Description of Implementation

1. A single mutex protecting the entire array.

A single mutex was generated to protect the entire array of strings. The size of the string array was taken as a command line argument. Each of the 1000 server threads did the following up receiving a request:

1. Parse the array index, type of request (read or write), and message.
2. (Optional - Only if write request) Lock the array mutex, write the message to the requested array index, then unlock the array mutex.
3. Lock the array mutex, read the string at the index indicated in the request, then unlock the mutex.
4. Respond to the client with the data read from the array.

2. Multiple mutexes, each protecting a different string.

An array of mutexes was generated in order to protect each string. The size of the string array was taken as a command line argument, and was used to initialize both the string array and mutex array of the same length. When a server thread wanted to write to a particular string, if first needed to open the mutex at the same index as the string.

Whenever a client connection was made to the server, it spawned a thread to perform the requested write/read or read operation. The thread would poll the mutex at the same index as the required string, and would block until it was available. It would then perform the operation(s), and return the resulting value at the string index.

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

A single read-write lock, the default pthread_rwlock_t implementation, was used to protect the entire array of strings. The core implementation of the server for receiving and returning responses to requests follows the same as in case 1 where we used a single mutex protecting the entire array, but with a read-write lock instead of the mutex. Other differences include specifically locking as a writer for write requests to the array, and locking as a reader for read requests to allow for concurrent reads but single, protected writes.

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

An array of read-write locks, all using the default pthread_rwlock_t implementation, was used to protect each string of the array. The core implementation of the server for receiving and returning responses to requests, and protecting writes and reads to a

specific string in the array with a specific lock is the same as in case 2 except we use an array of read-write locks in place of the array of mutexes. Following case 3, we also differentiate between read locking and write locking depending on what the request is, as this allows for concurrent reads but single, protected writes.

Performance Discussion

Array Size (n)	main1 average memory access latency (s)	main2 average memory access latency (s)	main3 average memory access latency (s)	main4 average memory access latency (s)
10	1.33E-02	2.21E-04	5.11E-03	1.80E-04
100	1.42E-02	1.60E-04	5.06E-03	1.59E-04
1000	1.45E-02	1.57E-04	5.04E-03	1.56E-04

Table 1. Average memory access latency of 100 memory access latencies of server implementations main1 to main4 for varying array sizes.

Discussion

Overall, we found the ordering of our programs from fastest to slowest (for any array size) to be: main4, main2, main3, main1. We can analyze the performance benefit of protecting the entire array versus each element individually by comparing the average memory latencies of main1 to main2 and main3 to main4. In these two comparisons we see an decrease of 2 and 1 orders of magnitude, respectively. We can analyze the performance benefit of using a mutex versus a read-write lock by comparing the timings of main1 to main3, and main2 to main4. In these two comparisons we see an average increase in performance of less than 1 order of magnitude. It should also be noted that the timing values for main2 and main4 are nearly equal for array sizes 100 and 1000. From these three observations we conclude that protecting each array element individually gives a greater performance boost than using a read-write lock over a mutex. Using a read-write lock over a mutex still provides a performance boost.

One reason for seeing the large increased benefit of providing either a mutex or read-write lock for each of the strings in the array is because all requests have to target the array. Instead of treating the array itself as 1 large critical section, by splitting up the array such that each element has its own critical section, we see the largest benefits in average memory access latency going from 1 mutex to an array of mutexes in main1 to main2, and 1 read-write lock to an array of read-write locks in main3 to main4. We also see the benefits of using a read-write lock over a mutex when protecting a single critical

section that has both reads and writes, as this is due to read-write locks allowing for concurrent reads and locking for single writes whereas a mutex will lock for each read and write. However, this decrease in latency is not seen in going from main2 to main4 where we changed all mutexes in the array to read-write locks. One reason for this is that there aren't enough concurrent requests on each string itself to utilize the benefit of read-write locks over mutexes. If each string within the array also had 1000 read and write requests occurring on it, we may see the same speedup as we have seen in main1 going to main3 by using a read-write lock for each string.