# CS4533 Concurrent Programming

## Take Home Lab 1

**Group Members:**

- 200081B - Chandeepa E.A.C.
- 200698X - Weerasekara W.M.T.B.

## Step 1

### Design Explanation

**Objective:**

The first step in this lab involves designing a solution to generate operations for the linked list based on specified fractions (mMember, mInsert, mDelete) using a given number of threads. This design must efficiently distribute the operations among threads and accurately measure the execution time of these operations.

**Files Structure:**

linked_list.c and linked_list.h : Implementation of the linked list data structure.
serial.c : Source code for the serial version.
mutex.c : Source code using mutex locks.
rw_lock.c : Source code using read-write locks.
Makefile : Automates the build and run processes.

**Explanation:**

First, the linked list is initialized by n random values, and then the m number of operations is distributed among the number of threads equally. For instance, if there are 10,000 operations (m=10000) and 8 threads (thread_count=8), each thread will handle 1,250 operations so that each thread works on a specific set of operations.

Inside the ThreadFunction, it runs Member operations (m/thread_count)*mMember times, Insert operations (m/thread_count)*mInsert times, and Delete operations (m/thread_count)*mDelete times. ThreadFunction runs Member, Insert, and Delete operations in a random order. Used Pthreads library for mutex and read-write lock implementations.

- For the serial implementation, no synchronization is required.
- For the mutex-based parallel implementation, a single mutex lock controls access to the linked list, ensuring that only one thread can modify the list at any time.
- For the read-write lock implementation, utilize read-write locks to allow multiple concurrent reads operations (Member) while ensuring exclusive access for writes operations (Insert and Delete).

The program runs 100 initial runs at the beginning and calculates the mean and standard deviation. Then it calculates the required number of runs to get results within an accuracy of ±5% and 95% confidence level using the following equation,

$$n = \left(\frac{z \times \sigma}{E}\right)^2$$

Where:

- z is the z-score corresponding to the desired confidence level (1.96 for 95% confidence).
- σ is the population standard deviation, which is estimated by the sample standard deviation.
- E is the margin of error.

Then if the required number of runs is greater than the initial number of runs (100), then it will rerun the required number of times.

**Performance Evaluation Outcomes**:

- The average execution time and standard deviation for each configuration (number of threads) will be saved in the results folder after each run.
- Diagrams will be saved in the diagrams folder after each run.

**Computer used:**

CPU:

- Intel(R) Core(TM) i7-119G7 @2.90GHz 1.80 GHz
- 4 physical cores, 8 logical processors

· Memory:

- size 8GB
- type SODIMM

· Operating system:

- Windows 11 64-bit

· Tools:

- Compiler: gcc
- Libraries: pthread, sys/time
- Visual studio code

## Step 3

**Case 1: (mMember = 0.99 | mInsert = 0.005 | mDelete = 0.005)**

| Implementation | No of threads | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | | 2 | | 4 | | 8 | |
| | Average | Std | Average | Std | Average | Std | Average | Std |
| Serial | 9203.220 | 294.972 | | | | | | |
| One mutex for entire list | 10878.243 | 6856.175 | 16598.130 | 4345.589 | 15090.909 | 4897.508 | 16014.285 | 4413.820 |
| Read-Write lock | 11110.1836 | 6754.749 | 7295.441 | 7651.378 | 6039.328 | 723.983 | 5922.833 | 1001.251 |

