Parallel Programming: Dining Philosophers Problem

Şaban Kerem Yeğin 190316005
Manisa Celal Bayar UniversityComputer Engineering Dept. 190316005@ogr.cbu.edu.tr

Tuğcan Turunçkapı 19036082
Manisa Celal Bayar UniversityComputer Engineering Dept. 190316082@ogr.cbu.edu.tr

Ogün Ak 190316047
Manisa Celal Bayar University
Computer Engineering Dept. 190316047@ogr.cbu.edu.tr

***Abstract*—This Python code implements the Dining Philosophers problem using multithreading and a semaphore to synchronize access to shared resources. The program provides both textual and animated visualizations, showcasing the dynamic interactions among philosophers and forks in a dining scenario.**

Keywords*—Dining Philosophers, Multithreading, Semaphore, Resource Contention, Synchronization, Thread Safety, Python, Matplotlib, Visualization, Concurrency.

I. Introduction (*Heading 1*)

The Dining Philosophers Problem is a well-known example in computer science that illustrates the complexities of synchronization and resource management in concurrent systems. In this article, we delve into the problem, its significance, and propose a solution that employs Python's threading and semaphores.

II. Problem Statement

The Dining Philosophers problem is a classic example in computer science that explores challenges related to resource sharing and synchronization in a concurrent environment. In this scenario, a finite number of philosophers sit around a dining table, engaging in two activities: thinking and eating. The philosophers share a set of forks placed between them, with each philosopher requiring two forks to eat.

The primary challenge in this problem arises from the potential for deadlock and resource contention. Deadlock can occur when each philosopher holds one fork and is waiting for another, leading to a standstill. Resource contention may arise when multiple philosophers attempt to access shared forks simultaneously, creating a race condition that needs careful handling to ensure fairness and prevent conflicts.

III. Implementatİon Overvİew:

Utilizing the threading module for parallel execution and semaphores for resource synchronization. The Fork and Philosopher classes are designed to model the dining table scenario, and their interactions are managed using locks and semaphores.

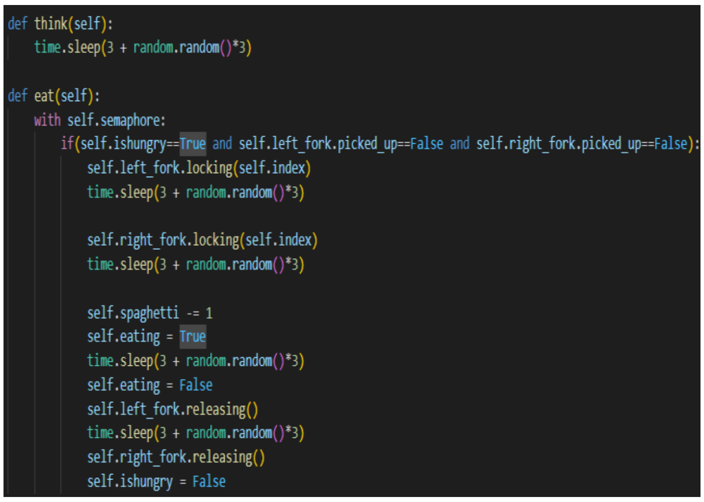
A. *Philosopher Class:*

The Philosopher class represents an individual philosopher and extends the threading.Thread class.

It includes methods for simulating thinking (think()) and eating (eat()), with random durations to emulate real-world variability.

The implementation introduces a state where philosophers are marked as 'hungry' and attempt to acquire the necessary forks for eating. The use of semaphores ensures that only a subset of philosophers can acquire forks simultaneously, preventing conflicts.

The philosopher interacts with shared resources using the left_fork and right_fork instances and employs a semaphore (semaphore) to control access to the dining table.

