Dining Philosopher Problem Arbitrator Solution

Group BEEM

Ebru Çelik – 200315063

Melisa Şahin – 200315046

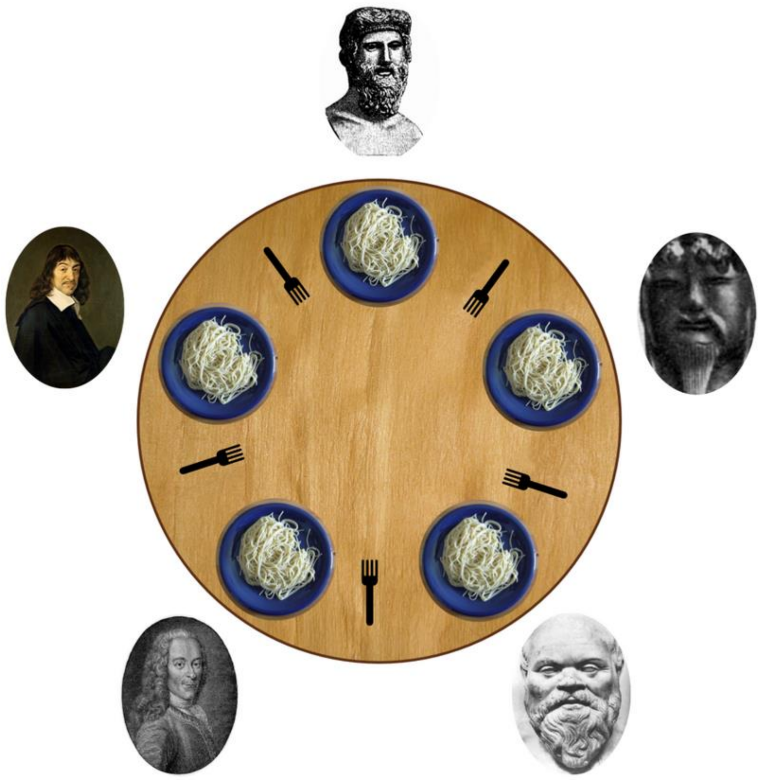
Elif Melike Özçay – 200315019

Batuhan Çetin - 200315024

Introduction to the problem:

The **Dining Philosophers Problem** was first given by Edsger Dijkstra in 1965. This problem is faced by **5** philosophers sitting around a circular table. These philosophers can only eat if they have both left and right chopsticks; otherwise, they will sit and think (without eating) until they starve.

Five philosophers dine together at the same table. Each philosopher has his own plate at the table. There is a fork between each plate. The dish served is a kind of spaghetti which has to be eaten with two forks. Each philosopher can only alternately think and eat. Moreover, a philosopher can only eat his spaghetti when he has both a left and right fork. Thus two forks will only be available when his two nearest neighbors are thinking, not eating. After an individual philosopher finishes eating, he will put down both forks.



Several solutions have been proposed to address this problem. Some of the well-known strategies include:

- Odd-even
- Timeout
- Resource Hierarchy
- Maximum Number of Philosophers
- Arbitrator
- Chandy- Misra
- Dijkstra's Solution
- Message Passing
- Monitors

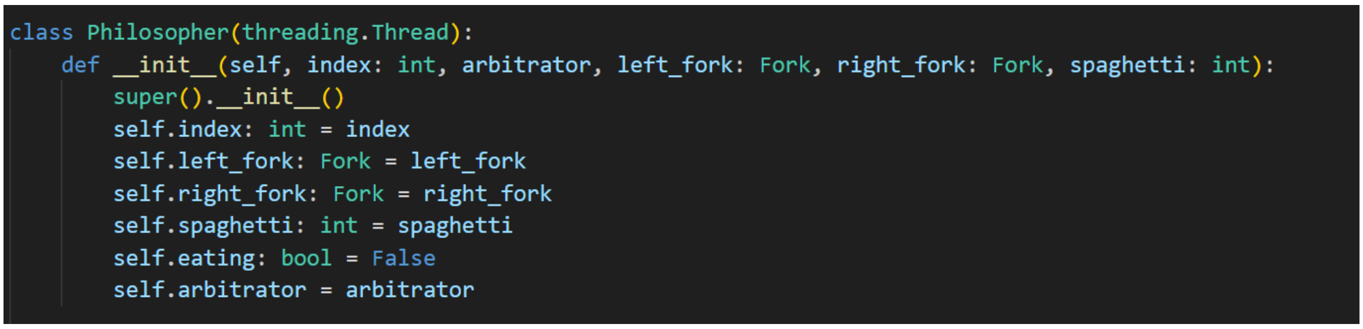
We chose the Arbitrator Strategy among these solution strategies.

