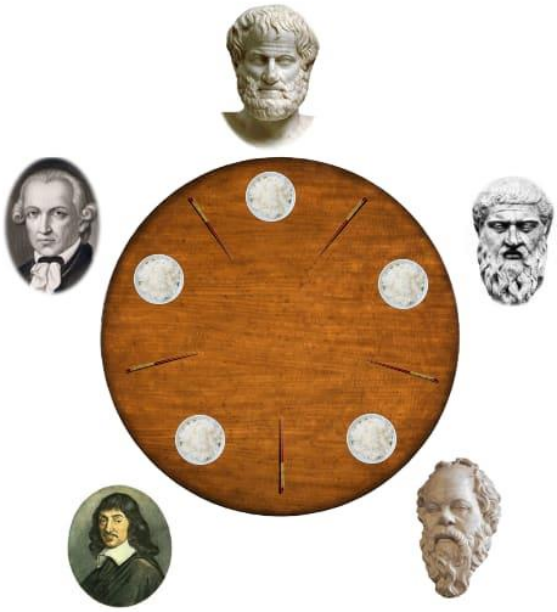

BUG BUSTERS

Parallel Programming Project

Resource Hierarchy Solution

The Resource Hierarchy Solution is based on establishing a strict order for acquiring resources (in this case, forks) and ensuring that philosophers follow this order consistently.

İrem Dilşat Köse

200315029

Atakan KARAKOÇ

200315088

Burak GÜLLÜLER

200315090

1. Philosophical Problem and Solution Strategy:

This program represents a classic example of the "Dining Philosophers Problem." In this scenario, five philosophers are seated around a table, with a single fork between each pair of philosophers, and a plate of spaghetti in the middle. The objective is to simulate the process of philosophers thinking and eating, while ensuring that each philosopher can only eat if they have both forks.

- **Problem Description:**
  - The challenge lies in preventing deadlock and ensuring that philosophers can make progress without conflicting for resources.
  - Philosophers must acquire two forks to eat, but they should release the forks promptly to allow others to use them.
- **Solution Strategy:**
  - The code uses a combination of threads and locks to represent philosophers and forks.
  - The Fork class manages the state of each fork, allowing philosophers to pick up and put down forks safely using locks.
  - Philosophers, represented by the Philosopher class, alternate between thinking and attempting to eat by acquiring two adjacent forks.
  - Locks are used to prevent conflicts between philosophers attempting to pick up the same fork simultaneously.
- **Concurrency and Synchronization:**
  - The program elegantly demonstrates the challenges of concurrent access to shared resources, emphasizing the importance of synchronization to avoid race conditions.
  - Each philosopher operates as an independent thread, and the locks associated with forks ensure that only one philosopher can pick up a fork at a time.
- **Realism through Randomization:**
  - The inclusion of random durations for thinking and eating adds a realistic touch to the simulation, introducing variability in the philosophers' behavior.
- **Philosopher and Fork Interaction:**
  - Philosophers use the think and eat methods to simulate periods of contemplation and eating, respectively.
  - Forks are represented by the Fork class, and the locking mechanism ensures exclusive access to a fork by a philosopher.

2. Fork Class:

The Fork class is responsible for managing the state of each fork and controlling its lock. This class serves as a fundamental building block of the program and encompasses the following key aspects:

- **Construction of the Fork:**
  - The class is structured to define a fork with a unique index, and through the lock property, it keeps track of the fork's state.
  - The __init__ method sets the initial state of the fork, creates a lock, and specifies the owner of the fork (initialized as -1).
- **Picking Up and Putting Down the Fork:**
  - The pickup and put_down methods govern the locking and unlocking operations of the fork.
- **Thread Safety with Locks:**
  - The use of the threading.Lock ensures thread safety when accessing and modifying the state of a fork.
  - The pickup method attempts to acquire the lock, sets the owner and picked-up status if successful, and returns a Boolean indicating the success of the operation.
  - The put_down method releases the lock, indicating that the fork is no longer in use.
- **String Representation:**
  - The __str__ method provides a string representation of the fork, including its index and the current owner.

3. Philosopher Class:

The Philosopher class represents each individual philosopher in the dining philosophers simulation. This class encapsulates the behavior and attributes associated with a philosopher, and its implementation involves several key components:

- **Initialization:**
  - The __init__ method initializes a philosopher with a unique index, left and right forks, and the initial amount of spaghetti.
  - The spaghetti attribute tracks the remaining portions of spaghetti available to the philosopher.
  - The eating attribute indicates whether the philosopher is currently in the process of eating.
- **Run Method:**
  - The run method, inherited from the threading.Thread class, defines the main behavior of the philosopher as they go through repeated cycles of thinking and eating.
  - The philosopher continues the cycle until there is no more spaghetti left.
- **Thinking and Eating:**
  - The think method simulates a period of thinking by introducing a sleep duration based on a random value.
  - The eat method attempts to pick up two forks (lower and higher), ensuring a consistent order to avoid deadlocks.
  - If successful in acquiring both forks, the philosopher simulates eating with a random duration.

