The Dining Philosophers Problem

Has-san Nazar

March 12, 2015

1 Introduction

In this assignment we are told to implement an Erlang application that shows the behavior of five philosophers that are seated at a round table. Each of the philosopher switches between being in a thinking state(or sleeping-state if you will..) and being in a wake state. Every philosopher has one chopstick to his right and one to his left. Thus we have 5 philosophers around a round table with exactly 5 chopsticks in total lying around them.
When a philosopher enters a wake state he will look for these eating-sticks to his left and right. If he finds two sticks adjacent to himself, he can start eating for a set amount of time. And if he does not find two chopsticks, he has to wait. Our task was here to implement a small program that will show us the behavior of these 5 philosophers and then run some experiments to examine the result of the problem. The following Report will the steps that were taken in order to finish the assignment.

2 Approach

Our approach in this assignment was to sequentially build up the application. Firstly defining modules in Erlang that would work much like objects in a object oriented language.

2.1 A Chopstick

First we need to be able to define a chopstick for our problem. This chopstick will be the cutlery that the philosophers will be in need of to eat. If one stick is available, then the philosopher should be able to pick it up. If it is already picked up(by another philosopher), then it should be in a state so that no one else can pick it up. Until the philosopher currently holding the stick has put it down. Furthermore, there should for every state be a 'quit' phase that will be used to terminate the program. These states of the chopstick is shown in figure 1.

Every single-lined circle represents a state in which the chopstick can be in. The functions that were implemented for the Chopstick module was:

1. **available()** The state at which the stick is available to be picked up by a philosopher, when called upon it returns a process message *available* to the caller.

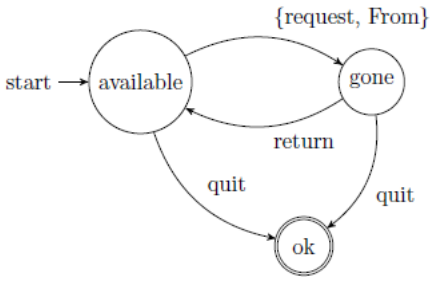

Figure 1: Picture Showing the different states of a Chopstick

2. **gone()** This state is active when a stick has been picked up by a philosopher, it awaits a *return* from the philosopher function to return to the available state

3. **request()** This function is called by the philosopher module and will ask for two of its adjacent sticks,we then call the granted() function

4. **granted()** We handle the allocation of sticks for a philosopher through a asynchronous request. If both calls return the message *available* from available() we know that the sticks are present and not taken. We now send a message *taken* to available(). This sends both sticks to the gone() state.

5. **quit()** Terminates the program.

2.2 A philosopher

The philosophers are those who are supposed to use the 5 processes that have been created of the chopstick module. Also we will create 5 processes of this philosopher module to resemble 5 philosophers at a round table. The philosophers have been implemented to(much like the chopstick module) follow different states. Either a philosopher is in a thinking state during a random interval of time, or he is awake and looking for chopsticks.If he finds chopsticks adjacent to himself available then he is in a eating state for some time, after this he returns back to sleep mode. The philosopher loops like this a given number of times.

These states are(Note that the attributes are not shown for the methods):

1. **dream()** This state is where the philosopher sleeps for a randomly given time

2. **order()**
   Version 1: The function sends a request to the chopstick module with the philosophers adjacent chopstick Processor Id's. The chopstick module then checks if sticks are available and returns true or false. If true is received the philosopher can now eat
   Version 2: The philosopher makes an order to the waiter, and the waiter either returns *grant* or *nogrant*, if he is granted then he can go to request the sticks from the chopstick module.

3. **eat()** If the philosopher has 2 sticks he can start to eat for a randomly generated time, then start to dream again.

At first the waiter module and **order()** was not implemented, rather we made direct calls to the chopstick module through the philosopher to get a hold of his adjacent sticks. At section 3.1 we will discuss the issue of "**Breaking the Deadlock**" and at that point we are not considering the waiter module optimization rather working to the deadlock from requesting the sticks directly to chopstick module.

