Description of Implementation

rwLock.c

In this file, we placed the lecture implementation of a read-write lock. This file is included in *main3.c* and *main4.c* which use read-write locks for mutual exclusion.

main1.c

For *main1.c*, we used a single mutex for the entire string array. A server thread first accepts and parses the message sent by the client. The thread then locks the mutex before doing any reads or writes to the array and then unlocks it right after. We contained only either the setContent() or getContent() function calls in the mutex to minimize the time a thread holds the semaphore such that other threads can process their requests quicker.

main2.c

For *main2.c*, we use a separate mutex for each element in the string array. We use an array of mutexes of the same length as the string array to store the separate mutexes where the index of a mutex refers to the index of the element in the string array it corresponds to. A server thread accepts and parses the message sent by the client first, then locks the mutex in the same index as the position of the string in the request. The thread is then safe to read from or write to the string to which the mutex corresponds. Similar to the *main1.c*, we contain only either the setContent() or getContent() function calls in the mutex to minimize the time a thread spends in it.

main3.c

main3.c is implemented in a similar way as *main1.c*; we use a single read-write lock for the entire string array. After a server thread parses the message it accepts, it locks only the operation it is performing. Therefore, a write-lock is run only when a thread is writing, and it only contains the setContent() function call and a read-lock is run only when a thread is reading, and it only contains the getContent() function call. This is different from a mutex which does not distinguish between reads and writes and, therefore, contains both the setContent() and getContent() functions calls in the same mutex.

main4.c

For *main4.c*, we used a separate read-write lock for each element in the string array. Like *main2.c*, we used an array of read-write locks the same size as the string array where the index of the lock is the same as the string it corresponded to in the array. The critical section is handled like *main3.c* where the server thread performs a write-lock only when using the setContent() function and the read-lock only when using the getContent() function. Additionally, it only locks the read-write lock of the string the request it is handling is made to.

Performance Discussion on the Lab Machine

The average and median server processing speeds over 100 runs at a given string array size (n) for each protection scheme

[Avg, Median]

	n = 10 (10^{-4}s)	n = 100 (10^{-4}s)	n = 1000 (10^{-4}s)	n = 1 000 000 (10^{-4}s)
Single Mutex(main1)	1.20, 1.18	1.25, 1.26	1.21, 1.20	1.19, 1.18
Multi Mutex(main2)	1.21, 1.21	1.19, 1.21	1.24, 1.24	1.18, 1.18
Single RWlock(main3)	1.18, 1.17	1.21, 1.21	1.21, 1.20	1.18, 1.18
Multi RWlock(main4)	1.22, 1.23	1.21, 1.20	1.19, 1.18	1.16. 1.15

Observations:

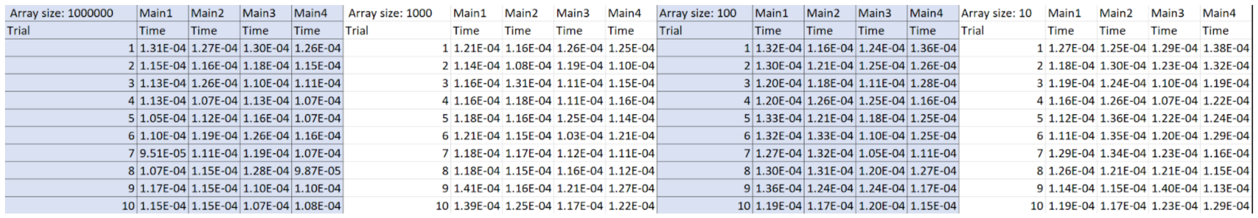
The mutex methods (main1, main2) tend to have slower results compared to their read-write lock alternatives since the mutex cannot distinguish between read and write. The read-write lock distinguishing between the different requests allows reads to execute with less blocking as multiple reads can access the same resource at once.

For the single protection scheme methods main1 and main3, the difference in speeds does not seem to follow the trend of the string array size. This is because there is only one mutex or read-write lock for the whole array. As such, the amount of blocking is not linked to the number of elements in the array, it would be linked to the number of threads attempting to access the array (which was kept constant for the tests). This means that the size of the data being accessed will have no effect on the efficiency of the program, however changing the thread count would change the rate at which each client thread is able to access the data.

