T.C.

MANİSA CELAL BAYAR UNIVERSITY

ENGINEERING FACULITY

DEPARTMENT OF COMPUTER ENGINEERING

Parallel Programming Homework Report

Dining Philosophers

190315080 - Ceyhun Binal

190315083 - Serkan Karahan

1. INTRODUCTION

The dining philosopher's problem is a classic synchronization and concurrency problem that illustrates the challenges of resource sharing and avoiding deadlocks in a multi-process or multi-threaded environment. The problem is often used to discuss issues related to process synchronization and the proper use of locks.

In this problem, several philosophers sit around a dining table. Each philosopher alternates between thinking and eating. To eat, a philosopher needs two forks, one on the left and one on the right. The challenge is to design a solution to ensure that the philosophers can dine without deadlock (where no philosopher can make progress) and without starvation (where a philosopher is unable to acquire both forks indefinitely).

The main issues to address are:

- **Deadlock:** Philosophers may reach a state where each holds one fork and waits indefinitely for the other fork.
- **Starvation:** A philosopher might be prevented from eating indefinitely, even if there is no deadlock.

Solving this problem involves careful coordination to avoid both deadlock and starvation. Various algorithms, such as using mutexes or semaphores, can be employed to achieve a proper solution to the dining philosopher's problem.

2. SOLUTION "MAXIMUM NUMBER OF PHILOSOPHERS"

We use the "Maximum Number of Philosophers" technique for solution this problem. It is a mechanism used to control the number of philosophers allowed to eat simultaneously in the Dining Philosophers problem. In the provided code, this is implemented using a threading.Semaphore named maximum_number_of_philosopher.

Here's how the technique works:

2.1. Semaphore Initialization

- maximum_number_of_philosopher = threading.Semaphore(2) is used to create a semaphore with an initial value of 2.
- The semaphore essentially acts as a counter that restricts the number of threads (philosophers) that can concurrently acquire it.

2.2. Semaphore Usage in Philosopher Class

- The Philosopher class constructor takes this semaphore as one of its parameters.
- The eat method uses the semaphore in combination with the with statement: with self.semaphore:. This ensures that only a limited number of philosophers can enter the critical section at the same time.
- By acquiring the semaphore, a philosopher signals that it is going to eat. If two philosophers try to acquire the semaphore simultaneously, only one will succeed, and the other will be blocked until the semaphore is released.

2.3. Limiting the Number of Philosophers Eating Simultaneously

- The value passed to the semaphore during initialization (2 in this case) determines the maximum number of philosophers allowed to eat concurrently.
- If the semaphore has a value of 2, then up to two philosophers can simultaneously execute the critical section protected by the semaphore.

2.4. Ensuring Fairness

- Semaphores provide a mechanism for enforcing access control to a shared resource. In this case, the shared resource is the critical section where philosophers pick up both forks to eat.
- The semaphore helps prevent race conditions and ensures that philosophers take turns to access the critical section, avoiding conflicts.

In summary, the "Maximum Number of Philosophers" technique, implemented using a semaphore, is a concurrency control mechanism to regulate the number of philosophers allowed to eat concurrently, helping to prevent conflicts and ensure orderly access to shared resources.

3. PYTHON CODE

```
import threading
import random
import time
import math
import matplotlib.pyplot as plt
import matplotlib.animation as animation

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 __enter__(self):
        return self

    def __call__(self, owner: int):
        if self.lock.acquire():
            self.owner = owner
            self.picked_up = True
            print(f"{self} is picked up by Philosopher {owner}")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        self.lock.release()
        self.picked_up = False
        self.owner = -1
        print(f"{self} is released.")

    def __str__(self):
        return f"F{self.index:2d} ({'P' + str(self.owner) if self.picked_up
else ' '})"


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):
        print(f"{self} is thinking.")
        time.sleep(3 + random.random() * 3)
        print(f"{self} has finished thinking.")

    def eat(self):
        with self.semaphore:
            with self.left_fork(self.index), self.right_fork(self.index):
                self.spaghetti -= 1
                self.eating = True
                print(f"{self} is eating.")
                time.sleep(5 + random.random() * 5)
                print(f"{self} has finished eating.")
                self.eating = False

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


def animated_table(philosophers: list[Philosopher], m: int):
    fig, ax = plt.subplots()
    ax.set_xlim(-1, 1)
    ax.set_ylim(-1, 1)
    ax.set_aspect("equal")
    ax.axis("off")
    ax.set_title("Dining Philosophers")

    philosopher_circles: list[plt.Circle] = [
        plt.Circle((0.5 * math.cos(2 * math.pi * i / len(philosophers)), 0.5 *
math.sin(2 * math.pi * i / len(philosophers))), 0.2, color="black") for i in
range(len(philosophers))
    ]

    philosopher_names: list[plt.Text] = [
        plt.text(
            0.5 * math.cos(2 * math.pi * i / len(philosophers)),
            0.5 * math.sin(2 * math.pi * i / len(philosophers)),
            f"{philosophers[i].index}",
```

```
            ha="center",
            va="center",
            color="white"
        ) for i in range(len(philosophers))
    ]

    for philosopher_circle in philosopher_circles:
        ax.add_patch(philosopher_circle)



    def update(frame):
        nonlocal philosophers
        for i in range(len(philosophers)):
            philosopher_circles[i].center = (
                0.5 * math.cos(2 * math.pi * i / len(philosophers)),
                0.5 * math.sin(2 * math.pi * i / len(philosophers)),
            )

            if philosophers[i].eating:
                philosopher_circles[i].set_edgecolor("red")
                philosopher_circles[i].set_facecolor("red")
            else:
                philosopher_circles[i].set_edgecolor("black")
                philosopher_circles[i].set_facecolor("black")

            philosopher_circles[i].radius = 0.2 * philosophers[i].spaghetti /
m

            philosopher_names[i].set_position((0.5 * math.cos(2 * math.pi * i
/ len(philosophers)), 0.5 * math.sin(2 * math.pi * i / len(philosophers))))

            if sum(philosopher.spaghetti for philosopher in philosophers) ==
0:
                ani.event_source.stop()
                exit(0)

        return philosopher_circles + philosopher_names

    ani = animation.FuncAnimation(
        fig, update, frames=range(100000), interval=10, blit=False
    )
    plt.show()


    philosopher_circles: list[plt.Circle] = [
```

```
        plt.Circle((0.5 * math.cos(2 * math.pi * i / len(philosophers)), 0.5 *
math.sin(2 * math.pi * i / len(philosophers))), 0.2, color="black") for i in
range(len(philosophers))
    ]

    for philosopher_circle in philosopher_circles:
        ax.add_patch(philosopher_circle)


def main() -> None:
    n: int = 5
    m: int = 5
    forks: list[Fork] = [Fork(i) for i in range(n)]

    maximum_number_of_philosopher = threading.Semaphore(2)

    philosophers: list[Philosopher] = [
        Philosopher(i, forks[i], forks[(i + 1) % n], m,
maximum_number_of_philosopher) for i in range(n)
    ]

    for philosopher in philosophers:
        philosopher.start()

    threading.Thread( args=(philosophers, m), daemon=True).start()
    animated_table(philosophers, m)

    for philosopher in philosophers:
        philosopher.join()

if __name__ == "__main__":
    main()
```

