Dining Philosophers Problem

William Frid

Spring 2024

Introduction

The Dining Philosophers problem is a classical computer science problem created 1965 focusing on concurrency theory. The goal is through resource allocation and synchronization, to make 5 philosophers with 5 chopsticks (one between each) finish their meal as fast as possible. (You could say that that philosophers represent processes, and the chopsticks are available resources.) The rules are that each philosopher must first pick up the left chopstick, and hold that until the right one is available. Then he/she should eat followed by putting down the left and right chopsticks (in that specific order). This should be done until he/she is no longer hungry.[1] Note that these rules are somewhat wage as fixes such as starvation can be implemented.

As this is part of the course **KTH course ID1019 Programming 2**, it'll go outside the normal scope of this problem to give a deeper understanding of the problem and thus not just focus on solving it, but more on understanding it.[2]

Because this document contains a rather small amount of code, it's recommended to read it alongside the code written found at https://github.com/williamfridh/ID1019_VT19_TCOMK_Programming_2. Follow the link and go to Assignments/springs.

Chopstick

As the chopsticks have 3 stages (`available`, and tt gone, and `ok` [used for when ending the usage]), it can be created as a finite state machine (FSM). This can be accomplished by generating a new process via `spawn_link/1` that points to a function representing the base state, `available/0`(snippet below). Similarly, we have the state `gone/0`. These states are combined with the functions `request/2` (snippet below) and `return/2` that are called

[1] Dining philosophers problem. Wikipedia. 2024. https://en.wikipedia.org/wiki/Dining_philosophers_problem (2024-02-20).

[2] ID1019 Programming II 7.5 credits. KTH. https://www.kth.se/student/kurser/kurs/ID1019 (2024-02-20).

upon when a philosopher wishes to pick up or return a chopstick. These two functions can both change the status of each chopstick and by being sent via `send/1` from `available/0` and tt gone/0, also return a result.

```
def available() do
    receive do
        {:request, caller} ->
            send(caller, :ok)
            gone()
        {:return, caller} ->
            send(caller, :bad)
            available()
        :quit -> :ok
    end
end
def request(stick, name) do
    send(stick, {:request, self()})
    receive do :ok -> :ok end
end
```

In addition, the process mentioned is started inside the mother process to make sure it dies when the mother process does. There's also a function called `remove/1` added to exit the chopstick process when desired without killing the mother process.

Philosopher

The philosopher code too is an FSM, but one that's created with arguments so that it knows which chopstick is on what side, an integer representing the hunger, a name for identification, and a controller process to keep track of its status. It's 3 possible stages are `dreaming/5`, `waiting/5`, and `eating/5`.

The logic behind these functions is as follows: `dreaming/5` starts with a random sleep, followed by switching state to `waiting/5`. There (as seen in the snippet below), the philosopher tries to grab the left chopstick, followed by the right. These requests will remain in the queue of each respective chopsticks queue until handled (more on this later). Lastly, when both chopsticks are received, the state changes to `eating/5`. There the hunger integer is decreased by 1, chopsticks are returned, and the state is changed back to `dreaming/5` (unless hunger is equal to 0).

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

```
                :ok -> eating(hunger, right, left, name, ctrl)
            end
    end
end
```

Note that the philosophers must be greedy and not put down the left chopstick if the right one is occupied. Doing so does solve the problem, but breaks the rules as this is what causes much of the challenge.

Unordered Returns

Running the program so far one might notice - if enough printing to the console is done - that the philosophers tend to return the chopsticks for each other. This happens because requests sent via `send/2` that have not yet resolved (because of a non-matching case), might suddenly have a much and trigger a return. In practice, this would mean that P1 would return P2's left chopstick, but P2 would still be able to eat even though he/she only has a single chopstick.

To solve this, references are added for each request. This is done to match requests and return them to each other so that only P2 can return his/her chopsticks. Grouping requests and returns together.