**Case 2: (mMember = 0.9 | mInsert = 0.05 | mDelete = 0.05)**

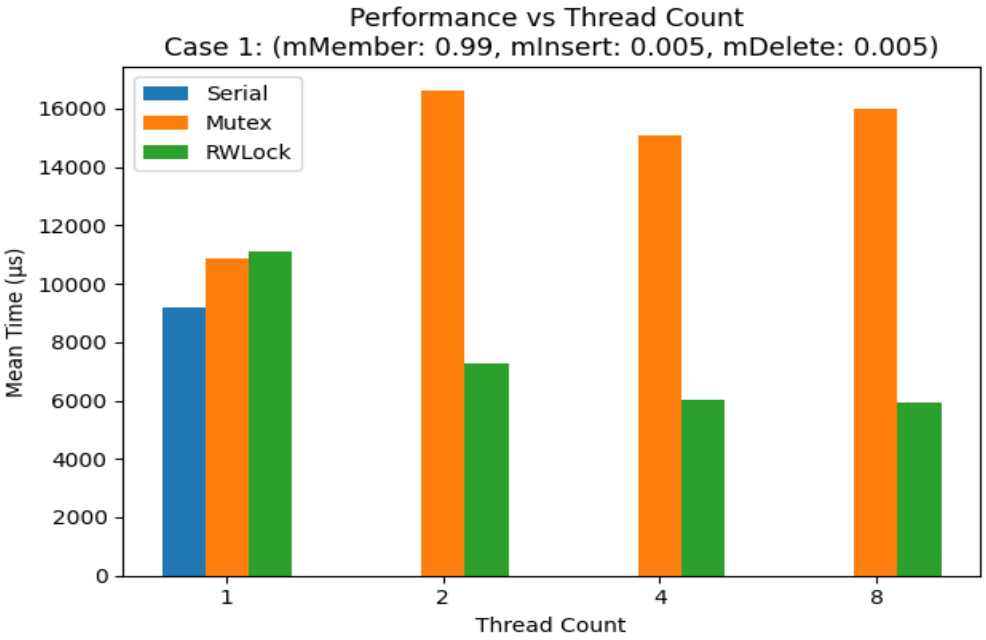
| Implementation | No of threads | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | | 2 | | 4 | | 8 | |
| | Average | Std | Average | Std | Average | Std | Average | Std |
| Serial | 13113.390 | 1034.505 | | | | | | |
| One mutex for entire list | 14199.438 | 5727.334 | 18874.172 | 5855.259 | 16467.105 | 5328.814 | 16550.000 | 3297.917 |
| Read-Write lock | 14875.776 | 5091.362 | 16455.357 | 4608.821 | 16217.687 | 5344.972 | 15410.000 | 3519.340 |

**Case 3: (mMember = 0.5 | mInsert = 0.25 | mDelete = 0.25)**

| Implementation | No of threads | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | | 2 | | 4 | | 8 | |
| | Average | Std | Average | Std | Average | Std | Average | Std |
| Serial | 17533.600 | 812.969 | | | | | | |
| One mutex for entire list | 29236.842 | 7627.735 | 31140.000 | 945.917 | 20216.748 | 6924.795 | 16511.450 | 4890.125 |
| Read-Write lock | 32890.000 | 6417.927 | 42900.000 | 7475.887 | 33570.000 | 4841.435 | 25755.102 | 7837.905 |

