CS3523: Programming Assignment-4

DOKKU HEMANADH
CS22BTECH11018

March 18 , 2024

1 Introduction

Solving the Readers-Writers Problem Using Semaphores in C++: The Readers-Writers problem deals with managing access to a shared resource where multiple threads (readers and writers) are involved. In the writer-preference solution, writers are given priority over readers to access the critical section. This ensures that once a writer is ready to access the resource, it should not be delayed unnecessarily. Fair Readers-Writers problem aims to provide fairness in accessing the critical section to both readers and writers. We aim to implement both solutions using Semaphores in C++, comparing the average and worst-case times for each thread to access the critical section.

2 Aims and Objectives

2.1 Aim

The aim of the code is to implement solutions to the readers-writers problem using semaphores in C++, including prioritizing writers and ensuring fairness, with objectives of thread synchronization, resource access control, logging, parameterization, execution time measurement, resource cleanup, and measuring average entry time.

2.2 Objectives

1. **Implement Readers-Writers Problem**: The main aim of the code is to implement a solution to the readers-writers problem using semaphores.
2. **Thread Synchronization**: Ensure that readers and writers access the shared resource in a synchronized manner, preventing race conditions and ensuring data integrity.
3. **Resource Access Control**: Coordinate access to the shared resource between multiple readers and writers, allowing either multiple readers or a single writer to access the resource at any given time.
4. **Fairness**: Implement a fair solution to the readers-writers problem, ensuring that both readers and writers are granted access to the resource in the order they requested it.

5. **Logging**: Log the actions performed by reader and writer threads, such as when they request entry into the critical section, when they enter the critical section, and when they exit the critical section. This logging helps in understanding the execution flow and identifying any issues or bottlenecks.

6. **Parameterization**: Read parameters (number of readers, number of writers, mean delay for critical section, mean delay for remaining time) from an input file (`inp-params.txt`), allowing easy customization of the simulation.

7. **Measure Execution Time**: Measure the total execution time of the program, providing insights into the performance of the implemented solution.

8. **Resource Cleanup**: Properly destroy semaphores and close log files to release system resources and ensure a clean exit from the program.

9. **Measure Average Entry Time**: Calculate and record the average time taken by a thread to gain entry into the Critical Section for each algorithm (RW and Fair-RW). This measurement provides insights into the efficiency and fairness of the implemented solutions, helping to compare their performance in terms of thread waiting times. The results can be stored in a file named `AverageTime.txt` for analysis and further optimization if needed.

3 Code Overview

1. **Includes**: The code includes necessary C++ standard libraries (`iostream`, `fstream`, `thread`, etc.) and POSIX semaphore library (`semaphore.h`).

2. **Namespace Declarations**: `using namespace std;` and `using namespace chrono;` are used for convenience.

3. **Shared Variables**:

   - `readcount`: Number of readers currently accessing the resource.
   - `writecount`: Number of writers currently accessing the resource.
   - `writing`: Boolean flag indicating whether a writer is currently writing.
   - `muCS`: Mean value for the critical section time delay.
   - `muRem`: Mean value for the remaining time delay.

4. **Semaphores**:

   - `read_lock`: Semaphore for synchronizing access to `readcount`.
   - `write_lock`: Semaphore for synchronizing access to `writecount`.
   - `readTry`: Semaphore for controlling access to the critical section by readers.
   - `read_write_lock`: Semaphore for ensuring mutual exclusion between readers and writers.
   - `resource`: Controls access (read/write) to the resource. Binary semaphore.
   - `rmutex`: For syncing changes to shared variable `readcount`.

- `serviceQueue`: Semaphore for ensuring fairness.
- `l`: Additional semaphore used for locking file writing.

5. **Output Files**: An output file named ”RW-log.txt” is opened for writing.
6. **Output Files**:An output file named ”FaiRRW-log.txt” is opened for writing.
7. **Function Definitions**:
   - `getSysTime()`: Function to get the current system time in hours, minutes, seconds, and microseconds.
   - `reader()`: Function implementing the behavior of reader threads.
   - `writer()`: Function implementing the behavior of writer threads.
   - `reader_fair()`: Function implementing the behavior of reader threads.
   - `writer_fair()`: Function implementing the behavior of writer threads.
8. **Main Function**:
   - Parameters are read from the file ”inp-params.txt”.
   - Semaphores are initialized.
   - Writer and reader threads are created and started.
   - Writer and reader threads are joined.
   - Total execution time is calculated.
   - Output file is closed.
   - Semaphores are destroyed.

