CELAL BAYAR
UNIVERSITY

Dining Philosophers

Strategy: Maximum Number of Philosophers

STUDENTS

Eyyup ÇOKER – 190316080

Eray AĞARER - 190316084

Ali Mert YILMAZ – 190316037

Fudayl ÇAVUŞ - 200316042

INTRODUCTION

The approach known as the "Maximum Number of Philosophers Strategy" concentrates on optimizing the concurrent presence of philosophers at the table. This method strives to maximize the coexistence of philosophers, enabling them to concurrently engage in both thinking and eating actions, aiming for the highest possible number of active participants at the table.

CODE DETAILS

```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 __enter__(self):
        return self

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

    def __exit__(self, exc_type, exc_value, traceback):
        self.lock.release()
        self.picked_up = False
        self.owner = -1

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

__init__(self, index: int) : The initializer for the Fork class.

__enter__(self) : This method is called when an object is placed within a with statement. In this case, it returns the object itself.

__call__(self, owner: int) : This method is invoked to signify the use of the fork by a philosopher.

__exit__(self, exc_type, exc_value, traceback) : This method is called when a with statement ends.

__str__(self) : This method returns a string representation of the fork object, including its index and owner.

```python
class Philosopher(threading.Thread):
    def __init__(self, index: int, 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

    def use_forks(fn, *args, **kwargs):
        def wrapper(self):
            with self.left_fork(self.index):
                time.sleep(2 + random.random() * 5)
                with self.right_fork(self.index):
                    fn(self, *args, **kwargs)

        return wrapper

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

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

    @use_seat
    @use_forks
    def eat(self):
        self.spaghetti -= 1
        self.eating = True
        time.sleep(random.random() * 5)
        self.eating = False

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

(__init__): Initializes a philosopher with an index , references to their left and right forks (left_fork and right_fork), a quantity of spaghetti (spaghetti), and an initial state of not eating (eating = False).

use_forks : A decorator that handles the use of forks (mutex) when a philosopher eats. It ensures that philosophers pick up both the left and right forks before eating.

run: Simulates the philosopher's activity while there is spaghetti available. It alternates between thinking and eating.

think: Simulates the philosopher thinking by sleeping for a random duration.

eat: Decorated with @use_seat and @use_forks decorators. Decreases the spaghetti count, simulates eating for a random duration, and sets the philosopher's state to not eating after finishing.

__str__: Returns a string representation of the philosopher including their index and remaining spaghetti.

```python
def animated_table(philosophers: list[Philosopher], forks: list[Fork], m: int):
    """
    Creates an animated table with the philosophers and forks.

    :param philosophers: The list of philosophers.
    :param forks: The list of forks.
    :param m: The amount of spaghetti each philosopher has.
    """
    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, 0), 0.2, color="black") for _ in range(len(philosophers))
    ]
    philosopher_texts: list[plt.Text] = [
        plt.Text(
            0,
            0,
            str(philosopher.index),
            horizontalalignment="center",
            verticalalignment="center",
        )
        for philosopher in philosophers
    ]
    fork_lines: list[plt.Line2D] = [
        plt.Line2D((0, 0), (0, 0), color="black") for _ in range(len(forks))
    ]
    fork_texts: list[plt.Text] = [
        plt.Text(
            0,
            0,
            str(fork.index),
            horizontalalignment="center",
            verticalalignment="center",
        )
        for fork in forks
    ]
    for philosopher_circle in philosopher_circles:
        ax.add_patch(philosopher_circle)
    for fork_line in fork_lines:
        ax.add_line(fork_line)
    for philosopher_text in philosopher_texts:
        ax.add_artist(philosopher_text)
    for fork_text in fork_texts:
        ax.add_artist(fork_text)
```

def animated_table:

Parameters:

philosophers : A list of Philosopher objects.

forks : A list of Fork objects.

m : The quantity of spaghetti each philosopher has.

fig, ax = plt.subplots() : Creates a new figure (fig) and axes (ax) using Matplotlib.

ax.set_xlim(-1, 1) and ax.set_ylim(-1, 1) : Sets the x and y-axis limits of the plot to -1 to 1, creating a square plot.

ax.set_aspect("equal") : Sets the aspect ratio of the plot to be equal, ensuring the circles appear as circles.

ax.axis("off") : Hides the axis lines and labels.

ax.set_title("Dining Philosophers") : Sets the title of the plot to "Dining Philosophers".

philosopher_circles : List comprehension creating circles (using plt.Circle) for each philosopher with a radius of 0.2 and color black.

philosopher_texts : List comprehension creating text labels for each philosopher's index, centered within each circle.

fork_lines : List comprehension creating line segments (using plt.Line2D) to represent forks, initially set to (0,0) coordinates

