Parallel Programming: Dining Philosophers Problem

Efecan Erdem
190316044
Manisa Celal Bayar University
Computer Engineering Dept.
190316044@ogr.cbu.edu.tr

Selahattin Deniz Döktür
190316010
Manisa Celal Bayar University
Computer Engineering Dept.
190316010@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 classic synchronization and concurrency challenge that illustrates the complexities of resource sharing in a multithreaded environment. In this scenario, a group of philosophers sits around a dining table, each thinking and eating. The philosophers share common resources, represented by forks, and must avoid potential deadlocks and race conditions.

The significance of the Dining Philosophers problem lies in its ability to highlight fundamental issues in concurrent programming, such as ensuring thread safety, preventing resource contention, and achieving synchronization. As software systems increasingly leverage parallel processing and multithreading for improved performance, understanding and addressing these challenges become crucial for developing robust and efficient applications. This problem serves as a practical and theoretical foundation for exploring solutions to concurrency issues in various real-world scenarios.

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.

Effectively addressing these challenges is essential for designing concurrent systems that are both efficient and robust, making the Dining Philosophers problem a valuable case study in the field of parallel and distributed computing.

III. Implementatİon Overvİew:

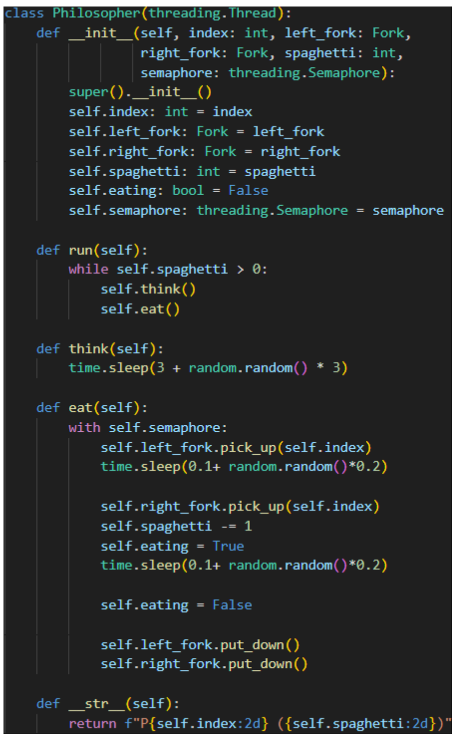
The Python code provides a practical implementation of the Dining Philosophers problem, addressing the complexities of concurrent programming and resource sharing. The code is structured around two main classes: Philosopher and Fork, each designed to encapsulate specific behaviors and ensure thread safety.

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 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.

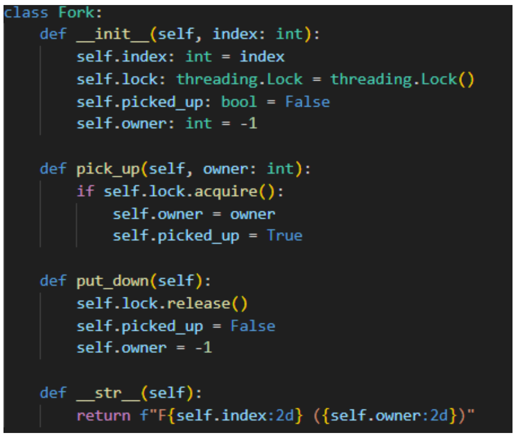


```
class Philosopher(threading.Thread):
    def __init__(self, index: int, left_fork: Fork,
                 right_fork: Fork, spaghetti: int,
                 semaphore: threading.Semaphore):
        super().__init__()
        self.index: int = index
        self.left_fork: Fork = left_fork
      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

    def run(self):
        while self.spaghetti > 0:
            self.think()
            self.eat()

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

    def eat(self):
        with self.semaphore:
            self.left_fork.pick_up(self.index)
            time.sleep(0.1+ random.random()*0.2)

            self.right_fork.pick_up(self.index)
            self.spaghetti -= 1
            self.eating = True
            time.sleep(0.1+ random.random()*0.2)

            self.eating = False

            self.left_fork.put_down()
            self.right_fork.put_down()

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

B. *Fork Class:*

The Fork class encapsulates the behavior of a fork, providing methods for picking up (pick_up()) and putting down (put_down()).

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

    def put_down(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

In conclusion, the provided code offers a practical and insightful solution to the Dining Philosophers problem, showcasing key concepts in concurrent programming. The implementation successfully models the complexities of shared resource management, synchronization, and parallel execution.

The code serves as a valuable educational tool, illustrating fundamental concepts in concurrent programming. It demonstrates the challenges of shared resource access and the importance of synchronization mechanisms in ensuring a coordinated and stable execution environment. Furthermore, the inclusion of visualizations enhances the code's pedagogical value, offering a tangible representation of concurrent scenarios and aiding in the comprehension of complex interactions.

Overall, this implementation contributes to a deeper understanding of concurrent programming principles, providing a hands-on exploration of synchronization, thread safety, and shared resource management through a well-crafted solution to the Dining Philosophers problem.