Philosophers and Concurrency

Ayda Ghalkhanbaz

Spring Term 2024

1 Introduction

The Dining Philosophers is a common programming problem in concurrency. The problem is described as a group of 5 philosophers sitting at a table with a bowl of spaghetti in front of each philosopher and they are either eating or dreaming (thinking). The challenge is that there are only 5 chopsticks and each philosopher needs 2 chopsticks to be able to eat. In this report, we will delve into the implementation details of the problem and try several ways to solve the deadlocks. At the end, we'll discuss each of these solution's efficiency using some benchmark results. In order to keep this report as short as possible, there are not too many codes included but all the code for this assignment can be found on GitHub.

2 Method

The program implemented for this assignment is divided in three main modules: chopstick, philosopher and dinner. Since the entire code for the dinner module was provided through the assignment, we only cover the details of chopstick and philosopher modules in the following sections. Furthermore, we'll look into some solutions to resolve the potential deadlocks.

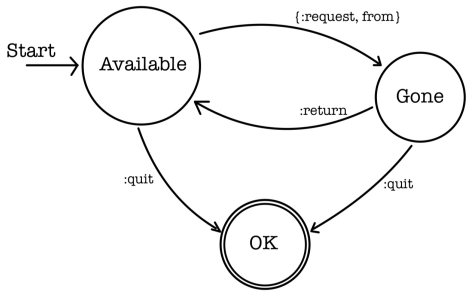

Figure 1: *A simple illustration of FSM representing the chopstick module*

2.1 Chopsticks

The general structure of this module can be illustrated as a FSM (see figure 1) with three states. A new chopstick (process) is created using `spawn_link/1` and as soon as a process is created, it should be `available`. In the `available` state we can either `quit` and go to the state `ok` or send a request for a chopstick (process) to be picked up by a philosopher, and if the `request` is `granted` then we go to the next state, `gone`. The `gone` state works also similarly and can either be returned to the `available` state or `quit` and move to the next state, `ok`. There are two other more functions defined in this module; i.e. `request/1` and `return/1`. These functions are rather similar to interface and are called by philosophers when they want to pick or return a chopstick.

2.2 Philosophers

The philosopher module is also based on a three states FSM (see figure 2), so a philosopher is either `dreaming`, or `waiting`, or `eating`. After creating a process (philosopher), we directly go to the `dreaming` state. The `dreaming/5` function, representing this state, starts with a dreaming delay (as the philosopher is dreaming) and then moves to the next state, `waiting`.

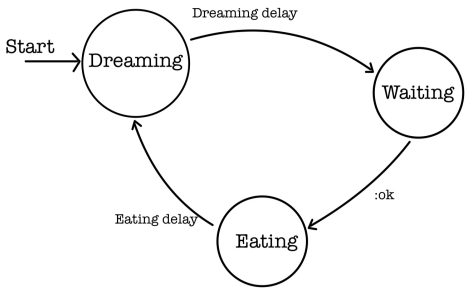

Figure 2: *A simple illustration of FSM representing the philosopher module*

It is noteworthy that each of the three functions in this module takes 5 arguments as follows; `name` (name of the philosopher), `left` (the process ID of the left chopstick), `right` (the process ID of the right chopstick),`hunger` (an integer representing the amount of philosopher's hunger) and `ctrl` (the mother process ID).

In the waiting state, the philosopher sends a `request` for the left chopstick, and after having the left chopstick in the hand, she requests for the right chopstick. Once she has both chopsticks in her hands, she starts `eating`, i.e. goes to the next state. You'll find the `waiting/5` function in the code snippet below.

```
def waiting(name, left, right, hunger, ctrl) do
  case Chopstick.request(left) do
    :ok -> sleep(@delay)
      case Chopstick.request(right) do
        :ok ->
          eating(name, left, right, hunger, ctrl)
      end
  end
end
```

Eating takes a while (an eating delay) and once the philosopher is done eating, the `hunger` parameter is decremented by 1, she returns both of the chopsticks and goes back to the `dreaming` state. This cycle keeps looping until the philosopher's `hunger` is 0, then she will stop eating (quit).

2.3 Deadlocks

Experimenting with the current solution causes deadlocks and decreasing the dreaming delay only slows down the program. However, there are several solutions that break the deadlocks, which we will cover some of them in this assignment.