For main4, as the size of the array increases, the trend is that its speed increases since the client threads are less likely to request access to the same strings and get blocked compared to if it were to have a smaller size array inputted.

The conclusion is for 1000 client requests with the average memory access, latency does not change much when increasing the string size of the array. We aren't sure if this is due to the overall efficiency of the code, or the machines themselves. As mentioned previously however, it can be seen that main4 does perform slightly better in most circumstances, which holds up to our expectations.

Example of 10 trials in raw data:



Array size: 1000000	Main1	Main2	Main3	Main4	Array size: 1000	Main1	Main2	Main3	Main4	Array size: 100	Main1	Main2	Main3	Main4	Array size: 10	Main1	Main2	Main3	Main4
Trial	Time	Time	Time	Time	Trial	Time	Time	Time	Time	Trial	Time	Time	Time	Time	Trial	Time	Time	Time	Time
1	1.31E-04	1.27E-04	1.30E-04	1.26E-04	1	1.21E-04	1.16E-04	1.26E-04	1.25E-04	1	1.32E-04	1.16E-04	1.24E-04	1.36E-04	1	1.27E-04	1.25E-04	1.29E-04	1.38E-04
2	1.15E-04	1.16E-04	1.18E-04	1.15E-04	2	1.14E-04	1.08E-04	1.19E-04	1.10E-04	2	1.30E-04	1.21E-04	1.25E-04	1.26E-04	2	1.18E-04	1.30E-04	1.23E-04	1.32E-04
3	1.13E-04	1.26E-04	1.10E-04	1.11E-04	3	1.16E-04	1.31E-04	1.11E-04	1.15E-04	3	1.20E-04	1.18E-04	1.11E-04	1.28E-04	3	1.19E-04	1.24E-04	1.10E-04	1.19E-04
4	1.13E-04	1.07E-04	1.13E-04	1.07E-04	4	1.16E-04	1.18E-04	1.11E-04	1.16E-04	4	1.20E-04	1.26E-04	1.25E-04	1.16E-04	4	1.16E-04	1.26E-04	1.07E-04	1.22E-04
5	1.05E-04	1.12E-04	1.16E-04	1.07E-04	5	1.18E-04	1.16E-04	1.25E-04	1.14E-04	5	1.33E-04	1.21E-04	1.18E-04	1.25E-04	5	1.12E-04	1.36E-04	1.22E-04	1.24E-04
6	1.10E-04	1.19E-04	1.26E-04	1.16E-04	6	1.21E-04	1.15E-04	1.03E-04	1.21E-04	6	1.32E-04	1.33E-04	1.10E-04	1.25E-04	6	1.11E-04	1.35E-04	1.20E-04	1.29E-04
7	9.51E-05	1.11E-04	1.19E-04	1.07E-04	7	1.18E-04	1.17E-04	1.12E-04	1.11E-04	7	1.27E-04	1.32E-04	1.05E-04	1.11E-04	7	1.29E-04	1.34E-04	1.23E-04	1.16E-04
8	1.07E-04	1.15E-04	1.28E-04	9.87E-05	8	1.18E-04	1.15E-04	1.16E-04	1.12E-04	8	1.30E-04	1.31E-04	1.20E-04	1.27E-04	8	1.26E-04	1.21E-04	1.21E-04	1.15E-04
9	1.17E-04	1.15E-04	1.10E-04	1.10E-04	9	1.41E-04	1.16E-04	1.21E-04	1.27E-04	9	1.36E-04	1.24E-04	1.24E-04	1.17E-04	9	1.14E-04	1.15E-04	1.40E-04	1.13E-04
10	1.15E-04	1.15E-04	1.07E-04	1.08E-04	10	1.39E-04	1.25E-04	1.17E-04	1.22E-04	10	1.19E-04	1.17E-04	1.20E-04	1.15E-04	10	1.19E-04	1.17E-04	1.23E-04	1.29E-04