Reader-Writer Synchronization Project Report

Objective

The objective of this project is to synchronize multiple reader and writer threads that access a shared data resource. The goal is to allow multiple readers to access the data concurrently while ensuring that writers have exclusive access when needed.

Overview

This project uses semaphores and mutexes in Java to manage synchronization. Readers are allowed to read concurrently as long as there are no writers active. When a writer is active, all readers must wait until the writer finishes. The synchronization mechanism prevents deadlocks and starvation.

Description of Code

In the provided code, two types of threads are created: Reader and Writer.

1. **Reader**: Each reader thread first checks if there are any writers waiting. If there are no writers, the reader acquires the `readmutex`, increments the reader count, and reads the data. After finishing, the reader decrements the reader count and releases the `readmutex`.
2. **Writer**: Each writer thread waits for the `readmutex` to be released, then acquires the `writemutex` for exclusive access to the shared data. It modifies the data, then releases both semaphores once done.

The system ensures that readers can access the data concurrently when no writers are present, and writers are given exclusive access to modify the data.

Key Components

- **Shared Data**: Integer `sharedData` represents the data being accessed and modified by the

threads.

- **Semaphores**: `readmutex` and `writemutex` are used to manage access to the shared resource and control concurrent access to it.
- **Reader Count**: `readercount` keeps track of how many readers are currently accessing the shared data.
- **Waiting Writers**: `waitingWriters` tracks how many writers are waiting for access.

Functional Flow

1. **Reader**: A reader checks if any writers are waiting, acquires `readmutex`, and increments `readercount`. After reading, it decrements the count and releases `readmutex`.
2. **Writer**: A writer waits for the `readmutex`, then acquires the `writemutex` for exclusive access to modify the data. After writing, it releases both semaphores.

Key Outcomes

- **Synchronization**: Writers get exclusive access to the shared data, while readers can access it concurrently when no writer is active.
- **Concurrency**: Multiple threads (readers and writers) run concurrently, demonstrating the management of shared memory access.

Conclusion

This implementation ensures that readers and writers are synchronized efficiently. Writers are guaranteed exclusive access, and readers can read concurrently when appropriate. The system prevents deadlocks and guarantees fairness between the threads.