4 Code Explanation

4.1 Reader_writerpreferencefunction

- The code sets up a random number generator, which is like a dice roll, using the current time to ensure randomness.
- It also sets up two different timers:
  - One timer to measure how long a reader spends inside the critical section (`distribution_muCS`).
  - Another timer to measure how long a reader spends outside the critical section (`distribution_muRem`).

1. **Loop**:
   - The code runs a loop a certain number of times (`kr` times), where each iteration represents a single time a reader accesses the critical section.
2. **Critical Section Entry**:
   - Before a reader enters the critical section, it needs to request access.

- It does this by signaling its intention to enter the critical section.
- Once signaled, it waits for its turn to enter the critical section without interfering with other threads.

3. **Entering Critical Section**:
   - When it's the reader's turn, it enters the critical section.
   - Inside, it performs whatever task it needs to do.
   - There might be a random delay to simulate how long this task takes.
4. **Critical Section Exit**
   - After completing its task inside the critical section, the reader signals that it's leaving.
   - If it's the last reader leaving, it signals that the critical section is now free for writers to enter.
5. **Remaining Time Simulation**:
   - After leaving the critical section, the reader simulates doing other work outside the critical section.
   - There's another random delay to mimic the time it takes to complete this work.

4.2 writer_writerpreferencefunction

1. **Initialization of Random Number Generator and Distributions**:
   - The `default_random_engine` is initialized with a seed based on the current time, ensuring randomization.
   - Exponential distributions `distribution_muCS` and `distribution_muRem` are created. These distributions are likely used to model the random delays in entering the critical section (`muCS`) and the remaining time outside the critical section (`muRem`).
2. **Loop**:
   - The loop runs `kw` times, where `kw` seems to represent the number of write operations the writer thread should perform.
3. **Request to Enter Critical Section**:
   - It waits on a semaphore `l` before logging a request to enter the critical section. This semaphore likely controls access to the shared resource.
4. **Acquiring Write Lock**:
   - It acquires a lock (`write_lock`) before incrementing the `writecount`. This is to ensure that only one writer can enter the critical section at a time.
5. **Blocking Readers**:

- If it's the first writer (`writecount == 1`), it waits on another semaphore `readTry`. This likely blocks readers from entering the critical section while a writer is active.

6. **Entering Critical Section**:

   - It waits on semaphores `read_write_lock` and `l` before logging the entry into the critical section.

7. **Simulating Activity in Critical Section**:

   - It introduces a random delay (`delay_CS`) to simulate activity within the critical section. This is done using an exponential distribution generated from the random number generator.

8. **Exiting Critical Section**:

   - It logs the exit from the critical section after the random delay, releases the semaphores, and signals that the critical section is now vacant.

9. **Decrementing Write Count**:

   - It decrements the `writecount` after exiting the critical section and potentially unblocks waiting readers (`writecount == 0`).

10. **Simulating Remaining Time Outside Critical Section**:

    - It introduces another random delay (`delay_Rem`) to simulate the remaining time outside the critical section.

4.3 reader_fair Function

1. **Initialization**: This function is responsible for simulating a reader thread in a system where multiple readers and writers compete for access to a shared resource. It initializes a random number generator and sets up exponential distribution objects to generate random delays.

2. **Looping for Read Operations**: Inside a loop, the reader thread tries to enter the critical section `kr` times. Each iteration represents a read operation.

3. **Request to Enter Critical Section**: The reader thread first requests permission to enter the critical section, indicating its intention to read the shared resource.

4. **Access Control**: It then acquires locks to control access to shared resources. It ensures mutual exclusion by incrementing a `readcount` variable and, if it's the first reader, acquiring a lock on the resource.

5. **Entering Critical Section**: Once access is granted, the reader enters the critical section, indicating it's reading the resource.

6. **Random Delay for Reading**: Inside the critical section, there's a random delay introduced to simulate varying execution times for reading operations.