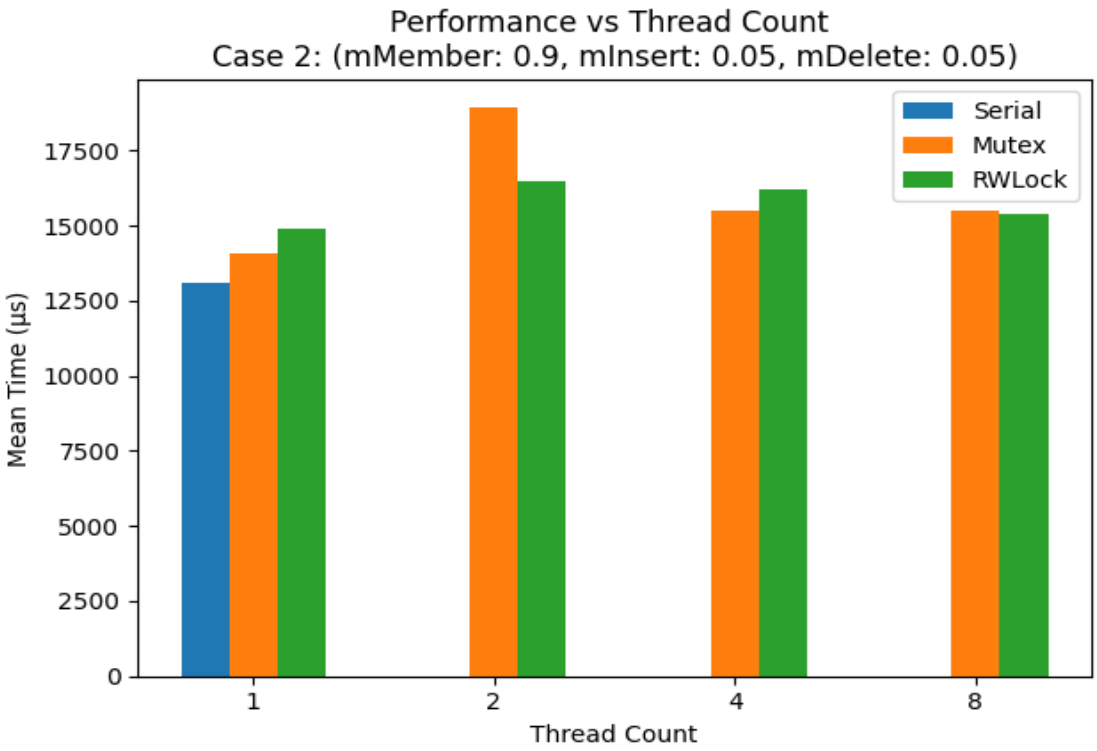
## Step 4

**Case 1: (mMember = 0.99 | mInsert = 0.005 | mDelete = 0.005)**



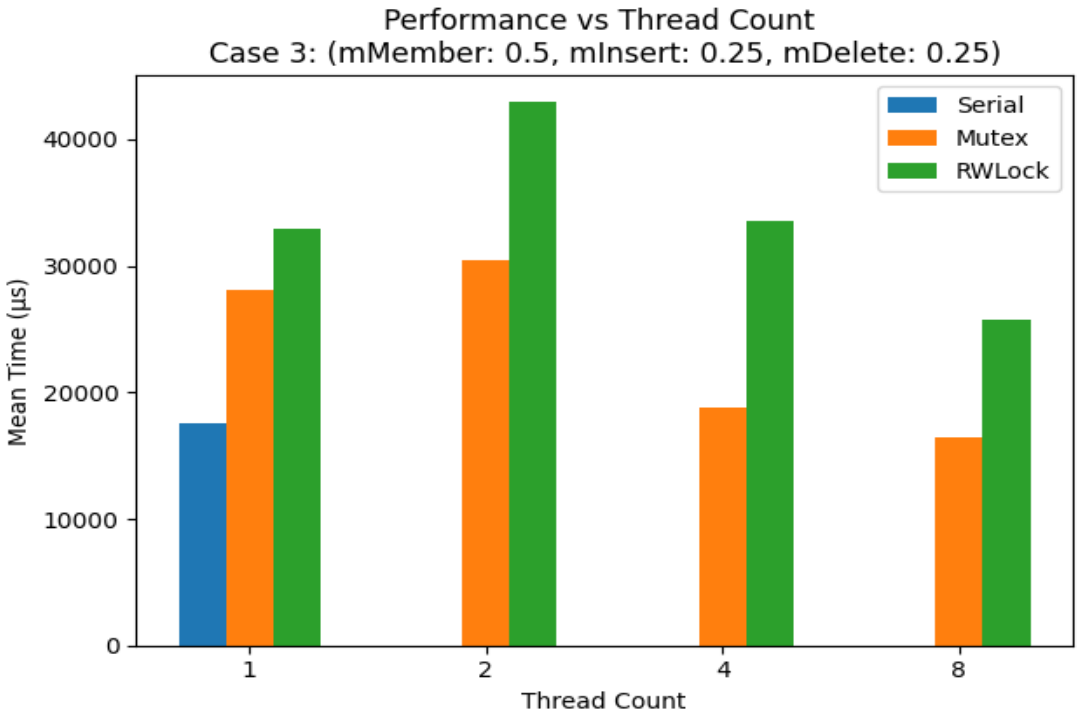

**- Diagram 1 -**

**Case 2: (mMember = 0.9 | mInsert = 0.05 | mDelete = 0.05)**




**- Diagram 2 -**

**Case 3: (mMember = 0.5 | mInsert = 0.25 | mDelete = 0.25)**



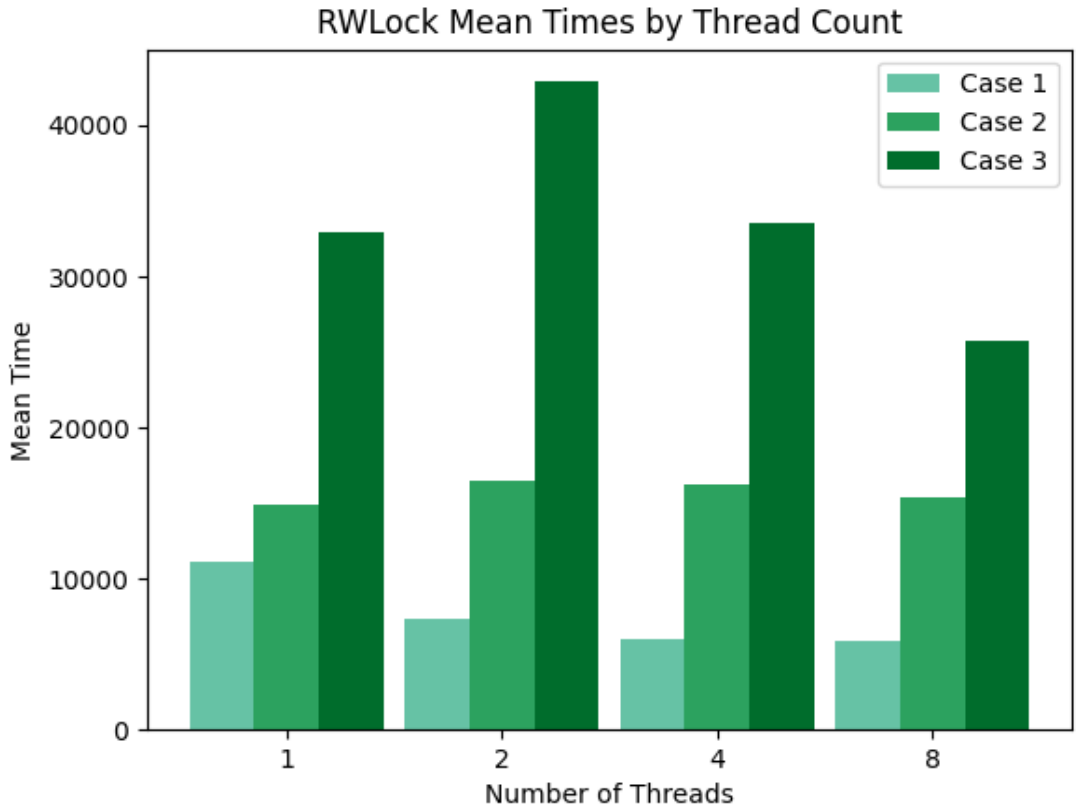

**- Diagram 3 -**

## Step 5

**Serial Implementation**: This implementation typically performs better than the mutex implementation, due to the absence of locking overhead. RWLock implementation performs better than serial implementation in Case 1, which has more Member operations that can be run in parallel (diagram 1).

**Mutex Implementation**: In the mutex implementation, the entire linked list is locked for each operation, causing all operations to execute serially, even when using multiple threads. This approach leads to increased execution times due to the high level of contention and the overhead of repeatedly acquiring and releasing the mutex. In situations with higher concurrency (more threads), the performance often deteriorates, indicating the overhead of frequent locking and unlocking operations.

**Read-Write Lock Implementation**: This implementation generally allows for better read throughput in scenarios with a high read proportion (mMember), as multiple threads can read simultaneously. However, when the number of write operations increases, the read-write lock's efficiency decreases.




**- Diagram 4 -**

The graph displays the performance of the RWLock implementation across three scenarios with varying thread counts.

- When the number of threads increases, the average running time of the RWLock implementation decreases.
- While RWLock performs optimally in scenarios dominated by read operations (Case 1), its efficiency decreases as the frequency of write operations increases in Cases 2 and 3. This suggests that RWLock is most beneficial in read-heavy environments.