Maryam Soleimani 401222075

Q1 (reader-writer----- reader priority):

This code implements a **Reader-Writer problem** with **reader-priority** using threading.

1. ReaderPriorityReaderWriter Class

This class manages the shared resource and synchronization mechanisms.

Attributes:

- read_count: Keeps track of how many readers are currently reading the shared resource.
- read_count_lock: A semaphore to protect updates to read_count, ensuring atomicity.
- resource_access: A semaphore to control access to the shared resource. Writers and the first/last readers use it to ensure exclusive or shared access.
- shared_resource: A dictionary representing the shared resource, with a single key-value pair ({"value": 0}).

Methods:

- reader(reader_id): Represents a reader thread.
    - **Steps:**
        1. Acquire read_count_lock to safely update read_count.
        2. Increment read_count. If it's the first reader, acquire resource_access, blocking writers.
        3. Release read_count_lock.
        4. Read the shared_resource (simulated by printing its value).
        5. Simulate a delay with time.sleep to mimic reading time.
        6. Acquire read_count_lock, decrement read_count, and if it's the last reader, release resource_access.
        7. Release read_count_lock.
        8. Sleep for a short time before attempting to read again.
- writer(writer_id): Represents a writer thread.
    - **Steps:**
        1. Print that the writer wants to write.
        2. Acquire resource_access to ensure exclusive access.
        3. Modify shared_resource["value"] (increment by 1).

4. Simulate writing time with time.sleep.
5. Print the updated value of the shared resource.
6. Release resource_access for other threads.
7. Sleep for a short time before attempting to write again.

- **Reader Priority**:
    - Readers can access the resource concurrently as long as no writer is writing.
    - If a reader is reading, writers must wait until all readers finish.
    - Priority for readers is enforced because:
        - When a new reader arrives (read_count is incremented), writers are blocked until all readers finish.
        - Readers don't wait for writers unless resource_access is already locked.
- **Synchronization**:
    - **read_count_lock** ensures updates to read_count are atomic.
    - **resource_access** ensures:
        - Writers have exclusive access.
        - Readers collectively block writers while any reader is reading.

Q2(reader-writer----------writer priority):

This code implements a **Reader-Writer problem** using a simplified approach with a single Lock to manage access to a shared resource. The behavior prioritizes writers, as readers must wait when a writer is using the resource.

1. Shared Elements

- **write_lock**: A threading.Lock that ensures mutually exclusive access to the shared resource. Both readers and writers use it to synchronize access.
- **shared_resource**: A dictionary representing the shared resource with an initial value ({"value": 0}).

2. Writer Function

The writer function represents a writer thread that modifies the shared resource.

- **Steps**:

1. Acquires the write_lock to ensure exclusive access to the shared resource.
2. Prints a message indicating the start of the write operation.
3. Updates the shared_resource["value"] by incrementing it.
4. Simulates writing time using time.sleep(1).
5. Prints the updated value of the shared resource.
6. Releases the write_lock, allowing other threads (readers or writers) to access the resource.
7. Sleeps for a short duration (time.sleep(1)) before the next writing attempt.

3. Reader Function

The reader function represents a reader thread that tries to read the shared resource.

- **Steps**:
    1. Sleeps for a random duration (1-3 seconds) to simulate irregular read attempts.
    2. Attempts to acquire the write_lock non-blockingly (blocking=False).
        - If successful:
            - Reads and prints the value of the shared resource.
            - Simulates reading time with time.sleep(1).
            - Releases the write_lock after reading.
        - If unsuccessful:
            - Prints a message indicating the reader is waiting because a writer is accessing the resource.
            - Sleeps for a short duration (time.sleep(1)) before retrying.

1. **Writer Behavior**:
    - Writers always acquire the write_lock to ensure exclusive access to the shared resource.
    - While a writer is writing, no reader can access the resource because the write_lock is held.
2. **Reader Behavior**:
    - Readers attempt to acquire the write_lock without blocking.
    - If the write_lock is already held (e.g., by a writer), readers print a waiting message and retry later.

3. **Writer Priority**:
    - If a writer is holding the write_lock, readers cannot proceed, ensuring writers always complete their operations before readers can access the resource.

The implementation of this code using semaphore as a lock mechanism works almost the same.

Q3:

DP(deadlock free using monitors):

This code simulates the **Dining Philosophers Problem** using **monitors** to ensure synchronization and avoid **deadlock**.

1. DiningPhilosophers Class

This class implements the logic for synchronization and state management of the philosophers.

Attributes:

- **state**: A list tracking the current state of each philosopher (THINKING, HUNGRY, or EATING).
- **self**: A list of Condition objects (one for each philosopher) used to manage individual synchronization.
- **lock**: A shared lock (threading.Lock) ensuring mutual exclusion for critical sections.

States:

- **THINKING** (0): The philosopher is thinking.
- **HUNGRY** (1): The philosopher wants to eat.
- **EATING** (2): The philosopher is eating.