- **More on eat method:**
  - The eat method within the Philosopher class is a crucial component responsible for simulating the philosopher's eating behavior. This method incorporates several key functionalities in alignment with the overall dining philosophers simulation:

Fork Selection:

- The method begins by determining the lower and higher forks based on their indices. This ensures a consistent order to prevent deadlocks. The lower fork is selected from the philosopher's left and right forks.

```
lower_fork, higher_fork = (
    (self.left_fork, self.right_fork)
    if self.left_fork.index < self.right_fork.index
    else (self.right_fork, self.left_fork)
)
```

Lower Fork Pickup:

- The philosopher attempts to pick up the lower fork using the pickup method of the Fork class. If successful, the process continues; otherwise, the method exits.

```
if lower_fork.pickup(self.index):
```

Higher Fork Pickup:

- Subsequently, the philosopher attempts to pick up the higher fork. If successful, the spaghetti portion is decreased, and the philosopher's eating flag is set to True.

```
if higher_fork.pickup(self.index):
    self.spaghetti -= 1
    self.eating = True
    time.sleep(5 + random.random() * 5)
```

Fork Put Down:

- After completing the eating process, the higher fork is put down, followed by putting down the lower fork.

```
    self.eating = False
    higher_fork.put_down()
lower_fork.put_down()
```

String Representation:

- The __str__ method provides a string representation of the philosopher, including their index and the remaining portions of spaghetti.
- **Concurrency and Synchronization:**
  - The use of threads allows multiple philosophers to run concurrently, and the synchronization mechanisms with forks ensure that conflicts are appropriately managed.
- **Dynamic Behavior through Randomization:**
  - The introduction of random durations in thinking and eating adds a dynamic and realistic element to the philosopher's behavior.

4. Main Function:

The main function serves as the entry point for the dining philosophers simulation. It orchestrates the creation of forks, philosophers, and the initiation of threads to simulate the dining scenario. Let's explore the key components of this function in more detail:

- **Initialization:**
  - The function begins by setting the number of philosophers (n) and the initial portions of spaghetti (m).
  - It then creates a list of Fork objects, assigning each fork a unique index.
- **Philosopher Initialization:**
  - The function proceeds to create a list of Philosopher objects, with each philosopher being associated with two adjacent forks.
  - The index of the philosopher and the corresponding forks is used to establish the connections.
- **Thread Start:**
  - A loop is employed to start the threads representing each philosopher, invoking the start method for each thread.
  - This initiates the independent execution of each philosopher's run method.
- **Thread Join:**
  - Finally, the main function waits for each philosopher thread to complete its execution using the join method.
  - This ensures that the program remains active until all philosopher threads have finished.
- **Overall Flow:**
  - The main function provides a clear orchestration of the simulation, managing the initialization, thread execution, and termination.

Resource Hierarchy Solution:

The Resource Hierarchy Solution is based on establishing a strict order for acquiring resources (in this case, forks) and ensuring that philosophers follow this order consistently. Here's why this approach was chosen:

Advantages:

- **Deadlock Avoidance:**
  - By assigning a unique index to each fork and having philosophers always pick up the lower-indexed fork before the higher-indexed one, a strict hierarchy is maintained.
  - This hierarchy eliminates the possibility of circular waiting, a condition that could lead to deadlocks.
- **Simplicity and Predictability:**
  - The solution is straightforward and easy to understand. Philosophers always follow a consistent order when picking up forks.
  - The deterministic nature of the solution makes it predictable, which aids in reasoning about the program's behavior.
- **Scalability:**
  - The approach scales well with the number of philosophers since the rules for acquiring forks are clear and can be extended to accommodate additional philosophers.
- **No Priority Inversion:**
  - The Resource Hierarchy Solution avoids priority inversion issues, where a low-priority task holds a resource needed by a high-priority task.

Disadvantages:

- **Potential for Starvation:**
  - If a philosopher is unable to acquire the lower-indexed fork due to constant contention, they might starve, unable to progress to the eating phase.
  - This could occur if the lower-indexed fork is continuously held by another philosopher.
- **Unequal Resource Utilization:**
  - The strict hierarchy may lead to suboptimal resource utilization, especially if philosophers are unable to pick up forks due to the chosen order.
- **Dependency on Fork Indexing:**
  - The solution relies heavily on the indexing of forks. If the indexing mechanism is disrupted or mismanaged, it could lead to unexpected behavior.

Comparison with Other Solutions:

- **Comparison with Arbitrator Solution:**
  - The Resource Hierarchy Solution is less centralized compared to an Arbitrator Solution, where a central authority (arbitrator) controls resource access.
  - It distributes decision-making among philosophers, avoiding a single point of contention.
- **Comparison with Chandy/Misra Solution:**
  - Unlike the Chandy/Misra Solution, which involves a complex system of token passing, the Resource Hierarchy Solution is simpler and more intuitive.
  - However, the Chandy/Misra Solution can potentially achieve higher parallelism.
- **Comparison with Wait-Die Solution:**
  - The Resource Hierarchy Solution differs from the Wait-Die Solution, which involves dynamic resource allocation and a transaction-based approach.
  - Resource Hierarchy is a static approach, and philosophers follow a predetermined order.

In summary, the Resource Hierarchy Solution was chosen for its simplicity, deadlock avoidance, and scalability. However, it comes with the trade-off of potential starvation and unequal resource utilization. The choice depends on the specific requirements and constraints of the system being modeled