# Philosopher 6.1

Simon Yllmark

February 23, 2024

## 1 Introduction

This assignment is more of an thinking assignment rather than a programming assignment. The problem is that we have 5 processes aka philosophers which most of the time execute the function dreaming() which requires no chopsticks and that is good. The problem comes when the philosophers want to eat then them as individuals must request there left and right chopstick. Here comes main problem when multiple processes requires other processes (aka two chopstick processes) and they are not always available.

## 2 Trying solutions

This section is for testing different solutions to solve the assignment and the describing the result. There will then be a discussing the section 3 about how good of a solution they are.

### 2.1 Try - Seminar Code

This refer to the code that Johan showed me during one of the seminars.

#### 2.1.1 Case 1 - Initial setting

Here I tested out the program and documented the ongoing operation. It helped me to better understand how the program works. The interesting thing is is that because the philosophers dream so much between between every operation and they dream in a random time they hardly ever collide with each other and if they do they have to wait for 1 second before committing seppuku(as processes).

#### 2.1.2 Case 2 - No sleep time in request chopstick

This case is the same as the previous case but here the waiting time before committing seppuku is 1 ms and there is no waiting time between each philosopher requesting two chopsticks. This resulted with a lot of philosopher committing seppuku, so it turned out to be a massacre.

### 2.2 Case 3 - The philosopher return request

When a philosopher send a request for a chopstick and if it's not granted then there will be a timeout and the philosopher will go back to dreaming but the request message will remain in the recipient process message box. The idea of this solution is that before the philosopher process start dreaming it first send a return to the recipient process, this will then cancel the first request sent. The result was that this canceled the deadlocks when they occurs or occurred and the other operation continued as nothing happened.

### 2.3 Case 4 - Asynchronous requests

The Idea here is that a philosopher will send a two request to the chopsticks to the left and right of the philosopher process then if the phil process get back two ok atoms back in a row then the phil process can eat. If the phil process get a sorry atom back in the first or the second case then the phil process send two return message in order too cancel the request messages like in case 3 and then the philosopher goes back to dreaming. Here is some code describing the implementation:

```
def chopsticks2(hunger, right, left, name, ctrl,strength,gui) do
   ref =  make_ref()
   Chopstick.request(left,ref)
   Chopstick.request(right, ref)
   case Chopstick.wait(ref,@timeout) do
      :ok ->
        sleep(@delay)
        case Chopstick.wait(ref,@timeout) do
        :ok ->
            eat(hunger, right, left, name, ctrl,strength,gui)
        :sorry ->
            Chop.return(left, ref)
            Chop.return(right, rref)
            send(gui, {:action, name, :leave})
        dreaming(hunger, right, left, name, ctrl,strength - 1,gui)
      end
      :sorry ->
            Chop.return(left, ref)
            Chop.return(right, rref)
            send(gui, {:action, name, :leave})
        dreaming(hunger, right, left, name, ctrl,strength - 1,gui)
   end
  end
```

### 2.4 Case 5 - ABC

At then end of a lecture by Johan Montelious he mentioned that deadlock could be prevented by just changing the order which the chopstick(the resource) is accessed by the phil processes. In the phil process p5 the chopstick order of c1 and c5 are reversed which can be viewed in subsection "Code" and then the dinner process proceeded without deadlocks. Thus, this solution prevented deadlock from occurring at all without adding some additional complexity to the existing program.

### 2.5 Code

```
p1 = Phil.start(:phil, c1,c2,n,s,ctrl,gui)
p2 = Phil.start(:amy, c2,c3,n,s,ctrl,gui)
p3 = Phil.start(:helen, c3,c4,n,s,ctrl,gui)
p4 = Phil.start(:amber, c4,c5,n,s,ctrl,gui)
p5 = Phil.start(:karl, c1,c5,n,s,ctrl,gui)
```

## 3 Discussing

### 3.1 Case 5 - ABC

After some testing with playing around with the chopstick order the result was that if the order is reversed up to the point where one phil process do not have it's chopstick order reverse then there occur no deadlocks, an example can be viewed under subsection "Code1". If the order are reversed for all of the phil processes then deadlock(s) did occur.

### 3.2 Code1

```
p1 = Phil.start(:phil, c1,c2,n,s,ctrl,gui)
p2 = Phil.start(:amy, c3,c2,n,s,ctrl,gui)
p3 = Phil.start(:helen, c4,c3,n,s,ctrl,gui)
p4 = Phil.start(:amber, c5,c4,n,s,ctrl,gui)
p5 = Phil.start(:karl, c1,c5,n,s,ctrl,gui)
```

### 3.3 Case 4 - Asynchronous requests

The main idea here is that a phil process request both chopsticks at the same time so if a deadlock occur it is resolved faster then in case 3. If one of the request fails then the phil process go dreaming straight away. One thing about the solution is that it bring more complexity to the program and the other thing is that there need be a unique identifier for the return and request messages to see which ones are linked. It's a more effective solution than case 3 but adds more complexity to the program.

## 4 Conclusion

Thinking about concurrent processes is complicated and weirdly enough the best solution turned out to be one where the initial order of the chopsticks where reversed. Johan say that in this solution there is an ABC order to the initial request by the phil processes where the chopsticks are listed as A to B to C but I tried that logic which can be viewed in subsection "Code2" which ended up with deadlock occurring even if the solution should prevent them from occurring.

### 4.1 Code2

```
p1 = Phil.start(:phil, c1,c2,n,s,ctrl,gui)
p2 = Phil.start(:amy, c3,c4,n,s,ctrl,gui)
p3 = Phil.start(:helen, c5,c1,n,s,ctrl,gui)
p4 = Phil.start(:amber, c2,c3,n,s,ctrl,gui)
p5 = Phil.start(:karl, c4,c5,n,s,ctrl,gui)
```

### 4.2 Best solution

The solutions that I liked the most was case 3 because when deadlock occurred it handled it and felt kind of intuitive to implement. To prove that no deadlock will occur with the case 5 ABS solution feels like it would be hard if not impossible. It feels like the best solution would be case 4 or a further improved version of that solution that is "better".

### 4.3 Frequent problem occurring

After testing around and attending Johan's lectures a problem that occur is a situation where 2 phil processes are eating at the same time, with 5 chopstick there can only be 2 eating at the same time. Then a problem might arise when 2 phil want to use the same chopstick. The problem where a third phil want to eat should be minor because then that phil just have to wait until some other phil are done eating.

### 4.4 Two main strategies

What would be the best solution would be a solution that handles deadlock as quickly as possible or a solution that prevent deadlocks from occurring at all. Where one is better than the other can depend on the case. We could generalise that there are two main strategies for handling deadlock where one is to handle them when they occur and the other is preventing them from happening and depending on the situation one might be better than the other.