2.3.1 Starvation

The first possible solution to avoid deadlocks is to add a waiting time limit to the `request` function. In this way, if the philosopher doesn't receive the requested chopstick during a certain amount of time, the `request` is then returned and the philosopher goes back to `dreaming` state. Moreover, an additional attribute is added to a philosopher that indicates her `strength`, i.e. how many times she can send a request and receive a `:no` (not moving to the `eating` state) before she loses all her `strength` and dies.

The potential problem with this solution is that a philosopher can die out of starvation. This issue can be solved by prioritising the requests from philosophers with low strength.

2.3.2 Asynchronous Requests

The second solution is that a philosopher sends the request for the `left` and `right` chopsticks at the same time. This approach included some changes in the `request` and `waiting/6` functions. The `request` function now takes 3 arguments; `left` chopstick, `right` chopstick and a `timeout`. When calling the `request/3` function, two requests are sent to receive the `left` and `right` chopsticks and it returns the requests if it can't receive them in time.

Meanwhile, in the `waiting/6` function, the philosopher requests for both chopsticks once and waits for the response.

Although this approach may seem to work fine, deadlocks can still happen and there is another slight problem that is we may lose track of the requests. This means that if we send a request to both of the chopsticks at the same time and get only one response back, we won't know if the received chopstick is the left one or the right one. However, this can be resolved by making a unique reference for each chopstick.

2.3.3 Non-Circular Ordering

The last approach avoids the circular wait conditions. We already know that philosophers sit at a round table and therefore they all can have a chopstick on their left hand and wait for the right chopstick from the person sitting to the right of them. In other words the sequence of the chopsticks would be `c1-c2-c3-c4-c5-c1`.

However if each philosopher points in the same direction and waits for the other chopstick, deadlocks will never happen. This might be a little confusing at first but hopefully the simple illustration of this approach shown in figure 3 will help understanding the solution. So considering our very first

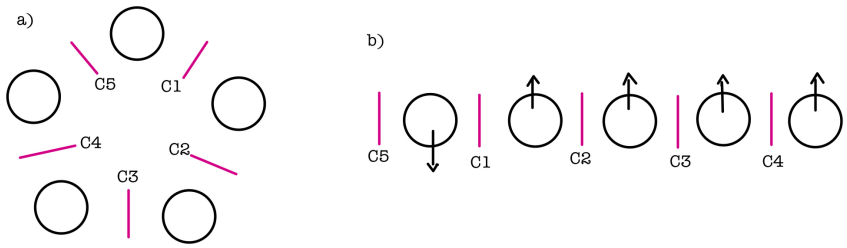

Figure 3: *a)A simple illustration of the circular way of sitting for philosophers in the 3 first approaches. b) A simple illustration of the non-circular way of sitting for philosophers in the last approach causing no deadlocks (the arrows show the philosopher's sitting directions)*

and naive solution, the only change that is needed to reach this approach is to send the first chopstick `c1` as the left one for the last philosopher, instead of sending `c5` as the left chopstick.

3 Result

The last approach, non-circular, was benchmarked with different hunger levels, and for each number of hunger the best time out of 3 has been chosen. The results are presented in table 1.

Hunger	Time(s)
5	0.4
10	0.7
20	1.4
40	2.5
80	5.6
100	7.0
200	14.0
500	34.0
1000	68.5

Table 1: *Execution time of the non-circular approach in s*

4 Discussion

According to the outcomes from table 1, we can observe that with a double hunger, the execution time also increases with the same factor approximately. This leads to a time complexity of $\mathcal{O}(n)$ for the non-circular approach. However, the benchmarking for the two other approaches indicated a similar time complexity, which couldn't be covered in this report due to the report restrictions.

So far we explored three ways of preventing deadlocks where each of these solutions has its own effectiveness and limitations. The first approach, starvation and timeout, prevent philosophers from waiting infinitely by adding a waiting time limit (timeout). Additionally, this approach introduces a strength attribute to avoid the potential deadlocks. However, there are still challenges with this solution like starving the philosophers to death, or finding the balance between preventing starvation and avoiding deadlocks.

The second approach, asynchronous requests, prevents deadlocks by minimising the possibility of a philosopher blocking another one while waiting for chopsticks. Although this approach seems to be effective, there is still the chance of getting deadlocks if the requests are lost or misinterpreted.

The last and simplest approach, non-circular, breaks the circular wait condition to avoid the deadlocks. However, this solution may not be applicable in scenarios where a circular waiting is a requirement.

In conclusion, each solution comes with its unique trade-offs and the choice of the solution depends on the system requirements.