**Operating Systems**

**Assignment 1 Report**

**Readers-Writers Problem with Synchronization**

**Introduction**

This project employs the “Readers-Writer’ synchronization problem” with specific focus on ensuring thread safety, writer priority, and balanced distribution of readers among multiple copies of a file. The program is written in Python and simulates file access through in-memory strings rather than real file I/O

The main goals were to:

- Ensure that the common resources are being utilized in a thread-safe manner.

- Proceed with writer access so that after a writer signals its interest, no new readers are permitted to start.

- Distribute reader load evenly across three replicas.

**Design Overview**

The system consists of multiple reader threads and a single writer thread:

* Reader Threads: Each reader selects the replica with the fewest active readers (tracked by a shared array) to balance the load. Readers wait if a writer is either active or waiting
* Writer Thread: The writer signals its intent by setting a waiting flag, then waits until all readers have finished before updating all replicas simultaneously. Once done, it resets the flag and notifies waiting threads
* Logging: All operations (reads and writes) are logged with a timestamp, actor identification, operation details, and the current state of the system. This log helps in verifying correctness and debugging

**Synchronization Mechanism**

Two key synchronization primitives are used in the original implementation:

Mutex (Lock)

* Purpose: ensures mutual exclusion when accessing shared state variables (“read\_counts”, “writer\_active”, and “writer\_waiting”)
* Implementation: the Python “threading.Lock()” is used. This lock prevents race conditions by allowing only one thread at a time to update or read shared data

Condition Variable (Monitor)

* Purpose: coordinates the waiting and notification between reader and writer threads
* Implementation: a “threading.Condition(lock)” is created with the previously defined lock. It allows reader threads to wait until a writer is not active or waiting and enables the writer to wait until all active readers have finished

**How They Work Together**

Readers:

* A reader acquires the condition lock and checks whether a writer is active or waiting
* If so, the reader waits on the condition variable
* Otherwise, it selects the replica with the fewest active readers and increments that count

Writer:

* The writer sets a waiting flag to block new readers
* It then waits (via the condition variable) until all active readers finish (for example, all values in “read\_counts” become zero).
* The writer sets an active flag, updates all replicas, logs the operation, and finally clears the active flag and notifies waiting threads

**Writer Priority and Reader Load Balancing**

Writer Priority:

* The writer sets a flag (“writer\_waiting”) before it attempts to write
* New reader threads check this flag upon entering. If the flag is set, they wait, thereby giving priority to the writer
* The writer only proceeds once all active readers have finished, ensuring exclusive access during the write operation

Reader Load Balancing:

* Each reader thread checks the shared “read\_counts” list to determine which file replica currently has the fewest active readers
* The reader then increments the count for that replica, ensuring that no single replica becomes overloaded
* This dynamic selection balances the read load across the three replicas

**Challenges Encountered**

During the development process, several challenges were addressed:

1. Ensuring Writer Priority:

* A primary challenge was preventing new readers from starting once a writer signaled its intent. This was effectively managed by using the “writer\_waiting” flag along with the condition variable, ensuring that new readers wait until the writer completes its update

2.Balancing Reader Load:

* Accurately tracking and updating the reader count for each replica required careful handling of the shared “read\_counts” array. The solution needed to ensure that updates to this array were done atomically to avoid race conditions, which was achieved by protecting these updates with the mutex

3. Avoiding Deadlocks:

* The use of multiple synchronization primitives (locks and condition variables) always carries the risk of deadlocks. Special attention was given to the order of lock acquisition and release. By keeping critical sections as short as possible and ensuring that locks are released immediately after updating shared state, the design avoids deadlock scenarios

4. Logging Overhead:

* To maintain clear and non-interleaved log entries, a dedicated log lock was used. This ensures that even when multiple threads try to log simultaneously, the log file remains consistent and readable

**Efficiency Considerations**

Minimized Lock Contention:

* Locks are acquired only, when necessary (during updates to shared variables), and released as soon as possible. This reduces the time threads spend waiting for access.

Condition Variables vs. Busy-Waiting:

* The use of condition variables avoids busy waiting, allowing threads to sleep until notified. This makes the implementation more efficient in terms of CPU utilization

Load Distribution:

* By selecting the replica with the fewest active readers, the solution ensures that no single file replica becomes a bottleneck, even under high load

**Conclusion**

The implemented solution meets the requirements for a thread-safe, efficient, and well-structured readers-writers problem with load balancing. The use of mutexes and condition variables provides a robust synchronization mechanism that guarantees writer priority and balanced reader distribution. Detailed logging facilitates debugging and validation of system behavior, while the design addresses common concurrency challenges such as race conditions and deadlocks. Overall, this implementation demonstrates a comprehensive understanding of synchronization concepts and provides a clear, maintainable solution for managing concurrent access to shared resources.