Dining Philosophers

Dhruba Das

Spring Term 2024

Introduction

By far the most complicated assignment, this one deals with **concurrency**. As the name suggests, we have a round table of five philosophers, who may either dream or eat. They each need two chopsticks in order to eat, and there are only five chopsticks. We will deal mainly with finding a way to break the **deadlock**, which occurs if every philosopher holds a left chopstick and waits perpetually for a right chopstick (or vice versa).

Chop module

The location of a chopstick is represented by a process; the process has two states: **available** and **gone**. We use methods *request* and *return* to ask for and return a chopstick. The *request* method simply sends a :request message to the provided chopstick process and waits for a :granted message in response. The *return* function is used by processes to return a chopstick; it sends a :return message to the provided chopstick process.

The *available* method represents the behavior of a chopstick when it is available. It waits for messages, and when it receives a :request message, it responds with :granted and transitions to the **gone** state. If it receives a :quit message, it terminates.

The *gone* method represents the behavior of a chopstick when it is currently in use. It waits for messages, and when it receives a :return message, it transitions back to the **available** state. If it receives a :quit message, it terminates. An example of code from this module is shown below:

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

Phil module

This module contains the functions which describe the three states a philosopher can be in: **dreaming**, **waiting**, and **eating**. Each philosopher also has 6 "traits" : name, left, right, hunger, strength and ctrl. Name, hunger, left and right are self explanatory; strength is the no. of lives the philosopher has left; ctrl is a a controller process that should be informed when the philosopher is done.

Every philosopher starts out in the **dreaming** state; this state is represented by two functions. In the base case, we assume that the philosopher is no longer hungry (i.e hunger is 0); in that case we simply send a :done message to the ctrl process for that philosopher. In the second function, we add a delay for dreaming using the *sleep* function, and then call the *waiting* function:

```
def dreaming(name, left, right, hunger, strength, ctrl) do
    IO.puts(" #{name} is dreaming")
    sleep(@dreaming)
    waiting(name, left, right, hunger, strength, ctrl)
end
```

The *waiting* function is responsible for requesting for the chopsticks; it simply requests for both the left chopstick and the right chopstick and then calls the *eating*.

The *eating* function adds a delay for the time spent eating, and once done, it calls the *return* function from the chop module to return both the left chopstick and the right chopstick; we then call the *dreaming* function to return to the dreaming state, but with hunger reduced by 1:

```
def eating(name, left, right, hunger, strength, ctrl) do
    IO.puts(" #{name} is eating")
    sleep(@eating)
    Chop.return(left)
    Chop.return(right)
    dreaming(name, left, right, hunger-1, strength, ctrl)
end
```

Experiments

Running the experiments with a dreaming delay of 10 (initially), eating for 20 and a hunger of 100, all philosophers are able to finish eating and return happy with a strength of 5 (max. strength). However, as we increase the amount of time to 20, and then 30, and all the way to 90, the amount of time in which all of them finish eating and return happy increases linearly. And once a delay is added between requesting the first chopstick and requesting the second chopstick, we enter a state of **deadlock**.

In order to get out of the deadlock, we introduce a **timeout** constant, which we give as an argument to the *request* function. Now, if a chopstick is not received after the *timeout* amount of time, it returns a :sorry message.

In response to this, we also change the *waiting* function, which now uses cases to check whether we receive a chopstick; if we do, then we ask for another, otherwise the philosopher now cancels their request and returns any chopsticks received by calling the *return* function, and goes back to dreaming with strength reduced by 1:

```
def waiting(name, left, right, hunger, strength, ctrl) do
    case Chop.request(left, @timeout) do
      :ok ->
        sleep(@delay)
        case Chop.request(right, @timeout) do
          :ok ->
            eating(name, left, right, hunger, strength, ctrl)
          :sorry ->
            Chop.return(left)
            Chop.return(right)
            dreaming(name, left, right, hunger, strength-1, ctrl)
        end
      :sorry ->
        Chop.return(left)
        Chop.return(right)
        dreaming(name, left, right, hunger, strength-1, ctrl)
    end
end
```

After implementing the above changes, the results are different. We now get that all the philosophers finish eating with a strength of 4:

```
iex(13)> Dinner.start()
#PID<0.211.0>
 elisabeth is happy (strength 4)
 simone is happy (strength 4)
 ayn is happy (strength 4)
 hypatia is happy (strength 4)
 arendt is happy (strength 4)
```

Asynchronous requests

Another interesting approach is to request both the chopsticks at once, and then wait for the replies for each of the requests. We implement a *wait* function in the Chop module, which does the waiting and the granting of chopsticks on behalf of the *request* function(which now only handles the sending of the request) as shown below:

```
def wait(ref, time) do
    receive do
      {:granted, ^ref} ->
        :ok
    after
      time ->
        :sorry
    end
end
```

The requesting of the chopsticks is still done in the *waiting* function, with the only difference being that, we now first request both chopsticks, before moving to the cases to check the replies of the *wait* function.

We can also see that in the *wait* function we use references in order to keep track of chopsticks which the philosopher has actually received. This **ref** variable is initialized in the *waiting* function.

Benchmarking

Shown below are the numbers obtained with varied levels of hunger (for asynchronous requests, with all of the philosophers at strength 5):

Hunger(n)	**Time taken (ms)**
10	37
20	78
50	204
100	416
1000	4599

Table 1: Time in milliseconds

Conclusion

As stated before, concurrent programming is very complicated, and as we can see, there are many factors to keep in mind. This is seen when, even with asynchronous requests, we aren't able to improve the times as drastically as one would expect AND STILL left with the possibility of a deadlock occurring. However, in the end, we could either go with this, OR (as revealed at the end of the lecture) one could simply change the order in which the chopsticks are picked up. We aren't able to exactly tell whether the system is fair, but it does indeed work.