Circuits

In this section you are required to create a circuit simulator. The circuits simulated by our actor system are composed of *wires* and *components*; both of which are actors. A wire transports a signal, which can be false (low voltage) or true (high voltage). A component (e.g. And Gate) receives a message through its *input wires* and outputs the resultant signal(s) using its *output wire(s)*. Changing the values of input wires does not immediately alter the value of the output wires; each component defines a delay that is specific to it.

The trait `SimConfig` defines the delays which you are required to use; **do not** add **delays** which are not present in the `SimConfig` trait.

```
trait SimConfig {
  val inverterDelay = 100 milliseconds
  val andDelay = 100 milliseconds
  val orDelay = 1000 milliseconds
}
```

We will start by creating a `Wire` actor. The `Wire` actor allows `component` actors to subscribe to state changes by using the `AddComponent` message. This message is composed of the `wireName` and the `ActorRef` of the subscribing actor. A `component` actor may subscribe to more than one `Wire` and hence will receive multiple `StateChange`

messages. The `wireName` used to subscribe to the `Wire` will be given back to the original subscriber on each `StateChange` message. *Note that different components might identify the same* ***`Wire`*** *using a different* ***`wireName`****.*

```
case class AddComponent(wireName: String, actor: ActorRef)
case class StateChange(wireName: String, state: Boolean)
class Wire(var currentState: Boolean) extends Actor with ActorLogging {
  var associations = Map[ActorRef, String]()

  def receive: Actor.Receive = {
    case AddComponent(name: String, b: ActorRef) => ???
    case current: Boolean => ???
  }

}
```

Write a `Probe` Actor which reacts to wire updates. Use the following Actor definition:

```
class Probe(name: String, input0: ActorRef) extends Actor with ActorLogging {
    implicit val ec = context.dispatcher
    def receive = ???
}
```

Next, you will write an Inverter, Or, and And Gate using the following Actor definition:

```
class Inverter(input0: ActorRef,
           output0: ActorRef) extends Actor with ActorLogging with SimConfig {
  implicit val ec = context.dispatcher
  def receive = ???
}

class And(input0: ActorRef, input1: ActorRef,
           output0: ActorRef) extends Actor with ActorLogging with SimConfig {
  implicit val ec = context.dispatcher
  def receive = ???
}

class Or(input0: ActorRef, input1: ActorRef,
           output0: ActorRef) extends Actor with ActorLogging with SimConfig {
  implicit val ec = context.dispatcher
  def receive = ???

}
```

Write an alternative implementation to the `Or` gate implementation by making use of the previously defined `Inverter` and `And` Gate actors.

```
class OrAlt(input0: ActorRef, input1: ActorRef,
              output0: ActorRef) extends Actor with ActorLogging {
```

```
    def receive: Actor.Receive = Actor.emptyBehavior
}
```

Now, let's have some fun creating more complex *components*. Create a `HalfAdder` and a `FullAdder` by making use of the previously defined gates. Use the following definitions:

```
class HalfAdder(input0: ActorRef, input1: ActorRef,
                outputSum: ActorRef,
                outputCarry: ActorRef) extends Actor with ActorLogging {
  def receive: Actor.Receive = Actor.emptyBehavior
}

class FullAdder(input0: ActorRef, input1: ActorRef, inputCarry: ActorRef,
                outputSum: ActorRef,
                outputCarry: ActorRef) extends Actor with ActorLogging {
  def receive: Actor.Receive = Actor.emptyBehavior
}
```

Finally, write a `Demux` actor which realises a demultiplexer with n control wires (and 2^n output wires). The demultiplexer directs the signal from an input wire based on the control signal. The rest of the output signals are set to 0. The Demux actor is defined as follows:

```
class Demux(input: ActorRef, controls: List[ActorRef],
            outputs: List[ActorRef]) extends Actor with ActorLogging {
    def receive: Actor.Receive = Actor.emptyBehavior
}
```

In order to implement the `Demux` component, we will first implement a simpler `Demux2` actor which has one control signal and two output signals. The `Demux2` actor is defined as follows:

```
class Demux2(input: ActorRef, control: ActorRef,
             output1: ActorRef,
             output0: ActorRef) extends Actor with ActorLogging {
    def receive: Actor.Receive = Actor.emptyBehavior
}
```

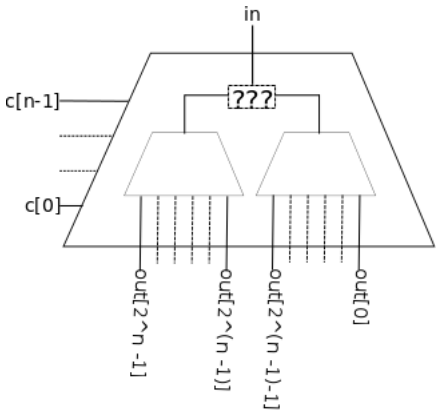
Refer to the image below for some intuition on how the `Demux` actor can be defined ***recursively*** using the `Demux2` actor. *Note that non-recursive implementation will be considered invalid.*

The specified *circuit simulator* should be implemented in a file called `Exercise2.scala`.

3 Report

This assignment is to be accompanied by a report. For each question include:

1. A diagram of the actor system



2. A message flow excerpt for some input combinations
3. Any assumptions and/or observations made

For example for an `And` Gate, the actor system diagram should show four actors: two input wires, one output wire and the and gate actor. The message flow section should illustrate clearly how messages are sent and received by actors for a subset of the 4 input combinations. Any observations or assumptions made while implementing and executing the program should be stated in the report.

4 Submissions

All submissions have to take place by the stipulated deadline.

1. Package all the source code together with the report and submit through VLE.
2. Make sure that the files and functions are named exactly as specified in this document.

5 Conclusion

If you have reasonable questions please contact me on my email address. Obviously I will not provide implementation details however if you require some clarification feel free to contact me. Dedicate enough time for completing this assignment since no **extensions** will be provided. Good luck and happy holidays!