fork_texts : List comprehension creating text labels for each fork's index, centered at (0,0)

```
def update(frame):
    """
    Updates the table.
    """
    # get the philosophers and forks from the global scope
    nonlocal philosophers, forks
    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)),
        )
        # update the labels as text on the plot
        philosopher_texts[i].set_position(
            (
                0.9 * math.cos(2 * math.pi * i / len(philosophers)),
                0.9 * math.sin(2 * math.pi * i / len(philosophers)),
            )
        )
        philosopher_texts[i].set_text(
            str(philosophers[i]) if philosophers[i].spaghetti > 0 else "X"
        )
        if philosophers[i].eating:
            philosopher_circles[i].set_color("red")
        else:
            philosopher_circles[i].set_color("black")
        philosopher_circles[i].radius = 0.2 * philosophers[i].spaghetti / m
        fork_lines[i].set_data(
            (
                0.5 * math.cos(2 * math.pi * i / len(philosophers)),
                0.5 * math.cos(2 * math.pi * (i + 1) / len(philosophers)),
            ),
            (
                0.5 * math.sin(2 * math.pi * i / len(philosophers)),
                0.5 * math.sin(2 * math.pi * (i + 1) / len(philosophers)),
            ),
        )
        # add the labels of the forks as text on the plot
        fork_texts[i].set_position(
            (
                0.5 * math.cos(2 * math.pi * i / len(philosophers))
                + 0.5 * math.cos(2 * math.pi * (i + 1) /
                                 len(philosophers)),
                0.5 * math.sin(2 * math.pi * i / len(philosophers))
                + 0.5 * math.sin(2 * math.pi * (i + 1) /
                                 len(philosophers)),
            )
        )
```

```
        fork_texts[i].set_text(str(forks[i]))
        if forks[i].picked_up:
            fork_lines[i].set_color("red")
        else:
            fork_lines[i].set_color("black")
    return philosopher_circles + fork_lines + philosopher_texts + fork_texts

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

def update(frame):

Retrieve Philosophers and Forks :

Accesses the philosophers and forks variables from the enclosing scope using nonlocal .

Update Philosopher Positions and Labels :

Loops through each philosopher:

Sets the position of the philosopher's circle using polar coordinates to arrange them evenly in a circle.

Updates the text labels of the philosophers based on their indices and remaining spaghetti. If a philosopher has no spaghetti left, their label is set to "X".

Changes the color of the philosopher's circle to red if they are currently eating (based on the eating attribute).

Adjusts the size (radius) of the philosopher's circle based on the remaining spaghetti compared to the total (m).

Update Fork Positions, Lines, and Labels :

Updates the positions and lines representing forks based on the positions of the philosophers.

Updates the text labels of forks based on their indices.

Changes the color of fork lines to red if they are currently picked up by a philosopher (based on the picked_up attribute).

Return Updated Elements :

Returns a list of all elements that need to be updated in each animation frame (philosopher circles, fork lines, philosopher texts, and fork texts).

fig : The figure object where the animation will be displayed.

update : The function that updates the plot elements in each frame.

frames=range(100000) : The number of frames the animation will run. In this case, it's set to a large number (100000) but can be adjusted.

interval=10 : The time interval between frames in milliseconds.

blit=False : Specifies whether to update only parts that have changed.

plt.show() : Displays the animation.

```python
def table(philosophers: list[Philosopher], forks: list[Fork], m: int):
    """
    Prints the table with the philosophers and forks.

    :param philosophers: The list of philosophers.
    :param forks: The list of forks.
    :param m: The amount of spaghetti each philosopher has.
    """
    while sum(philosopher.spaghetti for philosopher in philosophers) > 0:
        eating_philosophers: int = sum(
            philosopher.eating for philosopher in philosophers
        )
        # clear the screen
        print("\033[H\033[J")
        print("=" * (len(philosophers) * 16))
        # print a line for each philosopher if they are eating, thinking, or done
        print(
            "         ",
            "           ".join(
                ["E" if philosopher.eating else "T" for philosopher in philosophers]
            ),
        )
        print("       ".join(map(str, forks)), "     ", forks[0])
        print(
            "      ",
            "       ".join(map(str, philosophers)),
            "      : ",
            str(eating_philosophers),
        )
        print("Sum ", sum(philosopher.spaghetti for philosopher in philosophers))
        time.sleep(0.1)
```

def table:

philosophers : List of Philosopher objects representing the philosophers at the table.

forks : List of Fork objects representing the forks on the table.

m : Integer representing the amount of spaghetti each philosopher has.

Looping Until Spaghetti Runs Out : Uses a while loop that continues until the total amount of spaghetti among all philosophers reaches zero.

Displaying the Table : Clears the screen using ANSI escape codes (\033[H\033[J) to erase the terminal screen.

Prints a line for each philosopher indicating if they are eating ("E") or thinking ("T").

Prints the current state of the forks and their indices.

Displays the current state of each philosopher and their indices.

Shows the total count of remaining spaghetti among all philosophers.

Pause Between Updates :

Sleeps for a short duration (0.1 seconds) between updates to create a visual effect of updating the table at regular intervals.

Challenges Faced While Implementing the Code

Concurrency and Synchronization : Threads operating concurrently might lead to synchronization issues. Ensuring the proper usage of lock mechanisms in the Fork and Philosopher classes is crucial.

Graphical Interface and Animation Challenges : Creating a visual animation using matplotlib can pose challenges in proper display and synchronization, especially when multiple threads are involved.

Data Synchronization : Managing the state of philosophers and spaghetti quantities requires careful synchronization of this data. For instance, ensuring that when one philosopher decreases their spaghetti count, other threads accurately reflect this change.

Error Handling : Having a plan to handle unexpected errors that threads might encounter is essential. Determining how to handle crashes or synchronization errors, for instance, is crucial for the stability of the program.

REFERANCES

https://bilgisayarkavramlari.com/2012/01/22/filozoflarin-aksam-yemegi-dining-philosophers/

https://tr.wikipedia.org/wiki/Makarna_yiyen_d%C3%BC%C5%9F%C3%BCn%C3%BCrler_sorunu

https://serdarkuzucu.com/dining-philosophers-problem-cozumu-java-ornekli/