This approach is to guarantee that a philosopher can only pick up both forks or none by introducing an arbitrator, e.g., a waiter. In order to pick up the forks, a philosopher must ask permission of the waiter. With this strategy, it can be guaranteed that an individual philosopher can only pick up both or none of the forks. We can imagine a waiter performing this service in the case of dining philosophers. The waiter gives permission to only one philosopher at a time until he picks up both forks. Putting down a fork is always allowed. The waiter can be implemented in code as a mutex. This solution introduces a central entity (arbitrator) and can also result in reduces parallelism. If one philosopher is eating and another request a fork, all other philosophers must wait until this request is fulfilled, even if other forks are still available to them.

We created code implements a solution to the dining philosophers problem using an arbitrator (in this case,we use a semaphore) to prevent deadlock and ensure that each philosopher can only pick up both forks or none.

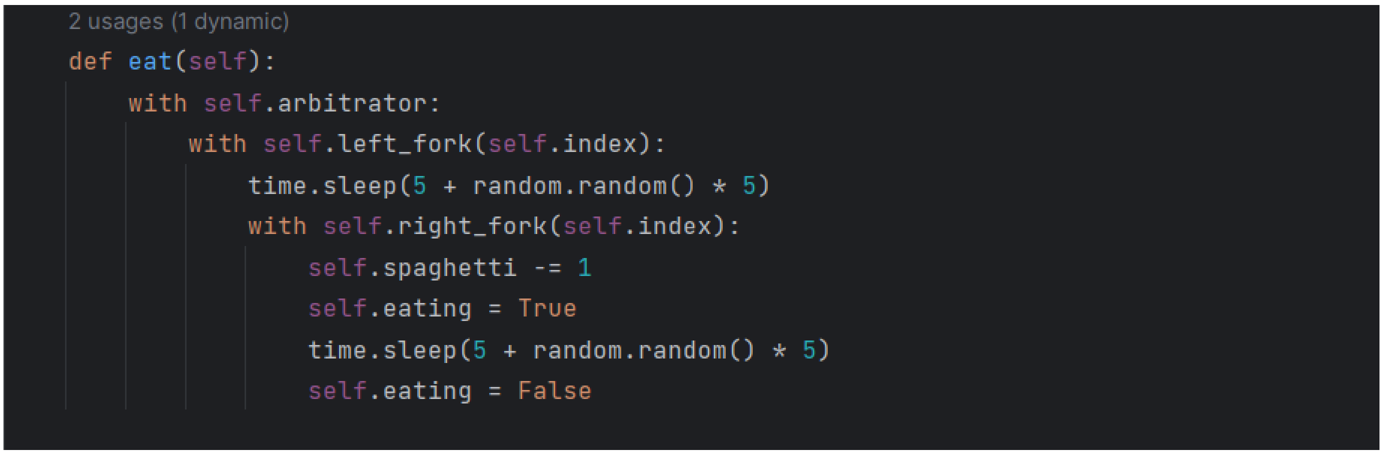
We implemented the Arbitrator solution strategy by making some additions to the code consisting of the Philosopher and Fork classes.

First, we added the arbitrator as a parameter to the Philosopher class. The arbitrator parameter here represents a semaphore that regulates the philosopher's entry into the critical region.



```
class Philosopher(threading.Thread):
    def __init__(self, index: int, arbitrator, left_fork: Fork, right_fork: Fork, spaghetti: int):
        super().__init__()
        self.index: int = index
        self.left_fork: Fork = left_fork
        self.right_fork: Fork = right_fork
        self.spaghetti: int = spaghetti
        self.eating: bool = False
        self.arbitrator = arbitrator
```

The Philosopher class has a method called eat where the philosopher requests permission from the arbitrator (waiter) before picking up both forks. This ensures that only one philosopher can pick up both forks at a time.



```
2 usages (1 dynamic)
def eat(self):
    with self.arbitrator:
        with self.left_fork(self.index):
            time.sleep(5 + random.random() * 5)
            with self.right_fork(self.index):
                self.spaghetti -= 1
                self.eating = True
                time.sleep(5 + random.random() * 5)
                self.eating = False
```

