ECE 420

Lab 2 Report

Cynthia Li, Michael Kwok, Samuel Lojpur

Description of Implementation

The purpose of the server was to store an array of data that was both read and written by clients. To do that, we first dynamically allocated the array of strings that the clients would be making changes to, using the array size argument provided. The array was initialized with the string value “String i: the initial value” for each index i in the array. The server creates its socket with the provided server ip and port from the program’s arguments before listening for connections. It accepts connections until the number of threads spawn reaches the constant COM_NUM_REQUEST. Finally the server cleans up its own threads, saves elapsed time to a file, and restarts.

Each connection established to the server from a client is handled by a thread handler function in the respective mainN.cpp (where N corresponds to the algorithm from the lab manual). In our implementation of algorithm 1, “a single mutex protecting the entire array”, we statically allocated a single mutex for the server. This was then locked each time we entered the section where we operate on the server’s string array regardless of a read or write. In the second algorithm, “multiple mutexes, each protecting a different string”, we dynamically allocated an array of mutexes with the data table array size provided when the server was being initialized. This was then used further down the chain when the client request was being handled, where the position argument from the client request is used to determine which mutex to lock when proceeding to read or write the string at that position in the server’s string array.

For the third algorithm, “a single read/write lock protecting the entire array”, we copied the implementation of algorithm one, except swapping out the use of the standard mutex with the built in read/write mutex. Instead of calling the mutex before any operations are done to the string array as we did in algorithm 1, we first check for the type of operation before calling the corresponding lock function for the mutex. This allows multiple processes to read the thread simultaneously, or a single process to write. In algorithm four, we combined the implementation of algorithm two and three to create an array of read/write mutexes, and locking only the string in the array that is being addressed, rather than all of them.

Performance Discussion

The average memory access latencies, which are measured by the time between when the server finishes receiving the client’s request to before the server begins sending the response back to the client, is found for each algorithm across 100 runs for n values of 10, 100, and 1000. The tables representing the data are presented below.

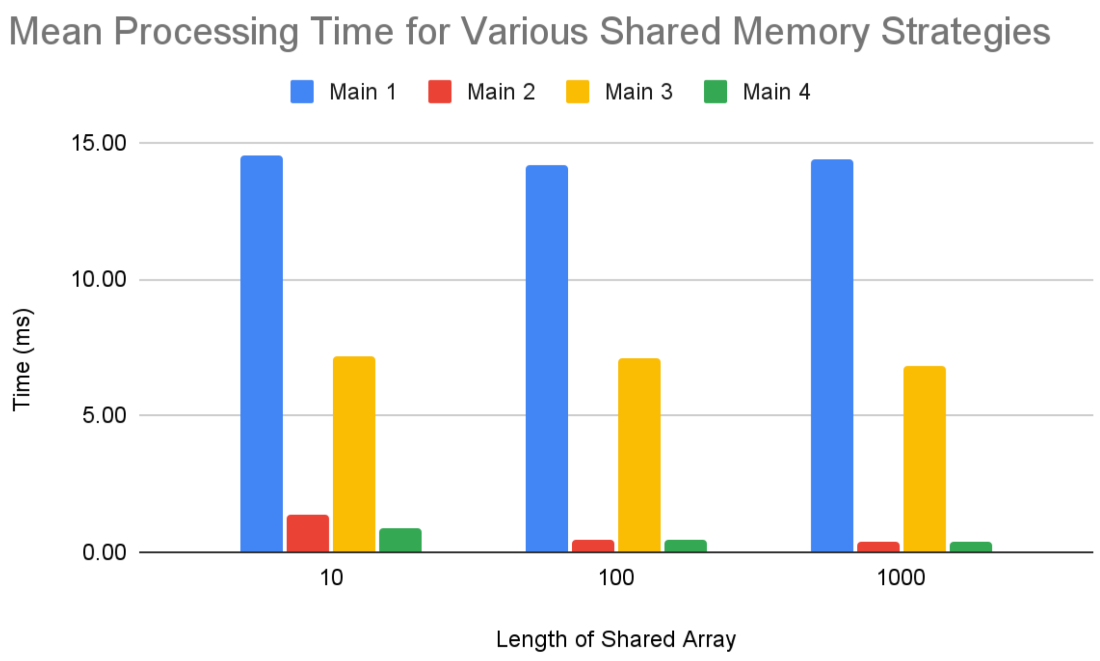
Table 1: Mean memory access latency in ms for each strategy of 1000 client requests run by each of 100 clients

n	10	100	1000
Main 1	1.45E-02	1.42E-02	1.44E-02
Main 2	1.40E-03	4.43E-04	3.73E-04
Main 3	7.15E-03	7.10E-03	6.81E-03
Main 4	8.56E-04	4.26E-04	3.56E-04

Table 2: Median memory access latency in ms for each strategy of 1000 client requests run by each of 100 clients

n	10	100	1000
Main 1	1.45E-02	1.43E-02	1.45E-02
Main 2	1.25E-03	3.78E-04	3.37E-04
Main 3	7.06E-03	7.00E-03	6.79E-03
Main 4	7.63E-04	3.99E-04	3.37E-04

Figure 1: Bar chart demonstrating differences in mean memory access latency in seconds for each strategy of 1000 client requests run by each of 100 clients (based on Table 1)



The data suggests that the use of an array of locks provides the biggest speedup, followed by the use of read/write locks over standard mutexes . This can be explained by the fact that when thread A wishes to access the element at X while another thread is operating on the element at Y, it is more efficient to grant thread A access to element X than to wait until thread B has completed its task on element Y, since an operation on element X would not affect the read or write on element Y. On a similar note, using a read/write lock instead of a single mutex on an element (comparing algorithm 1 and 3, as well as algorithm 2 and 4) also increases the efficiency of the program, since it allows reads to be performed concurrently, only blocking reads when something is being written.

The problem size, in this case, the size of the string array, has a small impact on the performance of the program. There is a decrease in average memory access time across multiple mutex algorithms as the program size increases, due to the fact that the 1000 random requests are more spread out over the array. This decreases the chances that two threads collide on attempting to operate on the same element, thus saving time spent waiting for a mutex to be unlocked. In the case of single mutexes, the trend is less pronounced since there would be no benefit to increasing string array size because the mutex prevents the entire array from being accessed instead of single elements.

If we wanted to further improve performance, we could increase the array size even further, while using the ‘multiple r/w locks’ strategy. We could also look into aligning the array data with cache line boundaries when compiling, which would avoid frequent cache invalidations that might be happening. Also, if we knew the size of our array at compile time, we could allocate space for it on the stack, which would decrease access time.

From this lab, we have seen that keeping mutexes for each element in a multi-element data structure provides potential orders of magnitude speed improvements (in the case of multiple read/write mutexes vs single mutex). However, a drawback of this approach is the extra memory required to store the mutex data structure, as well as the overhead of accessing the required mutex to lock or unlock from the data structure. Nevertheless, as long as this access control method is used only when necessary, it is a great way to improve program efficiency.