The Python code is an implementation of the Dining Philosophers problem using threading and animation in Python. Here's a brief explanation of the approach:

3.1. Fork Class

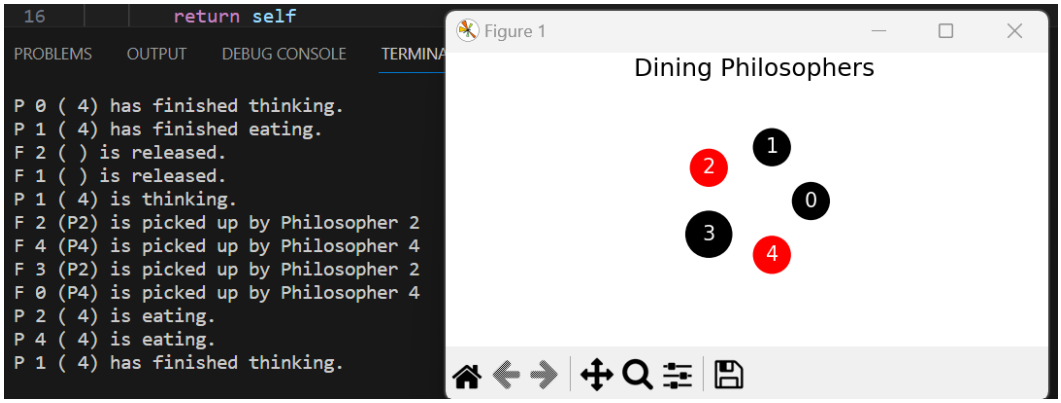
- The Fork class represents the forks on the dining table.
- Each fork is associated with a lock (threading.Lock) to handle mutual exclusion when a philosopher tries to pick it up.
- The __enter__ and __exit__ methods are implemented to provide a convenient way to acquire and release the fork using the with statement.

3.2. Philosopher Class

- The Philosopher class represents a philosopher sitting at the dining table.
- Each philosopher has a left and right fork, spaghetti count (indicating the number of times they can eat), and a semaphore (threading.Semaphore) to limit the number of philosophers allowed to eat simultaneously.
- The think and eat methods simulate the philosopher's actions, with random durations for thinking and eating.
- The philosopher uses the semaphore to control access to the critical section where they pick up both forks to eat.

3.3. Animated Table Function

- The animated_table function sets up a visual representation of the dining table using matplotlib.
- It creates circles for each philosopher and updates their positions, colors, and sizes based on their eating status and spaghetti count.
- The function uses an animation to continuously update the visual representation.



3.4. Main Function

- In the main function, the number of philosophers (n) and the maximum spaghetti count (m) are defined.
- Forks are created, and a semaphore (maximum_number_of_philosopher) is used to limit the number of philosophers allowed to eat simultaneously.
- Philosopher threads are created, each associated with left and right forks, spaghetti

count, and the semaphore.

- The philosophers are started, an animated table thread is started, and the main thread waits for the philosophers to finish.

The code aims to visually represent the dining philosophers scenario and uses semaphores to control access to the critical section, preventing conflicts between philosophers trying to pick up forks. The animated table provides a graphical representation of the philosophers' actions, helping to illustrate the synchronization of their activities.