PARALLEL PROGRAMMING REPORT

190315058 - Havva Beste Tekçeli

190315021 - Yaren Mamuk

190315028 - Mehmet Can Tekin

Contents

1. INTRODUCTION

1.1 DINING PHILOSOPHERS

The Dining Philosophers problem is a classic synchronization problem in computer science and concurrency theory that illustrates challenges with resource allocation and avoiding deadlocks in a concurrent system.

The problem is framed around a scenario where a certain number of philosophers are seated around a dining table, and each philosopher alternates between two activities: thinking and eating. There are bowls of spaghetti on the table, and the philosophers require two utensils (forks or chopsticks) to eat.

The rules for the philosophers are:

1. Philosophers spend time thinking and eating, but they require two utensils to eat.
2. Philosophers pick up the utensils on their right and left to eat.
3. A philosopher can only pick up a utensil if both required utensils are available (i.e., not being used by neighboring philosophers).
4. Once a philosopher finishes eating, they put down their utensils and start thinking again.

The challenge arises when multiple philosophers attempt to pick up the available utensils simultaneously, potentially leading to a deadlock. For instance, if every philosopher picks up the fork on their right simultaneously, they would all be waiting for the fork on their left, resulting in a deadlock where no philosopher can eat.

Solving the Dining Philosophers problem involves designing a solution that ensures:

1. **Mutual exclusion:** Only one philosopher can use a utensil at a time.
2. **Deadlock avoidance:** No situation arises where all philosophers are waiting indefinitely for a utensil held by another philosopher.

The Dining Philosophers problem is a fundamental concept in concurrent programming and demonstrates the complexities of managing shared resources among multiple processes to prevent issues like deadlocks and resource contention.

2. ARBITRATOR SOLUTION

Arbitration is a way to resolve disputes outside of court. When two parties have a disagreement, they can agree to bring in a neutral third party, called an arbitrator, to make a decision. The arbitrator listens to both sides of the argument, examines evidence, and then makes a binding or non-binding decision, depending on the arbitration agreement.

2.1 ARBITRATOR SOLUTION FOR DINING PHILOSOPHERS

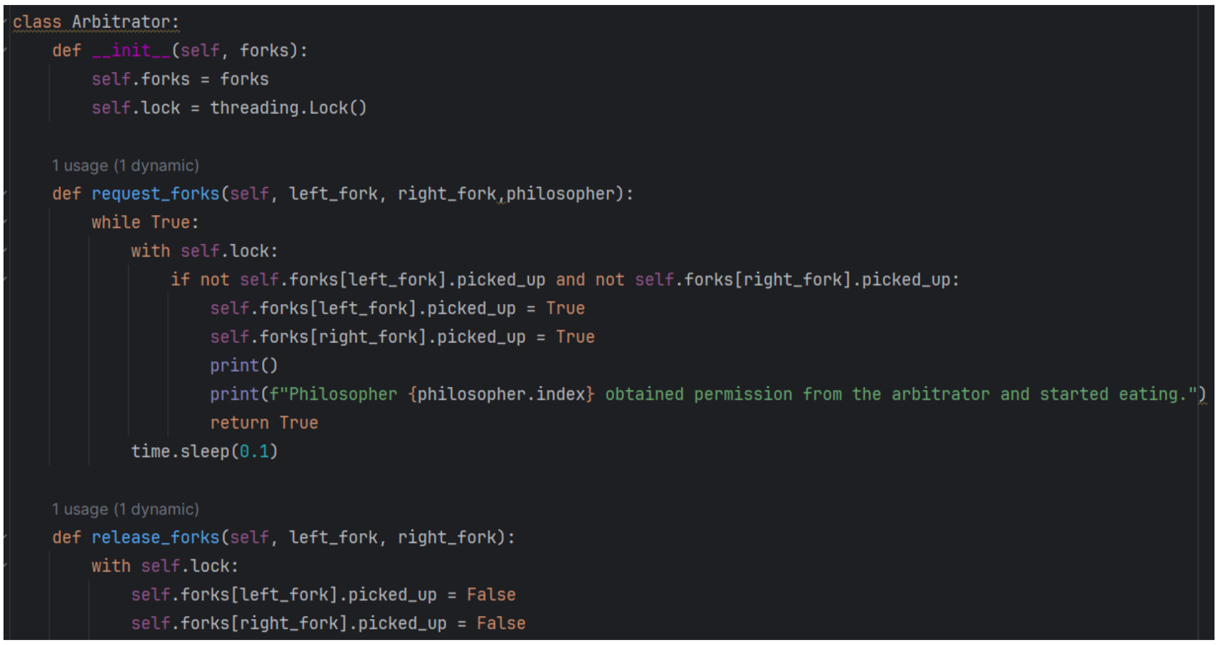
In the context of the Dining Philosophers problem, an arbitrator solution refers to using a centralized entity or mechanism to control and manage the behavior of the philosophers (or processes) to ensure they can access the shared resources (like forks or chopsticks) without leading to a deadlock or resource contention.

