CMP 3001 – Project Report (Hamdi Kaan Kahveci – 2002746)

1. Introduction

The Reader-Writer problem is a classic example of a multi-threading synchronization issue in computer science. It occurs when a shared resource is being accessed concurrently by multiple processes, typically for reading and writing purposes. The main challenge is to synchronize the access to the resource in such a way that multiple readers can simultaneously read from the resource, but write operations require exclusive access.

2. Objective

The objective of this solution is to implement a synchronization mechanism that allows multiple readers to read concurrently while ensuring exclusive access for writers. Additionally, the solution aims to prevent both writerand reader starvation, ensuring fair access for both parties.

3. Solution Overview:

- **ReadWriteLock:** A custom lock mechanism that manages access to the shared resource
- **SharedData:** A representation of the shared resource
- **Writer:** A writer thread that modifies the shared resource
- **Reader:** A reader thread that reads the shared resource

4. Detailed Explanation:

- The **ReadWriteLock** class uses three semaphores: **readLock**, **writeLock**, and **turnstile**, all configured as fair semaphores. Fair semaphores ensure that threads acquire permits in the order they were requested, thereby maintaining an orderly access to the shared resource and preventing starvation. The **readLock** controls the access to the reader count, **writeLock** ensures exclusive access for writers, and the **turnstile** semaphore regulates the order of readers and writers, preventing reader starvation.

- More on turnstile logic:

  (i) When a writer wants to write, it acquires the **turnstile**, blocking any new readers from starting. This ensures that once a writer is ready, it will eventually get access to write, even if there are readers that are currently reading.

  (ii) Readers briefly acquire and then immediately release the **turnstile**. This means that if a writer

has acquired the turnstile, readers will queue up and wait for their turn after the writer is done.

(iii) Once the writer completes its task and releases the turnstile, the waiting readers can then proceed in the order they arrived, preventing starvation.

- The **SharedData** class represents the shared resource, which in this case is a simple integer variable. It includes synchronized methods to read and increment this variable, ensuring thread safety during these operations.

- **Writer Thread Flow**:

  1) Increment writerWaiting: A writer increments the **writerWaiting** counter, indicating its intention to write.
  2) Acquire turnstile semaphore: The writer acquires the **turnstile**. This step is crucial as it blocks new readers, ensuring that the writer will get a chance to write.
  3) Acquire writeLock: The writer then acquires **writeLock**, ensuring exclusive access to shared data.
  4) Writing operation: The writer performs the write operation on the shared resource.
  5) Decrement writerWaiting: After completing the write operation, the writer decrements the **writerWaiting.**
  6) Release locks: The writer releases both **writeLock** and **turnstile**, allowing other writers or readers to proceed.

- **Reader Thread Flow**:

  1) Acquire and release the turnstile: A reader acquires and immediately releases the **turnstile**. This ensures that readers queue up and wait for their turn, particularly after a writer is done.
  2) Acquire readLock and modify readCount: The reader acquires **readLock** to safely increment **readCount**. If it is the first reader, it attempts to acquire **writeLock** (blocking writers).
  3) Reading operation: If allowed (checked using **canRead()**), the reader then reads the shared resource.
  4) Release locks: The reader then releases **readLock**, and, if it was the last reader, it releases **writeLock** too, allowing writers to proceed.