7. **Exiting Critical Section**: After completing reading, the reader exits the critical section, releasing locks appropriately.

8. **Random Delay Outside Critical Section**: Finally, the reader introduces another random delay before attempting to enter the critical section again. This delay ensures fairness and simulates varying time intervals between consecutive read operations.

4.4 writer_fair Function

1. **Initialization**: Similar to the reader function, this function initializes a random number generator and sets up exponential distribution objects for random delays.

2. **Looping for Write Operations**: Inside a loop, the writer thread attempts to enter the critical section `kw` times. Each iteration represents a write operation.

3. **Request to Enter Critical Section**: The writer thread requests permission to enter the critical section, indicating its intention to write to the shared resource.

4. **Access Control**: It acquires locks to control access to shared resources. Unlike readers, writers directly acquire a lock on the resource without the need for a separate `readcount` mechanism.

5. **Entering Critical Section**: Once access is granted, the writer enters the critical section, indicating it's writing to the resource.

6. **Random Delay for Writing**: Inside the critical section, there's a random delay introduced to simulate varying execution times for write operations.

7. **Exiting Critical Section**: After completing writing, the writer exits the critical section, releasing locks appropriately.

8. **Random Delay Outside Critical Section**: Similar to the reader function, the writer introduces another random delay before attempting to enter the critical section again. This delay ensures fairness and simulates varying time intervals between consecutive write operations.

5 Graph observations

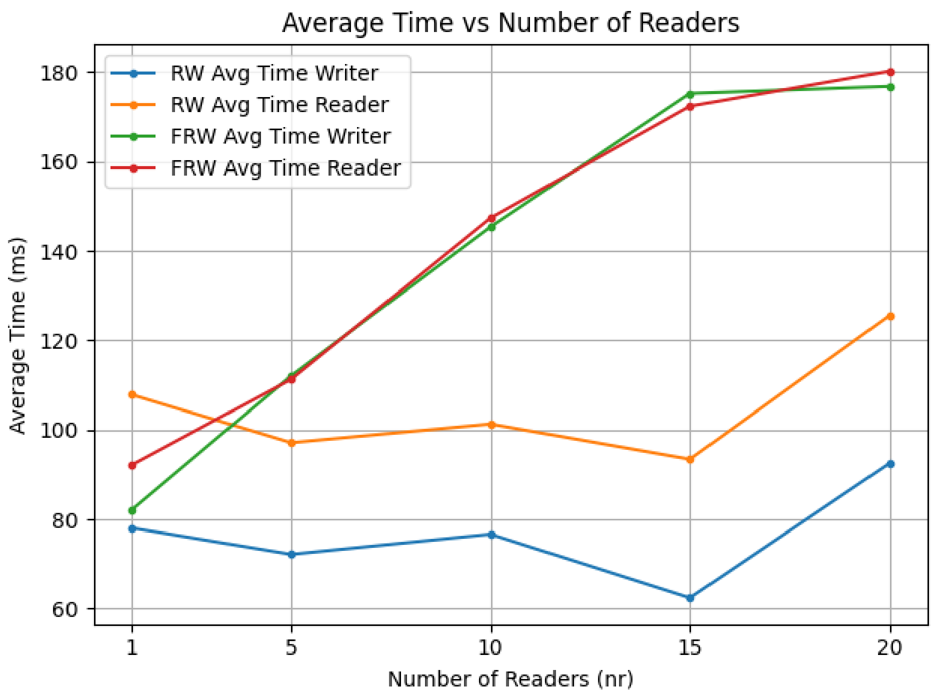

Figure 1: avg waiting time with constant Writers

Number of Readers (nr)	RW Avg Time Writer (ms)	RW Avg Time Reader (ms)	FF
1	78.0671	107.8836	
5	72.1453	97.1218	
10	76.6279	101.283	
15	62.4276	93.428	
20	92.545	125.603	