An arbitrator solution in this problem might involve introducing a central arbiter or controller that manages access to the forks. The philosophers would request permission from the arbitrator before picking up the forks, ensuring that only a certain number of philosophers can eat simultaneously, preventing situations where all philosophers try to pick up both adjacent forks simultaneously (which could lead to deadlock).

This solution helps in controlling and coordinating access to shared resources, preventing deadlocks and ensuring that all philosophers get a chance to eat without contention.

3. IMPLEMENTATION THE ARBITRATOR SOLUTION TO OUR CODE

In this homework, we use python file which already implemented dining philosophers problem in the class. We added a few new pieces of code to convert this implemented python code into our arbitrator solution of choice. First of all, we need an arbitrator, separate from the philosophers, who will control the philosophers. This arbitrator will decide whether philosophers can eat their spaghetti. While making this decision, the arbitrator controls the forks on the right and left of the philosopher who wants to eat and allows them to eat. After taking permission to eat, the philosopher tells the arbitrator that he left the forks behind after finishing the eating sphagetti’s. In the next time of eating, the philosopher asks the arbitrator permission to eat again.



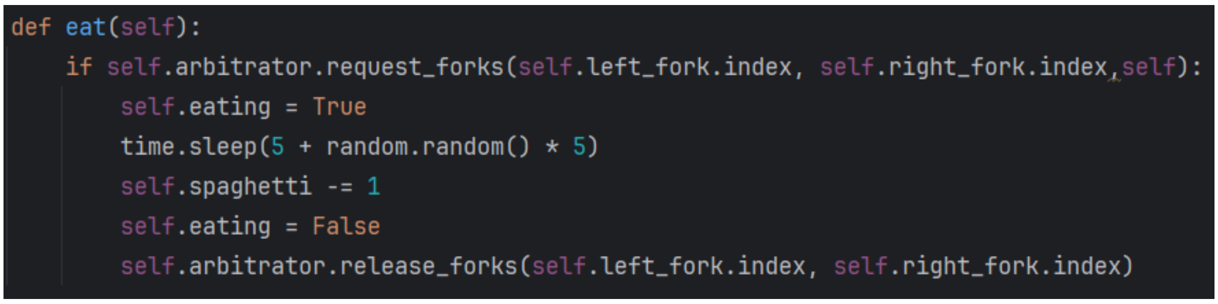
```python
class Arbitrator:
    def __init__(self, forks):
        self.forks = forks
        self.lock = threading.Lock()

    1 usage (1 dynamic)
    def request_forks(self, left_fork, right_fork,philosopher):
        while True:
            with self.lock:
                if not self.forks[left_fork].picked_up and not self.forks[right_fork].picked_up:
                    self.forks[left_fork].picked_up = True
                    self.forks[right_fork].picked_up = True
                    print()
                    print(f"Philosopher {philosopher.index} obtained permission from the arbitrator and started eating.")
                    return True
            time.sleep(0.1)

    1 usage (1 dynamic)
    def release_forks(self, left_fork, right_fork):
        with self.lock:
            self.forks[left_fork].picked_up = False
            self.forks[right_fork].picked_up = False
```

Figure 1: Code

While implementing this solution method into the code, we added arbitrator as a separate class. Every time any philosopher wants to eat, they request forks for eating to this arbitrator. If arbitrator gives the permission, after the eating they inform about the arbitrator to released their fork.



```python
def eat(self):
    if self.arbitrator.request_forks(self.left_fork.index, self.right_fork.index,self):
        self.eating = True
        time.sleep(5 + random.random() * 5)
        self.spaghetti -= 1
        self.eating = False
        self.arbitrator.release_forks(self.left_fork.index, self.right_fork.index)
```

Figure 2: Code

4. CONCLUSION

As a result, the arbitrator solution has enabled every philosopher to enjoy his meal without being deadlocked. Although it provides the solution in general, it allows philosophers who come and have the opportunity. Therefore, while it may be fair, it may not always be completely efficient. This efficiency can be changed by making various changes to the arbitrator's selection algorithm.