# ECE 420 Lab 2 Report

# Christina Ho

# LAB H2

# February 17, 2017

# Description of Implementation

This implementation of a multi-threaded server and client uses sockets to communicate via TCP. The server maintains an array of strings in its memory. The client sends requests to the server to either retrieve or modify a particular string in the array.

After connecting to the server, the client launches 1000 threads that send a request string containing either a read or write identifier and the index to the intended string. The server launches 1000 threads once the client connects to handle the incoming requests.

Each server thread reads the request from the client and parses the read or write request. If the client requests a read, the server will forward the message to the same client file descriptor that it received the request from. If the requests a write, server will modify the given string and echo the new string back to the client. This interaction with the array of strings is the critical section. In the initial implementation, a single mutex is used to control access to the array. Whenever a client requests access to the array, the processing thread on the server will attempt to lock the array from other processes that try to access. Otherwise, it will wait until the process in the critical section unlocks the mutex before it attempts to lock it.

In a second implementation of the server, an attempt was made to optimize the server. Here, instead of a mutex on the entire array of strings, a read-write lock was used to handle concurrent accesses to the array. This allows concurrent reads to array and generally allows more processes to complete if most of the accesses are read operations.

The server will forward the specified string (either read the original or the modified string) back to the client.

# Testing and Verification

In an attempt to verify the correctness of the multi-threaded implementation as well as the mutex implementation, initially the size of the array and the number of threads was set small manageable numbers.

Initially, no randomization is used and only read requests were sent to the server. This was to ensure that the communication between client and server returned the intended string to be accessed in the array.

Next, the number of threads corresponded to the number of strings in the array. Each thread would access a corresponding *rank* index in the array. For example, Thread 0 would access String 0, Thread 1 would access String 1, and so on. Out of 100 threads, a selected 10 would write. The output printed from the server program and echoed back to the client would be verified.

Finally, randomized requests were introduced by verifying the proper ratio of write requests to read requests.

# Performance Discussion

To guide the discussion for the performance of the mutex and read-write lock optimization, each client is run 100 times for each array size of 10, 100, 1000, and 10000 strings for both the mutex and the read-write lock server. The processing time is measured from when the server receives a client request to when it returns an output to the client over all 1000 client requests (i.e. the total processing time of all 1000 client threads).

From the processing times of the mutex at different array sizes, it can be inferred that the mutex performs poorly as the size of array increases. For example, for 100 runs of 1000 client requests, the 90% of all readings are less than 1.5 seconds for 10 strings, 6 seconds for 100 strings, 7 seconds for 1000 strings, and 8 seconds for 10000 strings. As the number of strings increases in the array, the maximum processing time increases.

As we learned in the lecture, implementing a mutex on the entire array would be considered slow. Every access to the array would require operations to lock and unlock the mutex, even if the array contents have not been modified for many processing threads. The probable reason for this is because the accesses to the array or basically performed in series.

On the other hand, from the CDF for the 100 runs using the read-write lock implementation, it is clear that the performance is much faster. For all trials, the CDF yields almost 90% of all processing times is less than 1 second and seems pretty consistent when the size of the array is increased.

As expected, read-write locks decrease the total processing time of 1000 threads because the implementation of two different locks allows for read operations to proceed when the array does not require updates. However when a write operation is in progress, all other threads must block, effectively producing the same effect of a mutex implementation. However, this is still an improvement on the mutex implementation processing times.

Comparing the mean and median processing times of the mutex and read-write lock implementation it is also clear that read-write locks are superior in performance, yielding faster average processing times and more consistent times compared to using a mutex.

An additional implementation was considered where each string in the array would have its own read-write lock. It is likely that having a locks for each element may provide speedup, as a write operation would not block all accesses to the array. However, given that a single read-write lock on the entire array provides sufficient speedup, a read-write lock for every element may introduce additional overhead for locking and unlocking the elements and result in about the same performance increase.