2.3 Dinner at the table

When we have the necessary parts ready(the philosopher and chopstick module) we now create a small module that will only execute the so called "*dinner*" of the 5 philosophers. This module creates 5 processes of the Chopstick module and in addition, it creates 5 processes of the philosopher module we created at the previous section, this to start the simulation.

3 Experiments

3.1 Breaking the Deadlock

At the first run(without the waiter implementation), we immediately experienced a so called *deadlock*. That means that the simulation abruptly stops due to that every philosopher is requesting a chopstick at the same time, which yields that no one can start eating, hence the program gets stuck in a recieve clause in the chopstick module. The reason for this was the fact that all to many philosophers were requesting eating-sticks at the exact same time. **This has 2 main reasons:**

1. **The random:uniform() function** uses a pattern to generate a pseudo-random number for us. This means in simple terms that as long as we feed the function with the same parameters we will keep getting the exact same values. This lead to that every processor was waking up from their initial sleep at the exact same moment in time and thus 10 hands were requesting 5 eating sticks at the same time, which ended up in a deadlock.

   This issue was solved by providing every philosopher with a unique seed value.

   ```
   start(Hungry, Right, Left, Name, Ctrl,Waiter)->
      {N1,N2,N3} = now(),
      spawn_link(fun()-> random:seed(N1,N2,N3),
        init(Hungry, Right,Left,Name,Ctrl,Waiter) end).
   ```

2. **The Request() issue** did not have support for multiple calls to the function. For every call recieved it checked if the chopstick was available by making a call to the available() function. If a value *true* was returned it granted the philosofer permission to "pick it up". But at the point where all philosophers were calling for sticks at the same time there came a point where all sticks were gone(), but we were still getting requests

from the processes. So by adding a *after* to the function we could tell the calling module(the philosopher) that: *"the stick that is being requested for is busy, try again another time!"*

```
available()
   receive
      ...
   end
after Timeout->
   Phil ! no
```

3.2 A waiter

Even if the program with the above made changes was working, it was not flawless. We make alot of reduntant calls to the chopstick module just to get denied. A better alternative would be to have a observer from above, a so called *waiter* that only lets a given amount of philosophers eat at the same time. The waiter would never allow for a philosopher to request sticks if a adjacent philosopher is already eating or holding the sticks. This is because the new guy would need one stick that is already busy. To solve this issue we let the waiter get 2 orders at one time from 2 different processes. The waiter checks if the incoming orders need different chopsticks or not. Only if 2 processors make an order that are in need of entirely different chopsticks will the waiter give them permission to eat. This way we can always avoid a possible kollision and hence a possible deadlock.

The code below shows a part of the waiter module implementation at which the functions looks for 2 tuples recieved from 2 different philosopher processes from the **version 2** order() function of the philosopher module. In this part we check simply if the acquired chopstick processor ID's are in any way equivalent to each other, if they are not, then the 2 processer are granten permission to eat.

```
receive
   {Right1, Left1, F1}->
      receive
         {Right2, Left2, F2} ->
            if
               Right1/=Left2 ->
                  if
                     Left1/=Right2 ->
                        F2 ! grant,
                        timetowait();
                     true ->
                        F2 ! nogrant,
                        permission(Right1, Left1, F1)

                  end;
             true ->
                F2 ! nogrant,
                permission(Right1, Left1, F1)
```

```
                end;
          done ->
          ...
       end
end.
```

4 Evaluation

Using the waiter module and asynchronous requests we ran some tests of the application whilst having the hunger on 5 and varying the sleep time intervals of the philosophers.The below table shows the result of the tests.

SleepTime	Arendt	Hypatia	Simone	Elizabeth	Ayn
100-500	5380 ms	6058 ms	7366 ms	7508 ms	5758 ms
10-50	458 ms	839 ms	492 ms	636 ms	488 ms
1-5	75 ms	93 ms	76 ms	82 ms	200 ms

Table 1: Philisopher finish time at varying sleep-time intervals.

5 Summary

The biggest problem of the program was probably to implement the waiter module, although the idéa was pretty clear and simple. The translation of thought to code was something that took some time. Nevertheless the main requirement to easily complete the assignment was to understand the problem itself, before diving in to the code.