Dining Philosophers

Aditya Gupta

February 2023

Introduction

In this assignment, we solve the classic dining philosophers problem and use this to learn to write concurrent programs in Elixir.

Problem Description

In this problem, we have 5 philosophers sitting at a round table with food in front of them and 5 chopsticks on the table between each other. Each philosopher switches between dreaming and an eating state. The problem begins when they try to grab the chopsticks to eat some food. The chopsticks may or may not be available to a philosopher depending on if another philosopher has already taken one of the chopsticks already.

We can also end up in a deadlock scenario where every philosopher ends up with just one chopstick and then none of them can eat as they are all just stuck waiting for one of the other philosophers to drop the chopsticks so that they can take them but no one does. Our task is to program the behaviour of the philosophers so that they don't end up in a deadlock and manage to get something to eat. We have to represent every chopstick and philosopher as their own process as they can all interact with each other simultaneously. My approach towards solving this involves handling how the philosophers grab and release the chopsticks and handling any deadlocks we face along the way.

Implementation

Chopsticks

We manage the use of chopsticks by making use of processes. When a philosopher gets hold of a chopstick, the state of the chopstick goes from "available" to "gone". Whenever a philosopher requests a chopstick then they need to provide the process ID of the chopstick in order for us to identify which chopstick the philosopher is asking for. This approach is based on the state diagram provided

inside the assignment document. If a chopstick is "gone" then no other philosopher can access it but they can still request it. These requests are queued inside the process mailbox and addressed once the chopsticks are "available" again. After a philosopher is done using a chopstick they return it.

```
defmodule Chopstick do
    def start do
        stick = spawn_link(fn -> available() end)
    end

    def available() do
        receive do
          {:request, from} -> send(from, :granted); gone()
          :quit -> :ok
        end
    end

    def gone() do
        receive do
          :return -> available()
          :quit -> :ok
        end
    end

    def request(stick, timeout) do
        send(stick, {:request, self()})
        receive do
          :granted -> :ok
          after timeout ->
            IO.inspect("Timeout")
            :no
        end
    end

    def return(stick) do
        send(stick, :return)
    end

    def quit(stick) do
        send(stick, :quit)
    end
end
```

Philosophers

The philosophers switch between the dreaming, waiting and eating states in my implementation. Each philosopher has 5 parameters:

- *Hunger*: The number of times the philosopher needs to eat in order for them to be completely fed.
- *Left and Right*: The left and right chopstick's process ID.
- *Name*: Name of the philosopher.
- *Ctrl*: The controller process keeps track if a philosopher is done or not.

Besides this the philosopher switches between the following states (which also define their behaviour):

1. ***Eating*** → A philosopher only reached this state upon acquiring both chopsticks. The philosopher eats some food. After that, they release both chopsticks as their hunger has been reduced a bit. Now they go back to dreaming. The hunger parameter is reduced by one each time a philosopher eats something.

```
def eating(hunger, left, right, name, ctrl) do
    IO.inspect("#{name} eats a bite")
    Chopstick.return(left)
    Chopstick.return(right)
    dreaming(hunger - 1, left, right, name, ctrl)
end
```

2. ***Dreaming*** → When the philosopher gets to this state then the philosopher goes to sleep for some time. If the hunger has been reduced to 0 then the philosopher is satisfied and would not eat any more food. Thus the process for that philosopher can be terminated. Otherwise, the philosopher wakes up after a certain amount of time and goes into the waiting state.

```
def dreaming(hunger, left, right, name, ctrl) do
    IO.inspect("#{name} is dreaming")
    sleep(hunger*100)
    case hunger do
      0 -> send(ctrl, :done)
      _ ->
        IO.inspect("#{name} has woken up and is now waiting for her
        sticks to eat more")
        waiting(hunger,left,right,name,ctrl)
    end
end
```

3. ***Waiting*** → When the philosopher has woken up from the dreaming state and is still hungry she starts waiting for the chopsticks and starts sending requests for them. If the philosopher manages to acquire both chopsticks then they can move on to the eating state. The first implementation of this state involved requesting just the first chopstick first, and waiting to receive an ":ok" message, we have a small delay after which we send a request for the other chopstick. In the case, the philosopher doesn't manage to get either of the chopsticks then both the chopsticks are returned by the philosopher so that we don't end up in a deadlock scenario. Upon failing to acquire chopsticks the philosophers just return to the dreaming state.

```
def waiting(hunger, left, right, name, ctrl) do
    IO.inspect("#{name} is now waiting with hunger:#{hunger}")
    case Chopstick.request(left, 200) do
      :ok ->
        IO.inspect("#{name} has acquired the left chopstick")
        sleep(250)
        case Chopstick.request(right, 200) do
          :ok ->
            IO.inspect("#{name} has acquired both chopsticks")
            eating(hunger, left, right, name, ctrl)
          :no ->
            IO.inspect("#{name} has waited for too long for the chopsticks and
            now goes back to sleep")
            Chopstick.return(left)
            Chopstick.return(right)
            dreaming(hunger, left, right, name, ctrl)
        end
      :no ->
        IO.inspect("#{name} has waited for too long for the chopsticks and
        now goes back to sleep")
        Chopstick.return(left)
        Chopstick.return(right)
        dreaming(hunger, left, right, name, ctrl)
    end
end
```

The assignment asks us to generate random seeds for all the philosophers so I use the *:rand.uniform/1* with an argument of 15 to generate random seeds instead of the default value of 5 that's been given in the assignment. I keep the argument value relatively small so that it's easy for me to debug the program. So far I do not encounter any deadlocks with my implementation of this problem.

