Mamadou Diao Kaba (27070179)

Report

The Philosopher Class (Thread)

- **Methods**:
    - **public void eat()**: Simulates the philosopher eating. It prints a start message, induces a sleep for a random interval to simulate eating duration, and prints an end message.
    - **public void think()**: Simulates the philosopher thinking. Similar to **eat()**, it involves sleeping for a random duration to mimic thinking.
    - **public void talk()**: Allows the philosopher to engage in talking, selecting and printing a random philosophical statement.
    - **public void run()**: The main lifecycle of the philosopher. It dictates the philosopher's behavior, alternating between thinking, trying to eat (which involves synchronization with the **Monitor**), and optionally talking.
    - **private void saySomething()**: Prints a random philosophical statement from a predefined list, used within the **talk()** method.

The Monitor Class

- **Methods**:
    - **public synchronized void pickUp(final int piTID)**: Called by a philosopher when they want to eat. It checks the states of adjacent philosophers and either allows the calling philosopher to proceed with eating or forces them to wait.
    - **public synchronized void putDown(final int piTID)**: Called by a philosopher after eating. It changes the philosopher's state back to thinking and notifies other philosophers that might be waiting for chopsticks.
    - **public synchronized void requestTalk()**: Ensures that only one philosopher can talk at a time. If another philosopher is talking, the calling philosopher is put into a wait state.
    - **public synchronized void endTalk()**: Called when a philosopher finishes talking, allowing others the chance to talk.
    - Additional methods for managing pepper shakers (if implemented) would similarly synchronize access to this shared resource.

Flow of the Program

1. **Initialization**:

- The **DiningPhilosophers** main class initializes the simulation, taking an optional command-line argument to set the number of philosophers.
- It creates a **Monitor** instance, responsible for managing synchronization between philosophers.

2. **Philosopher Threads Start**:
    - For each philosopher, a **Philosopher** thread is created and started. Each thread represents a philosopher's lifecycle.
3. **Philosopher Lifecycle**:
    - **Thinking**: Each philosopher begins in a thinking state, simulating contemplation by sleeping for a random duration.
    - **Hungry (Attempting to Eat)**: After thinking, a philosopher becomes hungry and attempts to pick up chopsticks by invoking the **Monitor**'s **pickUp** method. If both adjacent philosophers are not eating, the philosopher proceeds; otherwise, they wait.
    - **Eating**: Once chopsticks are acquired, the philosopher eats (sleeps for a duration) and then puts down the chopsticks, signaling others by calling the **Monitor**'s **putDown** method.
    - **Optional Talking**: At random intervals, a philosopher may attempt to talk, invoking **requestTalk** and **endTalk** on the **Monitor** to ensure only one philosopher talks at a time.
4. **Cycle Repeats**: The cycle of thinking, eating, and optionally talking repeats for a predefined number of times (**DINING_STEPS**).
5. **Completion**: After completing their cycles, philosopher threads terminate. The program waits for all philosopher threads to finish before terminating, ensuring a clean and orderly shutdown.

Thread Lifecycle

A philosopher's lifecycle involves thinking for a period, attempting to eat by acquiring chopsticks (and optionally a pepper shaker), and then returning to thinking. The option to talk introduces variability in their behavior.

Synchronization Solution

Mechanisms Used

Our solution employs Java's **synchronized** keyword, along with **wait()** and **notifyAll()** methods, to ensure that philosophers don't simultaneously access shared resources (chopsticks/pepper shakers) and to manage their states effectively.

Role of the Monitor Class

The **Monitor** class is crucial for synchronization, enforcing rules that prevent deadlock (e.g., a philosopher can only eat if both neighbors are not eating) and coordinating access to shared resources.

Starvation-Free Method

To prevent starvation, we implement a fair ordering for philosophers attempting to eat, using a **PriorityQueue** based on the time a philosopher became hungry. This ensures every philosopher eventually gets to eat, regardless of their position at the table.

Rationale Behind the Solution

Design Choices

We chose a centralized **Monitor** to simplify the management of shared resources and philosopher state transitions. The use of Java's built-in synchronization primitives offered a robust and straightforward way to implement necessary locking and signaling mechanisms.

Challenges and Solutions

Avoiding deadlocks while preventing starvation was a significant challenge. Our solution ensures that philosophers only attempt to eat when it's safe, and by introducing a fair ordering system, we guarantee that all philosophers eventually eat, addressing potential starvation issues.

Conclusion

Our implementation of the Dining Philosophers problem demonstrates a balanced approach to solving synchronization challenges in concurrent programming. By carefully managing resource allocation and philosopher states, we avoid deadlocks and ensure no philosopher starves, providing a fair and efficient solution to this classic problem.