Table 1: Average Time vs Number of Readers

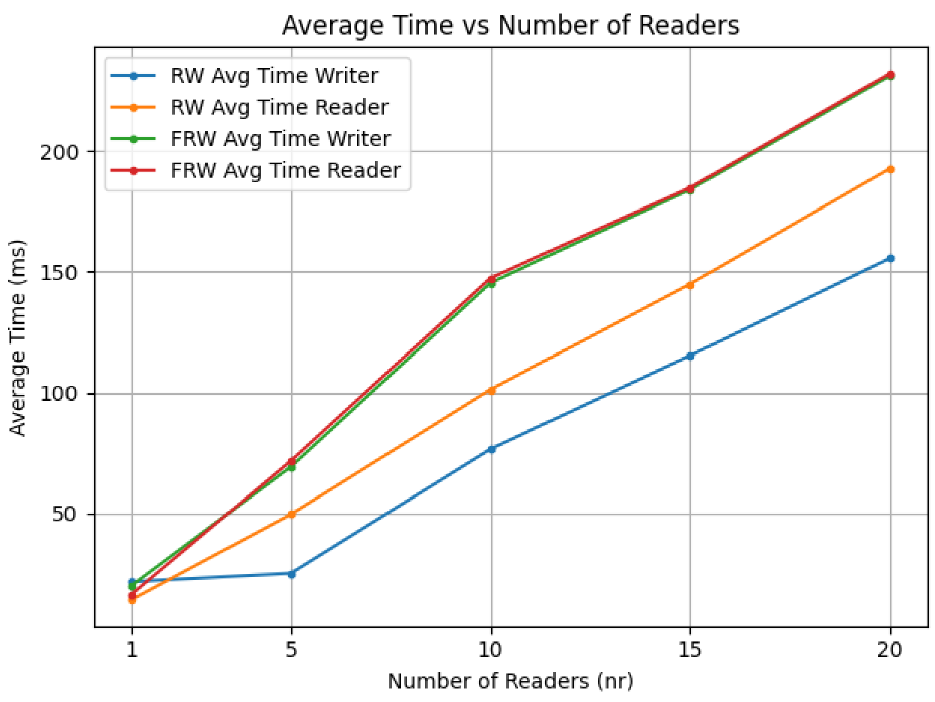

Figure 2: avg waiting time with constant Readers

Number of Writers (nw)	RW Avg Time Writer (ms)	RW Avg Time Reader (ms)	F
1	21.8171	14.2936	
5	25.2953	49.6518	
10	76.6279	101.283	
15	115.276	144.868	
20	155.545	192.603	

Table 2: Average Time vs Number of Writers

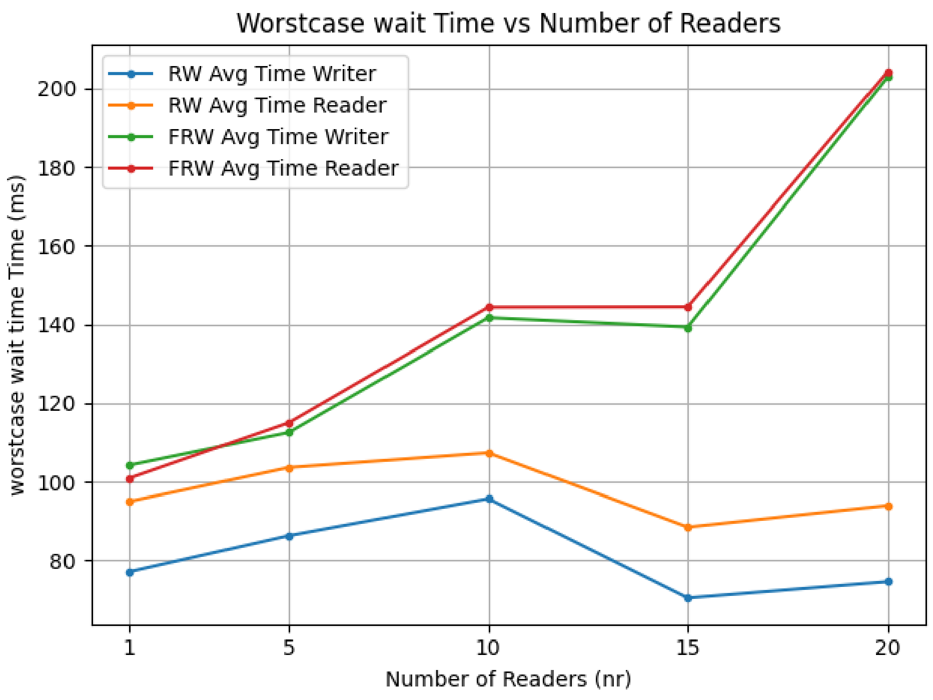

Figure 3: worstcase waiting time with constant Writers

Number of Readers (nr)	RW worstcase Time Writer (ms)	RW worstcase Time Reade
1	77.0671	94.9136
5	86.253	103.5618
10	95.6279	107.283
15	70.4276	88.428
20	74.545	93.903

