OS Project Report

"System Call for Semaphore (Example: Reader- Writer Problem)"

1. Introduction

Modern operating systems require careful management of shared resources to avoid conflicts between concurrent processes. The **Reader-Writer Problem** is a classic synchronization challenge that models such situations, where multiple readers can access shared data simultaneously but writers need exclusive access. This project simulates the Reader-Writer problem by implementing a user-space multithreaded application in C using POSIX threads and semaphores and integrates a **custom system call** that tracks and measures the average wait time between request and entry into the critical section. It ensures fair reader priority while preserving mutual exclusion for writers.

2. Group Members

- Saad Zaidi (23K-0874)
- Muhammad Umer Khan (23K-0581)
- Ebad Ahmed (23K-0560)
- Waliuddin Ahmed (23K-0719)

3. Objectives

1. **Simulate Multithreaded Concurrency:** Create an environment where multiple readers and writers operate concurrently while ensuring data consistency.
2. **Implement Synchronization Techniques:** Use semaphores and mutexes to avoid race conditions, ensure mutual exclusion, and enable fair resource sharing.
3. **Integrate System Call:** Develop a custom system call that calculates and outputs the average time spent by threads waiting to enter the critical section.
4. **Emphasize Reader Priority:** Implement a synchronization mechanism where multiple readers can access the critical section simultaneously unless a writer is waiting.

4. Tools & Technologies

- **Language:** C (GCC Compiler)
- **Threading Library:** POSIX Threads (pthread.h) for thread creation and management.
- **Synchronization Primitives:** Semaphores and Mutexes for safe concurrent access.
- **Operating System:** Linux (Ubuntu)

5. System Overview

1 Architecture

- **Reader Threads:** Multiple threads representing readers that access the shared resource.
- **Writer Threads:** Threads that require exclusive access to modify the shared resource.
- **Semaphores:**
    - `mutex`: Protects the `read_count` variable.
    - `rwmutex`: Allows exclusive access for writers.
    - `avgmutex`: Protects shared variable `avg_time` during updates.
- **Custom System Call:** Measures the average wait time experienced by threads.

2 Synchronization Concepts

- **Multiple Readers:** Multiple readers can enter the critical section concurrently if no writer is active.
- **Exclusive Writers:** Writers must have exclusive access — no other reader or writer can be inside during a write operation.
- **Fairness:** Readers have priority when accessing the shared resource unless a writer is already active.

6. Use Cases

1. Multiple Readers Accessing Simultaneously

Scenario: Five readers request access at the same time.

- Outcome: All readers enter the critical section together without waiting, demonstrating high concurrency.

2. Writer Accessing Resource

Scenario: A writer requests access while multiple readers are reading.

- Outcome: Writer must wait until all readers have exited before gaining access.

3. Heavy Mixed Load

Scenario: 10 readers and 3 writers access the resource randomly.

- Outcome: Readers are served concurrently until a writer arrives, at which point new readers are blocked until the writer finishes.

4. Performance Evaluation

Scenario: Admins monitor the average time taken for threads to access the critical section.

- Outcome: Average time is computed and displayed via a custom system call, helping evaluate system efficiency.

7. Main Flow of the Project

Program Initialization (main.c)

- Semaphores `mutex`, `rwmutex`, and `avgmutex` are initialized.
- Number of readers and writers are read from `input.txt`.
- Reader and writer threads are created and launched.
- Threads are joined after completion.
- Semaphores are destroyed.
- Custom system call invoked to print average access time.

Reader Thread Flow

1. **Request Phase:**
   - Record request time.
   - Lock `mutex` to safely increment `read_count`.

   - If it is the first reader, lock `rwmutex` to block writers.
   - Unlock `mutex`.
2. **Critical Section:**
   - Simulate reading operation with sleep.
3. **Exit Phase:**
   - Lock `mutex` and decrement `read_count`.
   - If last reader, unlock `rwmutex`.
   - Unlock `mutex`.
4. **Update:**
   - Lock `avgmutex`, update average waiting time, and unlock.

Writer Thread Flow

1. **Request Phase:**
   - Record request time.
   - Wait on `rwmutex` to gain exclusive access.
2. **Critical Section:**
   - Simulate writing operation with sleep.
3. **Exit Phase:**
   - Release `rwmutex` after writing.
4. **Update:**
   - Lock `avgmutex`, update average waiting time, and unlock.

8. Synchronization Mechanisms

- **Semaphore `mutex`:** Guards `read_count` to avoid race conditions.
- **Semaphore `rwmutex`:** Ensures mutual exclusion between readers and writers.
- **Semaphore `avgmutex`:** Protects updates to the shared `avg_time` variable.

9. File Operations and Data Consistency

- **Input File:** `input.txt` provides dynamic configuration for number of readers and writers.
- **Consistency:** Writers gain exclusive access; readers can access concurrently if no writer is active.

- **Safety:** Proper use of semaphores ensures that no thread can corrupt shared data.

10. Synchronization Guarantees

- **Concurrent Readers:** Multiple readers can operate simultaneously.
- **Exclusive Writing:** Only one writer allowed inside at a time.
- **Fair Access:** Priority given to readers but without starving writers.
- **Deadlock-Free Operation:** Semaphore order and logic ensures no circular waits.

11. Program Termination

- Reader and writer threads complete execution.
- Semaphores are destroyed.
- System call invoked to print final performance metrics.
- Program exits gracefully.

12. Summary of Synchronization Flow

- **Readers:**
  - Increment `read_count`.
  - First reader locks `rwmutex`.
  - Last reader unlocks `rwmutex`.
- **Writers:**
  - Wait for `rwmutex`, write exclusively, and release lock.
- **avgmutex:**
  - Ensures safe updates to average access time.

13. Dry Run

Scenario: 2 Readers, 1 Writer

Step	Action	Semaphore Operations	Comments
1	Writer requests CS at 10:15	wait(rwmutex)	Writer locks critical section
2	Writer enters at 10:15	-	Writes for 2 seconds
3	Writer exits at 10:17	post(rwmutex)	Critical section free
4	Reader 1 requests at 10:18	wait(mutex), increment(read_count), wait(rwmutex) if first	First reader locks critical section
5	Reader 2 requests at 10:18	wait(mutex), increment(read_count)	Allowed, no wait needed
6	Reader 1 and 2 read simultaneously	-	Reading phase
7	Reader 1 exits at 10:20	wait(mutex), decrement(read_count)	Reader 2 still inside
8	Reader 2 exits at 10:21	Wait(mutex), decrement(read_count), post(rwmutex) if last	Critical section unlocked

15. Conclusion

This project demonstrates a robust solution to the Reader-Writer synchronization problem using semaphores and POSIX threads. The inclusion of a system call for performance monitoring adds depth to the project. It successfully models the key challenges faced in concurrent programming and highlights efficient synchronization techniques crucial for modern operating systems.