Breaking The Deadlock

Testing the base solution yields no good results as there's no resource management whatsoever. Deadlocks occur frequently and fast. Changing the dreaming and eating time doesn't do much to improve this. Neither does an artificial delay between picking up the left and right chopstick nor an exponential back-off wait triggered upon a failed attempt to take chopsticks.

Starvation

One solution is the starvation approach. Each philosopher is given a certain amount of life and a timeout is added for when requesting the chopsticks (`request/3`). Upon timeout, the chopsticks picked up so far are returned, and the philosophers lose one life or die. This does break the deadlock but at a great cost to the philosophers' lives! Fortunately, deaths can be prevented somewhat if protocols for prioritizing requests from starving philosophers (low on life) are implemented.

Asynchronous Requests

One potential approach for better resource management is to make `request/3` and `remove/2` asynchronous since each philosopher needs two chopsticks to

eat. This can be written for instance as seen in the snippet below.

```
def async_request(c1, c2, name, timeout, ref) do
    async_request(c1, c2, name, timeout, ref, 0) end
def async_request(c1, c2, name, timeout, ref, count)
when count == 2 do :granted end
def async_request(c1, c2, name, timeout, ref, count) do
    if (count == 0) do
        send(c1, {:request, ref, self()})
        send(c2, {:request, ref, self()})
    end
    receive do
        :granted ->
            async_request(c1, c2, name, timeout, ref, count + 1)
        after timeout ->
            return(c1, name, ref)
            return(c2, name, ref)
            :timeout
    end
end
```

This approach does however not solve the problem, but it does lead to somewhat better resource management. Deadlocks can still occur, so other options must be explored.

Bottom Line

The only solution so far found to solve the deadlocks is the starvation approach. It does come at a cost, but leads to the meal being finished and not to any deaths (tested with hunger of 1000 and starting life at 5).

Another approach discussed in this assignment was the introduction of a waiter to control the philosophers. While this does seem like a good idea, but adds a great bottleneck to the program. The idea here is to either feed two philosophers at a time and then one alone, alternatively swap between feeding two philosophers sitting across each other and finally feed the last one. This approach does fix the deadlocks, but when the last philosopher is fed, a lot of time is wasted as two could be fed at the same time and time can be saved by other philosophers starting to pick up their chopsticks.[3]

Benchmarks

To measure the performance of the final solution (with asynchronous requests of chopsticks and starvation support), a simple benchmark was run

[3] Dining philosophers problem. Wikipedia. 2024. https://en.wikipedia.org/wiki/Dining_philosophers_problem (2024-02-22).

where the best p performance of 5 runs was selected. All runs was made with each philosopher having a starting life of 5 with 0 loss (thus those tables are not shown). Removing the starvation support could potentially increase the speed of the program, but could be a good idea to keep as a fallback. Doing so can be seen as good practice due to how hard it is to guarantee no deadlocks, which can be extremely important in some programs.

As we can see in table 1, the time increases linearly (more exactly 0.2 seconds per hunger). Proving that this solution does perform somewhat well with time complexity $O(n)$.

Hunger	**Time (ms)**
10	20
20	35
40	80
80	165
100	250
1000	2650

Table 1: Benchmark exploring the time it takes for each philosopher to fishing his/her meal with zero loss in life.

Discussion & Conclusion

In the end, were left with a somewhat functional solution that cannot guarantee no deadlocks. This is to be expected as proving that a solution for this problem doesn't come with deadlocks and great speed is incredibly difficult. Two final options would be to break the rules in either two ways: 1. Flip the order in which one philosopher picks up the chopsticks. This breaks the pattern where for instance all five pick up their left chopstick at once. 2. Make the philosophers less greedy and let them put down the left chopstick as soon as they notice that their right one isn't available.

The Dining Philosophers problem is a challenge that grants a deeper understanding of processes, concurrency, and resource management. It requires a deep dive into the problem as analysis of concurrent computations such as this one is challenging to get a grasp of (especially with proper printing to the console). All important things for a developer who needs to make stable systems that won't freeze.