Dining Philosophers

Ruxandra-Stefania Tudose

Spring Term 2024

Introduction

This report aims to open a window onto how concurrent programming concepts can be understood and seen through the implementation of the Dining Philosophers problem. Key terms such as processes, messages, deadlocks and implicit deferrals will be introduced. Since it is quite an abstract concept, as far as my approach is concerned, I focused on understanding the guidelines and suggestions provided during lectures and most importantly, getting the essence of concurrent programming through this problem.

The problem context and concurrent programming

In a nutshell, the problem consists in five philosophers standing at a round table whose intention is to eat using the provided chopsticks. The issue? Not everyone can eat simulatneously and no one has to starve. Since not everyone can eat at the same time, we have to 'lock' the resources, namely the chopsticks and use them only under favorable circumstances by limiting the access to them. And this is precisely when, from my understanding, concurrent programming comes in handy. Both the philosophers and the chopsticks represent individual processes, meaning that they control their own memory and data structures and at the same time, they communicate with one another by sending messages through the actor model, which can also be seen as a FSM. Moreover, implicit deferral will also be used since the task mentions that not handled messages remain in queue. Last but not least, by having a concurrent programming model approach, we can implicitly increase efficiency since we can go to a parallel execution model hardware-wise.

The chopstick

The chopstick is a process which has two states: it either is available or it is gone. A philosopher can only use a chopstick if it is available, otherwise it has to wait for it so that he can eat. Below, I have included the simple initial version of the code for the two states according to the FSM included in the PDF instructions:

```
def start() do
  #initialize a process which dies if the mother process does
  spawn_link(fn() -> available() end)
end

def available() do
  receive do
    {:request, from} ->
      send(from, :granted) #send granted message to the process
      gone() #go into gone() state - chopstick not available
    :quit ->
      :ok
  end
end

def gone() do
  receive do
    :return -> #if a chopstick is returned it becomes available
       available()
    :quit ->
       :ok
  end
end
```

One problem that occurs is when all philosophers want to eat at the same time: this is a deadlock. How do we solve it? Well, one approach is quite similar to what happens in networks and communication when we do not know if a packet is lost or takes too much time for it to arrive - we, therefore, introduce a timeout. If the waiting time expires, the chopstick should be returned and we exit the deadlock, however the strength of the philosopher has to be decreased by one since we have given up. It should be taken into account that *self()* gives the current process' PID while the *send* command is used to send messages in between processes. I have included below the *request/2* method with the updated timeout:

```
def request(stick, timeout) do
   #send message to process with the chopstick and philosopher PID
   send(stick, self())
   receive do
     :granted ->
   :ok
```

```
    after timeout -> #condition if timeout is exceeded
    :no
    end
 end
```

The philosophers and the dinner

The philosopher is also an important process in this problem and can find himself in three different states: either dreaming, eating or waiting for the chopsticks. As indicated in the instructions, the *start/7* method has been created which spawns a new philosopher process. An extra parameter has been included for the interactive simulation - the *gui* parameter.

```
def start(hunger, strength, left, right, name, ctrl, gui) do
    spawn_link(fn -> dreaming(hunger, strength, left, right, name, ctrl, gui) end)
end
```

By using pattern matching, *dreaming/7* method either fulfills the conditions when he is not hungry (hunger = 0) so we do not care about the chopsticks, when the strength is equal to zero the philosopher has died and last but not least when after he is dreaming he is waiting for chopsticks. In addition, *defp* defines a private function which can only be accessed in the module it finds itself in.

```
defp dreaming(0, strength, _left, _right, name, ctrl, gui) do
    #philosopher no longer hungry (hunger = 0)
    IO.puts("#{name} is happy, strength is still #{strength}!")
    send(gui, {:action, name, :done})
    send(ctrl, :done)
end
defp dreaming(hunger, 0, _left, _right, name, ctrl, gui) do
    #starved philosopher - chopsticks don't matter
    IO.puts("#{name} is starved to death, hunger is down to #{hunger}!")
    send(gui, {:action, name, :died})
    send(ctrl, :done)
end
defp dreaming(hunger, strength, left, right, name, ctrl, gui) do
    #dreaming philosopher
    IO.puts("#{name} is dreaming!")
    send(gui, {:action, name, :leave})
    delay(@dream) #dreaming set delay
    IO.puts("#{name} wakes up")
    #waiting for chopsticks
    waiting(hunger, strength, left, right, name, ctrl, gui)
end
```

While the solution one has up until now is convenient, it can still be improved. How? Well, a philosopher needs two chopsticks to eat so just like in networks and communication, instead of sending a request for one of the chopsticks and wait for the response (which may not be the desired one), then send the request for the second chopstick and so on and so forth, time can be saved by sending the requests for both chopsticks at the same time by doing an ansynchronous request. Only then, when both chopsticks are available, can the philosopher eat.

In order to do so, the *waiting/7* method has been adjusted. Both chopsticks are requested at the same time and then, the results are stored checked in a case statement using pattern matching. Only if both chopsticks are available, can the philosopher eat.

```
defp waiting(hunger, strength, left, right, name, ctrl, gui) do
  send(gui, {:action, name, :waiting})
  IO.puts("#{name} is waiting, #{hunger} to go!")

  case {Chopstick.request(left), Chopstick.request(right)} do
    {:ok, :ok} ->  IO.puts("#{name} received both sticks!")
                   eating(hunger, strength, left, right, name, ctrl, gui)
  end
end
```

Last but not least, 'The dinner' is the mother process which controls the philosophers and the chopsticks. Here one initializes the processes and at the same if something goes wrong with a process, it controls the errors by killing all the other linked processes.

It should be taken into account that the code included in the PDF instructions has been slightly changed so that it can be used to measure both the time and send as parameter the times each philosopher should eat. The *gui* interactive parameter has also been added.

```
#each philosopher will eat n times
def start(n), do: spawn(fn -> init(n) end)
```

Benchmark

The last part of this assignment consists in running a benchmark in order to see how long it takes the philosophers to eat the number of given times. It should be taken into account that the benchmark results are the ones corresponding to the approach when one is sending requests for both chopsticks at the same time. In addition, the *:os.system_time(:second)* has been used in order to measure time.

As it can be noticed in Table 1, as expected the the larger n is, the more time it takes for the philosophers to eat. By looking at the trend, the execution time is more or less, half of the input n times. Therefore, one is definitely on the safe side to acknowledge that this solution, as far as the execution time is concerned, falls under the *O(n)* time complexity.

n	ExecutionTime[s]
5	3
25	12
50	25
100	47
500	240

Table 1: Benchmark analysis of the time it takes for philosophers to eat n times when both chopsticks are requested at the same time.

Conclusion

Concurrent programming is tricky and proving that a system is deadlock-free is even trickier. I think that it is so essential for one to understand the concepts that involve dealing with this programming paradigm before diving into anything that is related to actually sketching the code since then, everything will seem slightly easier.