Asynchronous Requests

As we saw before we first request one chopstick and then the other one, but this time we attempt to request both of them simultaneously. We are also keeping track of a timeout, where upon reaching the timeout the philosopher returns to sleep. There is a possibility in this case that if a philosopher goes to sleep and the message she sends to acquire the chopsticks is eventually answered. This means that the philosopher might get a reply that the philosopher can acquire a chopstick even when it's not in the state to do so. This can be fixed by assigning a unique identifier reference to keep track of the requests.

The philosopher requests both chopsticks and then waits for her request to be approved. If she manages to get approval for both chopsticks then she can eat and then return both chopsticks at once. In the scenario that a timeout event is observed, both chopsticks are still returned. This might not make sense, but this is done as eventually, the philosopher will manage to grab both chopsticks, so we cancel any case where the philosopher might just have one chopstick. The philosopher returns to sleep and a new reference is generated in that case.

The async() function

This function runs till the philosopher isn't hungry any more i.e. the hunger parameter is 0. If a timeout is reached, this function is called again with a new reference. If an :ok message is received then the function is called again with the hunger parameter decremented by 1.

```
def async(0, left, right, name, ctrl, ref) do
    IO.inspect("--###--#{name} has finished eating--###--")
end
def async(hunger, left, right, name, ctrl, ref) do
    sleep(hunger*100)
    IO.inspect("#{name} wants to eat; hunger at #{hunger}")
    case eat(hunger, left, right, name, ctrl, ref) do
      :timeout -> async(hunger, left, right, name, ctrl, make_ref())
      :ok -> async(hunger - 1, left, right, name, ctrl, ref)
    end
end
```

The eat() function

Here the philosopher requests both chopsticks. Then the granted function is called to check if we the philosopher manages to acquire both chopsticks or not. We return chopsticks no matter what the granted function returns followed by returning the value that the granted function returned to us.

```
def eat(hunger, left, right, name, ctrl, ref) do
    Chopstick.request(left, self(), ref)
    Chopstick.request(right, self(), ref)
    case granted(2, ref) do
      :timeout ->
        IO.inspect("#{name} doesn't get to eat this time")
        Chopstick.return(left, ref)
        Chopstick.return(right, ref)
        :timeout
      :ok ->
        IO.inspect("#{name} eats a bite")
        Chopstick.return(left, ref)
        Chopstick.return(right, ref)
        :ok
    end
end
```

The granted() function

This function keeps track of the message we receive upon requesting the chopsticks. If we get an :ok message from both chopsticks then we can return :ok. If we hit a timeout then :timeout is returned instead. We also pattern-match against our reference to make sure that we are not answering a message from a previous iteration. My implementation of this function is provided below:

```
def granted(0, ref) do :ok end
def granted(n, ref) do
    receive do
       {:ok, ^ref} -> granted(n-1,ref)
       after @timeout -> :timeout
    end
end
```

A waiter

A better approach to solving this problem could be if there are always 2 philosophers eating at any given time. If a waiter could control the behaviour of the philosophers then he would make it so that the philosopher at the position "rem(n,5)" and the philosopher at position "rem(n+2,5)" would be the ones eating at any given time. This means that both of these philosophers start and finish eating at the same time, and once they are done eating they both return their chopsticks and the n value is incremented, thus the next set of philosophers can start eating and so on. This way we go in a cyclic pattern and all the philosophers are fed eventually. This does remove the element of randomness, but if the waiter is capable of controlling their behaviour then I think that this might be a good enough solution to this problem.

Benchmark

For benchmarking this problem I use the *:erlang.timestamp* function to first find the time before we spawn the first process at the beginning of the Dinner.start() function and another one at the end in the base case of the Dinner.wait() function. We add the time we get at the beginning as a parameter for all functions in the dinner module (aside from start()). After all the processes have been terminated and we have taken the time after that. We print out the difference between the two timestamps to find the time taken by the entire program.

We find that for different seeds for the hunger parameter for each philosopher, we get an average time of 0.45 seconds of execution time and just 0.11 seconds if we do it with the default seed value of just 5 as the hunger parameter (testing with 10 iterations). For the case of the Asynchronous requests we end up with an execution time of a mere 2.1 milliseconds for randomized seeds and just 1.2 milliseconds for the case of the default seed (testing with 100 iterations). For all of these, the ratio between Iterations without deadlocks and successful iterations is 1 i.e none of the cases hit a deadlock scenario. This remains true as long as the value for the sleeping period in case of the asynchronous requests is greater than 0, as otherwise, it would mean that all philosophers are active at once thus they would all try to access the chopsticks at once and we can hit a deadlock scenario quite quickly.