**CMP3001 Operating Systems**

**Project Report (Synchronization)**

**1. Introduction**

In this project, the Reader-Writer problem is addressed. The Reader-Writer problem models scenarios where multiple threads attempt to access a shared resource. While reading operations can be concurrent, writing operations should not occur simultaneously with other operations. The project implements the **“ReadWriteLock”** class using the **“Semaphore”** class to control access.

**2. Explanation of the “ReadWriteLock” Class**

**<read && write>Lock() Methods:**

Called when a reading/writing operation begins, this method utilizes **“****<read || write>Semaphore”** to control access. If no other reading operation is ongoing, readLock locks **“writeSemaphore”** to prevent other writing operations. Each reading operation increments the **“readCount”**. At that time, writeLock blocks other operations. In summary, these methods implement a basic Read-Write Lock mechanism. The readLock method allows multiple readers to access the resource simultaneously but prevents writers from accessing it concurrently. The writeLock method ensures exclusive access for a writer, blocking both readers and other writers.

**<read && write>Unlock() Method:**

Called when a reading operation is completed, this method uses **“<read || write>Semaphore”** to control Access or allow other operations. If other reading operations are ongoing, it decrements the **“readCount”**, and if all reading operations are completed, it releases **“writeSemaphore”**.

**<read && write>Semaphore.acquire():**

The method starts by acquiring the read/write semaphore, which is a mechanism to control access to the shared resource for reading/writing.

**<read && write>Semaphore.release():**

Finally, the read semaphore is released, allowing other readers to acquire the read lock, and if there are no more active readers, the write semaphore is also released, allowing writers to acquire the write lock.

**3. Analysis of the “Test” Class**

The **“Test”** class serves as the orchestrator, managing the execution of a multithreaded environment to simulate concurrent reader and writer operations on shared resources. Through the utilization of the **“ExecutorService”** and the **“ReadWriteLock”** class, it establishes a cached thread pool and initializes the necessary synchronization mechanisms, namely semaphores, to control access to the shared resource. The class then launches multiple writer and reader threads, each interacting with the **“ReadWriteLock”** to emulate a scenario where readers can access the resource simultaneously, but exclusive access is granted to writers. This structured approach facilitates a controlled environment for testing the effectiveness of the implemented reader-writer synchronization mechanism.

**4. Description of the “Writer” and “Reader” Classes**

In the provided code, the **“Writer”** and **“Reader”** classes represent distinct components of a concurrent system employing a Read-Write Lock mechanism for shared resource access.

**Writer Class:**

The **“Writer”** class, encapsulated within the **“ReadWriteLock”** implementation, is responsible for simulating write operations on the shared resource. In its **“run”** method, the writer periodically acquires a write lock using the **“writeLock”** method, executes a writing task, and subsequently releases the write lock through the **“writeUnLock”** method. This continual loop imitates a scenario where multiple writers contend for access to the shared resource, and the implemented locking mechanism ensures that only one writer accesses the resource at any given time, preventing potential conflicts.

**Reader Class:**

On the other hand, the **“Reader”** class represents the counterpart responsible for simulating read operations on the shared resource. In its **“run”** method, the reader acquires a read lock using the **“readLock”** method, performs a reading task, and releases the read lock through the **“readUnLock”** method. Similar to the writer, the reader operates within an infinite loop, mimicking a scenario with multiple readers concurrently accessing the shared resource. The implemented Read-Write Lock ensures that writers and readers coexist peacefully, with writers given exclusive access while readers can access the resource concurrently.

This design ensures a synchronized and controlled access environment for both reading and writing operations, preventing potential conflicts and ensuring the integrity of the shared resource. The concurrent execution of multiple readers and writers is effectively coordinated through the mechanisms provided by the **“ReadWriteLock”** implementation.

**5. Application Results and Analysis**

**Success of Synchronization:**

The **“Semaphore”** and **“ReadWriteLock”** classes used in the project demonstrate successful synchronization. While writing operations maintain data integrity, reading operations occur in a coordinated manner.

**Performance Analysis:**

The project allows for an analysis of the costs introduced by synchronization and the performance of the solution. For example, how reading operations do not hinder writing operations and permit them once completed.

**6. Learning Outcomes**

This project, focusing on concurrent programming using a Read-Write Lock mechanism, has provided valuable insights into several key learning outcomes. Through the implementation of the project, the following outcomes have been achieved:

**Understanding of Concurrent Programming Concepts:**

Mastery of fundamental concepts related to concurrent programming, including semaphores, threads, and the challenges associated with shared resource access.

**Application of Read-Write Locks:**

Proficiency in applying Read-Write Locks to manage concurrent access to shared resources, ensuring a balance between read and write operations while preventing potential conflicts.

**Semaphore Usage:**

Hands-on experience in utilizing semaphores as synchronization primitives to control access to critical sections of code, fostering a deeper understanding of semaphore operations such as acquire and release.

**Implementation of Infinite Loops:**

Practical implementation of infinite loops in the **“Writer”** and **“Reader”** classes to simulate continuous read and write operations, contributing to a comprehensive understanding of how concurrent systems operate over an extended period.

**Error Handling in Concurrent Environments:**

Exposure to error handling techniques in concurrent environments, exemplified through the use of try-catch blocks to manage InterruptedExceptions, addressing potential issues arising from thread interactions.

**Thread Pool Management:**

Proficient use of the **“ExecutorService”** and **“Executors”** framework to manage a thread pool efficiently, allowing for the concurrent execution of multiple reader and writer threads.

**Practical Application of Sleep Utilities:**

Practical application of sleep utilities to simulate delays, contributing to a realistic representation of scenarios where threads may need to wait for certain conditions to be met.

**Real-World Relevance:**

Recognition of the real-world relevance of Read-Write Locks in scenarios where balancing concurrent read and write access is crucial, such as database management systems or file handling.

In conclusion, through this project, I have successfully addressed important learning outcomes related to concurrent programming and provided a solid foundation for understanding and implementing synchronization mechanisms in Java. The hands-on experience I gained through the development of the Read-Write Lock-based system has not only strengthened my theoretical knowledge but also honed my practical skills necessary for effective concurrent programming.