# Conclusion and Experience

In this lab, a multi-threaded server and client were implemented using sockets and handled multiple communication requests through the use of pthreads. Two approaches were used to avoid race conditions when two threads attempted to access the resource array: mutex and read-write locks. Each client launches 1000 request threads and the processing time of all 1000 threads is compared between the two implementations for 100 attempts. The conclusion made for this lab was that read-write locks perform faster than mutexes because they do not require blocking on every thread request. From this lab, the knowledge of handling and debugging multi-threaded socket communication was obtained, as well as the insight of different methods to handling race conditions.

# Appendix

## Macintosh HD:Users:cho:BSc_CompESW:ECE420:Lab:lab2:cdf_10.jpg

Figure 1: CDF Plot of Mutex vs. Read-Write Locks for 10 Strings

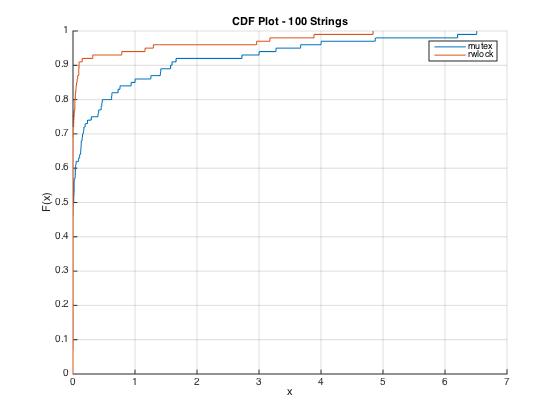


Figure 2: CDF Plot of Mutex vs. Read-Write Locks for 100 Strings

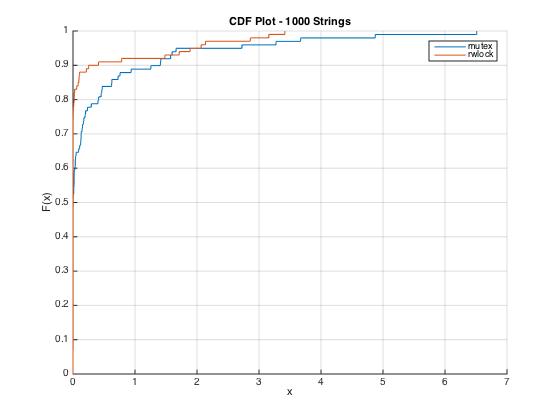


Figure 3: CDF Plot of Mutex vs. Read-Write Locks with 1000 Strings

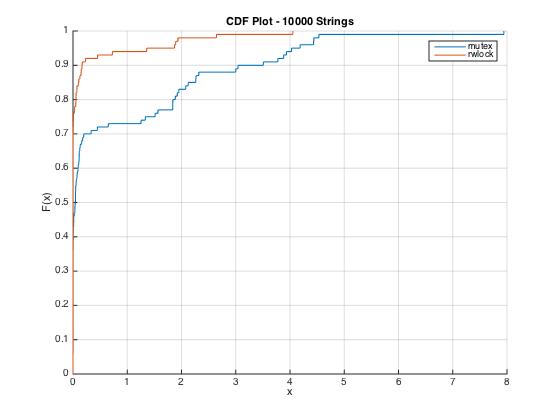


Figure 4: CDF Plot of Mutex vs. Read-Write Lock with 10000 Strings

Table 1: Mean Processing Times (seconds)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **No. of Strings** | | | |
|  | **10** | **100** | **1000** | **10000** |
| **Mutex** | 0.43384829 | 0.45010824 | 0.52520812 | 0.8232163 |
| **Read-Write Locks** | 0.1960247 | 0.19565109 | 0.2102226 | 0.16675949 |

Table 2: Median Processing Times (seconds)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **No. of Strings** | | | |
|  | **10** | **100** | **1000** | **10000** |
| **Mutex** | 0.001773 | 0.0032525 | 0.0085155 | 0.042554 |
| **Read-Write Locks** | 0.0011095 | 0.001203 | 0.001105 | 0.0013765 |