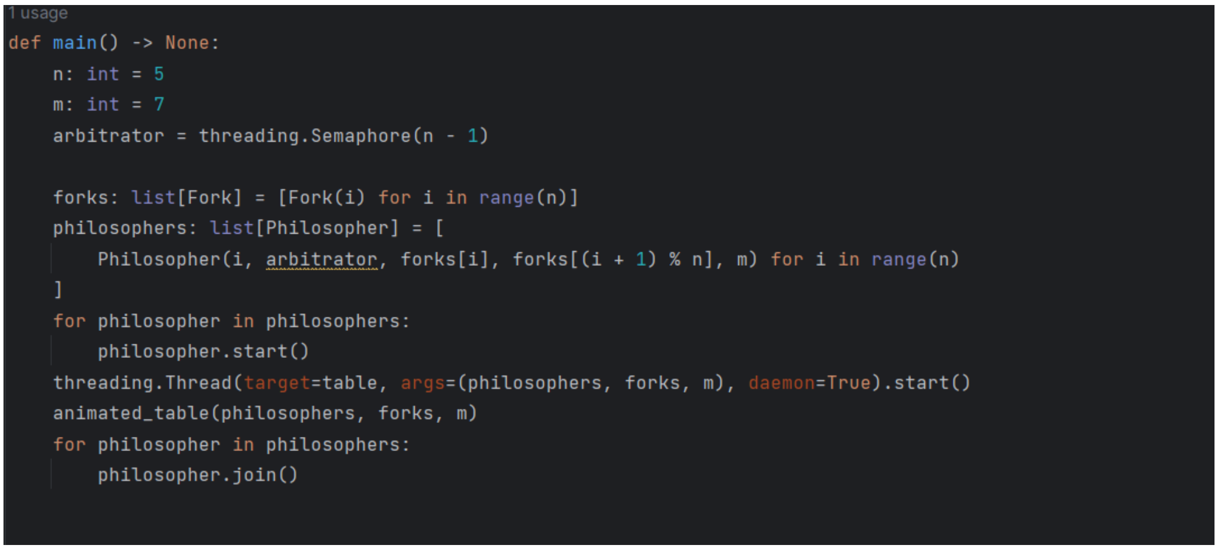
The **“with self.arbitrator:”** part specifies the use of the Semaphore named arbitrator, and this use occurs within the with statement. The with statement is a Python feature that allows us to securely use a resource (in this case the arbitrator Semaphore) and automatically release that resource.

Semaphore allows the philosopher entering this block to lock the Semaphore to start eating. If the value of Semaphore is 0 (that is, another philosopher has already started eating), then that philosopher's transaction is put on hold. If the value is 1, the philosopher continues its operation and reduces the value of the Semaphore by 1.

When the with block finishes, the Semaphore is automatically released. This increases the Semaphore's value by one and allows another philosopher to begin dining.

The code running background the **with self.arbitrator** statement can be simply expressed as follows:

```
# Lock Semaphore when entering block
self.arbitrator.acquire()
try:
    # Semaphore locking operations
    # This part represents the code inside the with block
    pass
finally:
    # Release Semaphore when exiting block
    self.arbitrator.release()
```



```
1 usage
def main() -> None:
    n: int = 5
    m: int = 7
    arbitrator = threading.Semaphore(n - 1)

    forks: list[Fork] = [Fork(i) for i in range(n)]
    philosophers: list[Philosopher] = [
        Philosopher(i, arbitrator, forks[i], forks[(i + 1) % n], m) for i in range(n)
    ]
    for philosopher in philosophers:
        philosopher.start()
    threading.Thread(target=table, args=(philosophers, forks, m), daemon=True).start()
    animated_table(philosophers, forks, m)
    for philosopher in philosophers:
        philosopher.join()
```

The arbitrator variable created in the main method is a Semaphore object. Semaphore is a synchronization tool used to achieve synchronization in parallel programming. In this case, the Semaphore called arbitrator allows at most n - 1 philosophers to start eating at the same time. So, when a philosopher wants to start eating, he must first obtain permission using the arbitrator semaphore.

The arbitrator is implemented using a semaphore with a count initialized to n-1, where n is the number of philosophers. This count represents the number of philosophers that can concurrently pick up forks. When a philosopher wants to eat, it must acquire the semaphore, ensuring that only up to n - 1 philosophers can eat simultaneously. After finishing eating, the philosopher releases the semaphore, allowing other philosophers to pick up forks.

The code also includes visualization using Matplotlib to show the state of philosophers and forks over time, and a simple text-based representation of the dining table is printed continuously in the console, showing which philosophers are eating or thinking.

Overall, the provided code addresses the dining philosophers problem by introducing an arbitrator to control access to the forks and avoid deadlock.