Methods:

1. **pickup(i)**:
    - Called when philosopher i wants to eat.
    - Steps:
        1. Acquire the lock to ensure exclusive access to the state.
        2. Set the philosopher's state to HUNGRY.
        3. Call test(i) to check if the philosopher can eat.
        4. If the philosopher cannot eat, wait on their condition variable (self[i]).
2. **putdown(i)**:

- Called when philosopher i is done eating.
- Steps:
    1. Acquire the lock to ensure exclusive access to the state.
    2. Set the philosopher's state to THINKING.
    3. Call test for the left and right neighbors of i to check if they can now eat.

3. **test(i)**:
    - Checks if philosopher i can eat based on the state of their neighbors.
    - A philosopher can eat if:
        - They are HUNGRY.
        - Their left neighbor ((i + 4) % 5) is not EATING.
        - Their right neighbor ((i + 1) % 5) is not EATING.
    - If the philosopher can eat, their state is set to EATING, and their condition variable (self[i]) is notified.
4. **initialization_code()**:
    - Initializes all philosophers' states to THINKING.

2. Philosopher Thread Function

The philosopher_thread function simulates the actions of a single philosopher.

- **Steps**:
    1. **Thinking**: The philosopher thinks for a random amount of time (1-3 seconds).
    2. **Hungry**: The philosopher becomes hungry and tries to pick up chopsticks by calling pickup.
    3. **Eating**: If successful, the philosopher eats for a random amount of time (1-2 seconds).
    4. **Done Eating**: The philosopher puts down the chopsticks by calling putdown.

The code avoids deadlock by implementing the following strategies:

1. **Monitor with Conditions**:
    - Each philosopher has a condition variable (self[i]) that allows them to wait until they can eat, ensuring fine-grained control.
2. **Neighbor Check (test Method)**:

- A philosopher can only eat if their neighbors are not eating, preventing circular waiting.

3. **Mutual Exclusion**:
    - The lock ensures that only one philosopher can modify the shared state list at a time, avoiding race conditions.

DP(deadlock free and starvation free):

This code is a **Dining Philosophers Problem** implementation that ensures the system is **deadlock-free** and **starvation-free** using an ordering mechanism to avoid circular waiting. It alternates the sequence in which philosophers acquire forks based on their ID (even or odd).

1. Global Variables

- n: Number of philosophers (set to 5 in this implementation).
- philosophers: A list to store threads representing each philosopher.
- forks: A list of Semaphore(1) objects, each representing a single fork. A semaphore allows synchronization and ensures that only one philosopher can hold a fork at a time.

2. Philosopher Functions

Philosophers are categorized into **even-indexed** and **odd-indexed**, and they acquire forks in a different order to avoid circular waiting:

Even Philosopher (even_philosopher):

1. **Acquire forks**:
    - First, the philosopher acquires the fork on their left (forks[philosopher_id]).
    - Then, they acquire the fork on their right (forks[(philosopher_id + 1) % n]).
2. **Eat**:
    - Simulates eating for a random time between 1-2 seconds.
3. **Release forks**:
    - Releases the right fork.
    - Then releases the left fork.

Odd Philosopher (odd_philosopher):

1. **Acquire forks**:
    - First, the philosopher acquires the fork on their right (forks[(philosopher_id + 1) % n]).

    - Then, they acquire the fork on their left (forks[philosopher_id]).
2. **Eat**:
    - Simulates eating for a random time between 1-2 seconds.
3. **Release forks**:
    - Releases the left fork.
    - Then releases the right fork.

3. Main Function

1. **Thread Initialization**:
    - Creates a thread for each philosopher.
    - Assigns even philosophers to the even_philosopher function and odd philosophers to the odd_philosopher function.

How Deadlock and Starvation are Avoided

1. Deadlock Prevention

A potential deadlock in the dining philosophers problem arises when all philosophers simultaneously pick up the fork on their left and wait for the fork on their right, causing circular waiting.

This implementation prevents deadlock by:

- **Ordering Fork Acquisition**:
    - Even philosophers pick up their left fork first, then their right fork.
    - Odd philosophers pick up their right fork first, then their left fork.
- **Breaking Symmetry**:
    - By alternating the order of fork acquisition between even and odd philosophers, the possibility of circular waiting is eliminated.

2. Starvation Prevention

Starvation occurs when a philosopher waits indefinitely because other philosophers continually block access to forks.

This implementation prevents starvation by:

- **Mutual Exclusion**:
    - A semaphore ensures that only one philosopher can hold a fork at any given time.

- Philosophers waiting for forks will eventually acquire them when neighboring philosophers finish eating and release the forks.

In conclusion:

The main idea is to prevent deadlock by alternating the order in which philosophers pick up forks: even-numbered philosophers pick up their left fork first, and odd-numbered philosophers pick up their right fork first. This ensures that neighboring philosophers do not simultaneously compete for the same fork.

By limiting competition to half the forks at any given time, the process avoids circular waiting and ensures that one philosopher can always proceed to eat. This approach also prevents starvation because the waiting time for forks is bounded. If a philosopher's neighbors are eating, they will eventually finish, allowing the philosopher to acquire the needed forks and eat.

For example, if philosopher 1 is waiting while philosophers 0 and 2 are eating, philosopher 1 will acquire the fork released by philosopher 0, then wait for philosopher 2 to finish. Thus, the waiting time remains finite, ensuring fairness and preventing starvation.