Parallel Programming

Homework Report

DINING PHILOSOPHERS PROBLEM

Furkan Eryılmaz - 190315043 1st Ed

Onur Yaşar – 190315031 1st Ed

Sinem Gençer - 200315051 1st Ed

Zehra Özeren – 200315034 2nd Ed

I. PROBLEM

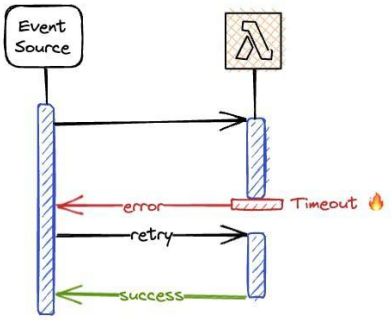
The dining philosopher's problem is the classical problem of synchronization which says that Five philosophers are sitting around a circular table and their job is to think and eat alternatively. A bowl of noodles is placed at the center of the table along with five chopsticks for each of the philosophers. To eat a philosopher needs both their right and a left chopstick. A philosopher can only eat if both the immediate left and right chopsticks of the philosopher are available. In case if both immediate left and right chopsticks of the philosopher are not available then the philosopher puts down their (either left or right) chopstick and starts thinking again.

The dining philosopher demonstrates a large class of concurrency control problems hence it's a classic synchronization problem.[1]

According to this problem, if all philosophers take the stick on their right, for example, they will all have a stick and they will not be able to eat. If they all try to take both of them, then the racing condition encountered in simultaneous operations will occur and whoever acts first (we have no guarantee on this) will be able to eat his meal. And maybe none of them will be able to eat because they will all get a stick. If they all leave their sticks for someone else to eat, then none of them will be able to eat. These types of problems can generally be considered as starvation problems. Accordingly, every philosopher has the possibility of eating, but it is by no means guaranteed that he will eat. For example, one of the philosophers may go hungry in any situation and never have his turn.

Another problem encountered in the problem is deadlock. As a result of a wrong design, a philosopher who takes a single fork and waits for the other philosopher to leave the fork can crash the system. This is the second risk found in the problem.[2]

II. TIMEOUT APPROACH



a) Timeout for Fork Acquisition:

Philosophers try to acquire both their left and right forks. If a philosopher can't acquire both forks after a certain time (timeout), they release the acquired forks and start thinking again.

This prevents a philosopher from waiting indefinitely for a fork and allows others to use it.

b) Randomized Delay:

Introducing a random delay before retrying to acquire forks helps avoid simultaneous attempts by adjacent philosophers, reducing contention.

c) Asymmetric Requesting:

Philosophers could be programmed to request the right fork first and then the left one. This can break the circular dependency that leads to deadlock.

III. EFFECTS OF TIMEOUT APPROACH

a) Prevents Deadlock:

By releasing forks after a timeout, it prevents situations where philosophers are indefinitely waiting for a resource held by another philosopher.

b) Reduces Livelock:

Livelock occurs when philosophers keep releasing and reacquiring forks, unable to make progress. The timeout approach mitigates this by allowing a philosopher to abandon the attempt and think again.

c) Potential Starvation:

There's a risk that a philosopher might starve if the timeout is too short or if the delay mechanism favors other philosophers more frequently.

d) Increased Overhead:

Implementing timeouts and delays adds computational overhead, potentially affecting the efficiency of the solution.

e) Dependency on Timeout Duration:

The effectiveness of the solution heavily relies on the chosen timeout duration. A longer timeout might reduce contention but could increase the likelihood of starvation.

IV. PROBLEM SOLUTION WITH TIMEOUT APPROACH

Within the code, the solution was implemented as extra code within the "eat" method of the Philosopher class.

The program flow is as listed,

- Attempt to pick up the left fork.
- If the left fork is not available, have a wait time of three seconds. If this fails, trigger the "timeout" response, as detailed below for the right fork.
- If the left fork is picked up successfully, then set the left fork's status as "picked up", to prevent others from going for it.
- Then, attempt to pick up the right fork.

During the normal workflow of the code, this point would be where the right fork is picked up successfully. However, with the timeout method implemented, the program continues as below,

- Implement a time limit of 3 seconds for the attempt at picking up the right fork.
- If the right fork is still not unoccupied after the time passes, print out a line to indicate that a timeout has occurred.
- Release the left fork and switch its status back to "not used".

In the case that the right fork becomes available during the waiting duration,

- Set the right fork's status to "picked up" and print a line to indicate that the philosopher now has both forks and is able to eat.
- Have the philosopher eat his spaghetti as he normally would.
- Have the philosopher "place down" the left fork first and release its lock.
- Have the philosopher then "place down" the right fork as well and release its lock.

```
    def eat(self):
        left_fork_successful = self.left_fork.lock.acquire(timeout=3)
        if left_fork_successful:
            self.left_fork.owner = self.index
            self.left_fork.picked_up = True
            time.sleep(5 + random.random() * 5)
            right_fork_successful = self.right_fork.lock.acquire(timeout=3)
            if not right_fork_successful:
                print(f"--- Philosopher #{self.index} timed out acquiring the
RIGHT fork. Releasing the left fork ---")
                self.left_fork.lock.release()
                self.left_fork.picked_up = False
                self.left_fork.owner = -1
```

Code Snippet

Essentially, the inspiration for this solution takes its roots from how human beings in real life would *most likely* eat together if they, for some reason, share their utensils.

In general, if a resource is not available after a certain duration, the human will stop waiting for it and switch their goal from "eating" to "thinking", which is the "idle" status in this case.

As such, in our code as well, the philosophers will wait for a specified amount of time for either the left fork or the right fork to be released to pick it up. If the time is up, the philosopher will cease waiting, giving up on eating for the moment and going back to thinking.

If, by chance, the same philosopher is unfortunate enough that whenever he attempts to eat, the forks happen to be unavailable until after his wait duration runs out, this will mean that he will **starve**. Starvation in this case refers to the philosopher having to continuously wait for a fork, give up, keep thinking, and try again, and have this be a loop. However, this is not a big issue due to the food on each philosophers' plates being limited. It is likely that at least one philosopher will soon finish his food *even in the worst case scenario* and this will increase the chances of the starved and misfortunate philosopher of ours to be able to finally have a taste.

Considering that it only takes a few eating sessions per philosopher for the plated food to be finished, the "starvation" issue is something small enough to be ignored for a problem of such small scale. However had this been a problem of a bigger scale, say, something involving hundreds of philosophers, then it would not be ideal as it could result in some less lucky philosophers starving for a much longer time.

V. CONCLUSION

In conclusion, the philosopher problem was solved by means of "timeout method" where a philosopher will wait for a specified amount of time for an occupied fork before (if they have any) releasing their own fork and reverting back to the thinking state.

While this solution does not entirely eliminate potential issues, it still greatly reduces them and gives us a luxury to ignore them due to their small scale and unlikelihood of posing any sort of threat that matters.

There are other methods that would suit better (yet being more difficult to implement) for a similar case had the problem covered a larger scale (hundreds of philosophers example as mentioned before), however our team chose this particular solution due to its simplicity, straightforwardness, ease of implementation and effectiveness at small scale. After all, a computer engineer is more often than not advised to take the "easier" route, following the simple logic of, "if it is not broken, don't fix it".

VI. REFERENCES

[1] - https://www.javatpoint.com/os-dining-philosophers-problem2

[2] - https://bilgisayarkavramlari.com/2012/01/22/filozoflarin-aksam-yemegi-dining-philosophers/