Table 3: Worstcase Wait Time vs Number of Readers

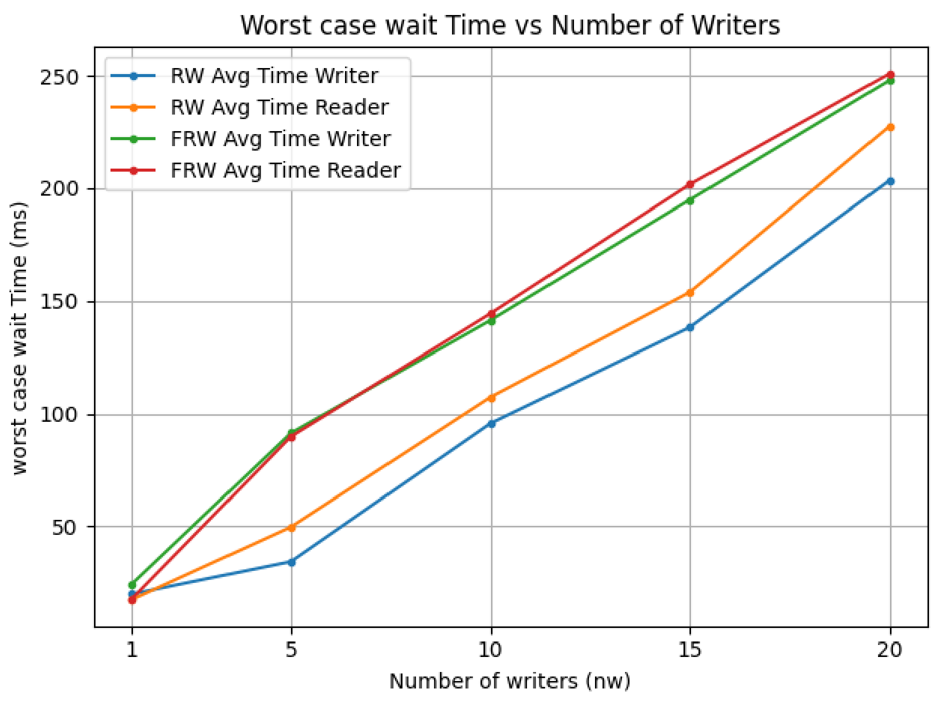

Figure 4: worstcase waiting time with constant Readers

Number of Writers (nw)	RW worstcase Time Writer (ms)	RW worstcase Time Reade
1	19.8171	17.2936
5	34.2953	49.6518
10	95.6279	107.283
15	138.276	153.868
20	203.545	227.603

Table 4: Worst Case Wait Time vs Number of Writers

6 Graph analysis

Experiment 1 and 2:

(a) Among the four threads, the writer thread in the writer-preference solution demonstrates superior performance. It outperforms threads in the fair solution due to the writer thread's ability to wait only for its designated time. In contrast, in the fair solution, writer threads must wait longer to ensure fairness, allowing all preceding reader threads to complete.

(b) In the fair solution, the wait times for both reader and writer threads are nearly identical, as expected, since both are treated with equal priority.

Experiment 3 and 4:

(a) The worst-case waiting times for reader threads in the writer-preference solution significantly surpass those of other threads because reader threads are susceptible to starvation. Consequently, it's highly probable that certain reader threads experience starvation, thereby increasing the worst-case wait time.

(b) In the fair solution, no thread experiences starvation, as both reader and writer threads are accorded equal priority. Consequently, the worst-case waiting times remain consistent.

(c) Writer threads in the writer-preference solution exhibit the least worst-case wait time because they are prioritized over reader threads.