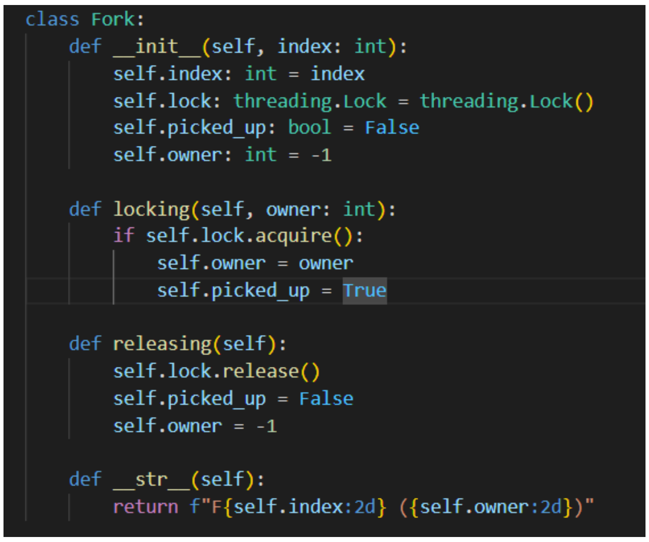


```python
def think(self):
    time.sleep(3 + random.random()*3)

def eat(self):
    with self.semaphore:
        if(self.ishungry==True and self.left_fork.picked_up==False and self.right_fork.picked_up==False):
            self.left_fork.locking(self.index)
            time.sleep(3 + random.random()*3)

            self.right_fork.locking(self.index)
            time.sleep(3 + random.random()*3)

            self.spaghetti -= 1
            self.eating = True
            time.sleep(3 + random.random()*3)
            self.eating = False
            self.left_fork.releasing()
            time.sleep(3 + random.random()*3)
            self.right_fork.releasing()
            self.ishungry = False
```

B. *Fork Class:*

The Fork class encapsulates the behavior of a fork, providing methods for locking a fork (locking()) and releasing a for (release()).

A threading.Lock ensures thread safety when philosophers attempt to pick up or put down forks, preventing conflicts and ensuring exclusive access to each fork.



```python
class Fork:
    def __init__(self, index: int):
        self.index: int = index
        self.lock: threading.Lock = threading.Lock()
        self.picked_up: bool = False
        self.owner: int = -1

    def locking(self, owner: int):
        if self.lock.acquire():
            self.owner = owner
            self.picked_up = True

    def releasing(self):
        self.lock.release()
        self.picked_up = False
        self.owner = -1

    def __str__(self):
        return f"F{self.index:2d} ({self.owner:2d})"
```

The code establishes a dining scenario with a predefined number of philosophers and forks, initializing them as threads and allowing them to execute concurrently. The use of semaphores and locks facilitates proper synchronization, preventing potential issues such as deadlocks or race conditions.

This structured implementation serves as a foundation for exploring the intricacies of concurrent programming and provides a tangible solution to the Dining Philosophers problem. The code elegantly balances the simulation of individual philosopher behaviors with the coordination of shared resources, offering insights into effective thread management and synchronization techniques.

IV. THREADİNG AND SEMAPHORES:

Multithreading is a fundamental aspect of the provided code, leveraging the threading module in Python to model individual philosophers as threads. This concurrent execution allows multiple philosophers to simulate thinking and eating simultaneously, creating a realistic dining scenario. The Philosopher class extends the threading.Thread class, enabling each philosopher to operate independently within the broader context of the dining table.

To prevent resource contention and ensure orderly access to shared forks, a semaphore is employed. The semaphore (semaphore) is initialized with a value equal to the total number of philosophers minus one (n - 1). This ensures that, at any given time, only a subset of philosophers (up to n - 1) can simultaneously attempt to pick up forks. The semaphore acts as a gatekeeper, allowing controlled access to the shared resources and preventing scenarios where all philosophers seek forks simultaneously, which could lead to deadlock.

The use of a semaphore effectively manages the concurrency of the dining philosophers, promoting a balance between parallel execution and controlled access to critical sections of the code. By carefully coordinating the actions of philosophers through the semaphore, the code demonstrates a key aspect of concurrent programming — the necessity of synchronization mechanisms to mitigate resource conflicts and enhance the overall stability and efficiency of the system.

V. CONCLUSİON

The presented solution effectively addresses the Dining Philosophers Problem by employing threading and semaphores in Python. Through this implementation, we have demonstrated how to mitigate concurrency issues, avoid deadlocks, and ensure the efficient utilization of resources in a scenario where multiple processes contend for shared items.

In conclusion, this article contributes to the understanding of concurrent programming challenges and provides a practical solution to a classic synchronization problem. The use of threading and semaphores in Python serves as a valuable example for developers grappling with similar issues in concurrent systems.