PARALLEL PROGRAMMING TERM PROJECT

Mehmet Osman Akgün 200316043
Utkay Güngör Battal 200316010

DINING PHILISOPHERS

Overview: The provided Python code presents a solution to the Dining Philosophers problem, a classic synchronization problem in computer science. This problem involves a set of philosophers seated at a table, each with a plate of spaghetti and a fork between every pair of plates. The challenge lies in allowing each philosopher to eat, which requires using both adjacent forks, without causing a deadlock.

Code Structure and Strategies Classes and Threading

Fork Class: Represents a fork at the table. It includes a lock for thread safety and methods for picking up and releasing forks. Each fork instance manages its ownership status, preventing multiple philosophers from using the same fork simultaneously.

Philosopher Class: Models a philosopher as a thread. Each philosopher has references to their left and right forks. They execute asynchronous actions of thinking and eating through think () and eat() methods. Philosophers use locks to acquire forks before eating, ensuring exclusive access to the forks they need.

Concurrency Management

Thread-Based Approach: Philosophers are represented as threads, enabling concurrent execution of their actions (thinking and eating). The use of threading allows each philosopher to perform actions independently.

Lock Mechanisms (threading.Lock): Utilized to ensure thread safety when philosophers attempt to pick up forks. The code implements a locking mechanism for each fork, preventing race conditions and ensuring exclusive access.

Resource Allocation Strategy:

Hierarchical Fork Access: Philosophers follow a specific order while attempting to pick up forks to avoid deadlocks. They acquire the left and right forks in a consistent order, ensuring a hierarchy to prevent situations where all philosophers are holding one fork and waiting for another.

Token Ring Solution:

Introduction of Token Ring: In addition to the hierarchical approach, a token ring solution can be introduced to control the access of philosophers to forks. The token ring can be implemented to pass a token, allowing only one philosopher to acquire both forks at a time. This token passing mechanism ensures fairness and prevents resource contention among philosophers.

Visualization (Matplotlib): animated_table() Function: Utilizes the Matplotlib library to create an animated visual representation of the Dining Philosophers problem. The function dynamically updates the state of philosophers and forks, displaying their actions visually. This visualization aids in understanding the concurrent nature of the problem and the synchronization achieved.

The provided code offers a robust solution to the Dining Philosophers problem, utilizing threading, locks, and a hierarchical approach for resource access. The introduction of a token ring solution further enhances the fairness in resource allocation, ensuring that only one philosopher holds both forks at a time. This combination of strategies effectively manages concurrent access to shared resources while preventing deadlocks and resource contention. Additionally, the visualization aspect adds clarity by illustrating the dynamic nature of the problem, making it easier to comprehend the concurrency management involved in the scenario.

Overall, the code provides a comprehensive resolution to the Dining Philosophers problem, incorporating both theoretical concurrency strategies and practical implementation techniques.

1. **class Fork:**

   Purpose: Represents a fork object.

   Methods:

   __init__(self, index: int): Initializes a Fork instance with a unique index, a lock for thread safety, and status indicators. __enter__(self), __exit__(self, exc_type, exc_value, traceback): Implement context management protocol to acquire and release the fork's lock safely. __call__(self, owner: int): Handles the picking up of the fork by a philosopher.

2. **class Philosopher:**

Purpose: Represents a philosopher thread.

Methods:

__init__(self, index: int, left_fork: Fork, right_fork: Fork, spaghetti: int, token: threading.Lock): Initializes a Philosopher instance with the necessary attributes. run(self): Overrides the run method from the Thread class, simulating the philosopher's actions of thinking and eating. think(self): Simulates the thinking process of a philosopher. eat(self): Handles the philosopher's eating process, acquiring the necessary forks using locks.

3. **def animated_table(philosophers: list[Philosopher], forks: list[Fork], m: int):**

Purpose: Generates an animated visualization of the dining philosophers scenario using Matplotlib. Functionality: Creates a dynamic visualization updating philosopher and fork states based on their actions (thinking, eating, fork pickup). Animates the positions and colors of circles and lines representing philosophers and forks.

4. **def update(frame):**

Purpose: Updates the table's visual representation during animation. Functionality: Updates the positions, colors, and sizes of circles (representing philosophers), lines (representing forks), and associated texts based on the current state.

5. **def table(philosophers: list[Philosopher], forks: list[Fork], m: int):**

Purpose: Prints the table with philosophers' status in the terminal. Functionality: Displays a table in the terminal with the status (eating/thinking) of each philosopher and fork.

6. **def main () -> None:**

Purpose: Main function to orchestrate the Dining Philosophers problem. Functionality: Initializes necessary objects (philosophers, forks, token). Starts philosopher threads and visualization threads. Controls the execution flow of the program by coordinating thread actions.

Each function contributes to simulating and visualizing the Dining Philosophers problem, ensuring synchronized access to shared resources (forks) and providing a visual representation of the scenario.

OUTPUT: