ECE 420 Lab 2 Report

Andrew Juwon Park - 1442834
John Tyler Beckingham - 1500702
Zhi Max Shen - 1494084

Description of Implementation:

1. A single mutex protecting the entire array.

For this server, we used one mutex to protect the entire array of strings. Any thread that wanted to read or write to any string in the array would have to acquire the lock first.

2. Multiple mutexes, each protecting a different string.

For this server, we used an array of mutex locks, such that each string in our string array had a corresponding mutex lock to protect it. Any thread that wanted to read or write to any particular string would have to acquire the corresponding mutex lock first.

3. A single read/write lock protecting the entire array.

For a single read/write lock protecting the entire array, we initialized one read/lock struct for the whole string. After the client request is received by the server, it locks the entire string whether it is a being read or being written to. Then, the read / write lock is released.

4. Multiple read/write locks, each protecting a different string.

For multiple read/write locks that protect a different string, we first initialized and allocated memory for an array of read/write locks corresponding to the number of string lengths. Then, we assigned each of the read/write locks to the specific string index which corresponds to their own index. Then, once a message was received, it locked and released only the specific index of a string.

Read-lock Implementation

For read/write locks, we created our own struct that kept track of the number of readers, pending writers, whether writer was present or not, 2 conditional mutexes, and a mutex for the struct. For initialization, we initialized all the variables to 0 and initialized all the mutexes. For read lock, we first locked the mutex and only allowed reading when all the writers were done and there weren't any more pending writers. For write locks, we waited until nobody was writing or reading, then we added ourselves to the pending writer, and wrote when the writer_proceed mutex opened. For unlocks, it first locks the mutex, checks if there are writers present, we reset the writer count and decrement the

readers count if there were more than 0. Then, if there were more than 1 readers, it opened up the readers_proceed mutex. If there were more than 1 pending writers, it allowed 1 writer_proceed to openand if there were no readers and no writers, it opened up the readers_proceed.

Performance Discussion

The following table is the mean value for the time that any given client request takes in seconds, averaged over 100 runs on each of our four server implementations.

Server Number/ Array Size	**10**	**100**	**1000**
Server 1	1.308e-2	1.276e-2	1.278e-2
Server 2	2.314e-4	1.708e-4	1.657e-4
Server 3	3.804e-3	3.693e-3	4.667e-3
Server 4	2.042e-4	1.710e-4	1.584e-4

There are a couple different things to notice in the above table:

The first being that having a lock for each string seems to be a large improvement over using one lock for all of the strings. This makes sense because in the former case every request has critical section problems with every other request, whereas in the latter case critical section issues only arise when two requests reference the same string index. The tradeoff here is that there is more memory requirement for each of the locks and a slightly higher program complexity in the latter case.

The second thing to notice is that using read-write locks are generally more effective than mutex locks, this is because in the case of read-write locks, many read operations can be executed simultaneously, while with a mutex only one can be. This improvement is immense when we are only using one lock for the entire array, and much smaller when we have a lock for each string. This is because in the latter case there are much less race conditions to begin with, so the improvement will be smaller.