Dining Philosophers

Strategy: Maximum Number of Philosophers

STUDENTS

CANBERK AKAR – 190316080

UMUTCAN TÜRK – 190316026

MURAT MERMER – 190316028

INTRODUCTION

The "Strategy: Maximum Number of Philosophers" focuses on maximizing the number of philosophers simultaneously present at the table. This strategy aims to have philosophers coexist at the table and perform their thinking and eating actions concurrently with the highest possible number.

CODE DETAILS

```
class Fork:
    def __init__(self, index):
        self.index = index
        self.lock = threading.Lock()
    def __enter__(self):
        if self.lock.acquire():
            return self
    def __exit__(self, exc_type, exc_value, traceback):
        self.lock.release()
```

The `Fork` class represents fork objects.

The `__enter__` method attempts to unlock a fork, and if successful, it returns the fork object.

The `__exit__` method releases the lock of a fork.

```
class Philosopher(threading.Thread):
    def __init__(self, index, left_fork, right_fork, spaghetti_quantity,
eating_sem):
        super().__init__()
        self.index = index
        self.left_fork = left_fork
        self.right_fork = right_fork
        self.spaghetti_quantity = spaghetti_quantity
        self.eating_sem = eating_sem
        self.eating = False

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

    def think(self):
        print(f"Philosopher {self.index} is thinking.")
```

```
        time.sleep(0 + random.random() * 3)

    def eat(self):
        print(f"Philosopher {self.index} is hungry and trying to pick up forks.")

        # Sol çatalı kilitle
        with self.eating_sem:
            with self.left_fork:
                print(f"Philosopher {self.index} picked up left fork.")
                time.sleep(2+ random.random()*5)
                # Sağ çatalı kilitle
                with self.right_fork:
                    print(f"Philosopher {self.index} picked up right fork and is
eating.")
                    self.spaghetti_quantity-=1
                    self.eating = True
                    time.sleep(0 + random.random() * 5)
                    self.eating = False
                    print(f"Philosopher {self.index} finished eating.")
```

The Philosopher class is a subclass of the Thread class, representing philosopher objects.

It has attributes such as index (philosopher's identifier), left_fork and right_fork (forks on the left and right of the philosopher), spaghetti_quantity (the amount of spaghetti the philosopher has), eating_sem (a semaphore to control access to the eating process), and eating (a boolean indicating whether the philosopher is currently eating).

The run method is the main method that gets executed when the thread is started. It contains a loop that continues until the philosopher has no spaghetti left.

Inside the loop, the philosopher alternates between thinking (think method) and eating (eat method).

The think method simulates the philosopher's thinking process by printing a message and sleeping for a random amount of time.

The eat method simulates the philosopher's eating process.

It uses the eating_sem semaphore to ensure that only a limited number of philosophers can attempt to eat at the same time.

The philosopher locks the left fork, then the right fork, and proceeds to eat if both forks are successfully locked.

After eating for a random amount of time, the philosopher finishes eating and releases the forks.

```
def animated_table(philosophers, forks, n,spaghetti_quantity):
    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 = [plt.Circle((0, 0), 0.2, color="black") for _ in
range(n)]
    philosopher_texts = [plt.Text(0, 0, str(philosopher.index), ha="center",
va="center") for philosopher in philosophers]

    for philosopher_circle in philosopher_circles:
        ax.add_patch(philosopher_circle)
    for philosopher_text in philosopher_texts:
        ax.add_artist(philosopher_text)

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

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

            #print(philosophers[i].spaghetti_quantity)
            philosopher_circles[i].radius = 0.2*
philosophers[i].spaghetti_quantity/spaghetti_quantity
        return philosopher_circles + philosopher_texts

    ani = animation.FuncAnimation(fig, update, frames=range(100000), interval=100,
blit=False)
    plt.show()
```

fig and ax create a Matplotlib figure and axis. The axis limits are set to (-1, 1), ensuring an equal aspect ratio. The visibility of the axis is turned off (ax.axis("off")), and the title is set to "Dining Philosophers.

Philosopher_circles creates a list of circles, each centered at (0, 0) with a radius of 0.2 and black color. One circle for each philosopher. philosopher_texts creates a list of text objects displaying the philosopher's index.

The `update` function is called for each frame of the animation. It updates the positions of philosopher circles based on their index, creating a circular arrangement. If a philosopher is currently eating (`eating` is True), the circle is colored red; otherwise, it is black. The radius of each circle is adjusted based on the remaining spaghetti quantity of the corresponding philosopher.

`animation.FuncAnimation` creates the animation by repeatedly calling the `update` function. The animation has a maximum of 100,000 frames and an interval of 100 milliseconds between frames.Finally, the animation is displayed using `plt.show()`.

```
def dining_philosophers(n, spaghetti_quantity):
    forks = [Fork(i) for i in range(n)]
    eating_sem = threading.Semaphore(n - 2)
    philosophers = [Philosopher(i, forks[i], forks[(i + 1) % n],
spaghetti_quantity, eating_sem) for i in range(n)]

    for philosopher in philosophers:
        philosopher.start()

    animated_table(philosophers, forks, n,spaghetti_quantity)

    for philosopher in philosophers:
        philosopher.join()

if __name__ == "__main__":
    n = int(input("Enter the number of philosophers: "))
    spaghetti_quantity = int(input("Enter quantity of spaghetti:(Each eat reduces 1
spaghetti) "))

    dining_philosophers(n, spaghetti_quantity)
```

The `dining_philosophers` function sets up and runs the dining philosophers simulation.It creates a list of forks (`forks`) and a semaphore (`eating_sem`) to control access to the eating process, allowing a maximum of `n - 2` philosophers to eat simultaneously.Each philosopher is instantiated with a unique index, left and right forks, the initial spaghetti quantity, and the semaphore.The function then starts each philosopher as a separate thread, initiates the animated table display (`animated_table`), and waits for all philosophers to finish using `join`.

The main block of the script prompts the user to enter the number of philosophers (`n`) and the initial quantity of spaghetti (`spaghetti_quantity`).It then calls the `dining_philosophers` function with the provided inputs, initiating the dining philosophers simulation.

Challenges Faced While Implementing the Code

- **Understanding and Applying the Strategy:**

  Challenge: Grasping the intricacies of the Dining Philosophers problem and implementing a strategy to maximize the number of philosophers eating simultaneously.

  Solution: Delving into the dynamics of philosopher interactions and determining a suitable strategy required careful consideration.

- **Animating Philosopher Interactions:**

  Challenge: Introducing animations using Matplotlib to visually represent philosopher positions, thinking, and eating actions.

  Solution: Overcoming the learning curve of Matplotlib's animation module and ensuring accurate depiction of philosopher states during the simulation.

- **Coordinating Threads and Animation:**

Challenge: Coordinating the execution of philosopher threads and integrating the animation seamlessly to provide a comprehensive visual representation.

Solution: Ensuring synchronization between thread execution and animation updates to depict a coherent simulation of dining philosophers.

REFERANCES

- https://www.geeksforgeeks.org/dining-philosopher-problem-using-semaphores/
- https://web.eecs.utk.edu/~jplank/plank/classes/cs360/360/notes/Dphil/lecture.html
- https://www.youtube.com/watch?v=3tl2YEYbaKk&list=PL30NBs02RsiUbmXVPDo56APsU0xa6gfL2