ECE 420

Lab 2 Report

Lab Section: H41

February 17th, 2017

By: Quentin Lautischer and Raghav Vamaraju

Description of Implementation:

The server and client code was largely based on the examples given in the Lab 2 Development Kit, and follows the instructions provided in section 3 ("Tasks and Requirements") of the lab manual. Specifically:

1. The server (main_server.c) utilizes pthreads (pthread.h) and sockets (sys/socket.h) to initialize multiple threads and accept connections from multiple clients. Each client is given its own thread and uses its own (unique) file descriptor. The socket address is set to 127.0.0.1, and the port is set to the provided command line argument. The number of threads (number of client connections) is predefined as a constant and set to 1000. There is a for-loop in the server that accepts clients one-at-a-time using the accept() function and spawns a thread to respond to each one, and after accepting all the clients, there is another for-loop that waits for all of them to finish their read/write task using pthread_join(). The ServerEcho() function is called by each thread upon creation.

2. There is a global array of strings ("char theArray[][]") in the server that is modified upon requests from clients. The client does not have direct access to this array. It can only read/write from/to this array by sending requests to the server. This way, any issues relating to race conditions can be handled in the server alone (using a mutex).

3. The client program spawns threads in a similar fashion to the server; there is a for-loop that runs pthread_create() a predefined number of times (1000). After spawning all the threads, the client program waits for them all to complete in a second for-loop using pthread_join(), just the same as in the server code. The Talk() function is called upon spawning a thread. Two random numbers are generated using the rand_r() function, and these numbers modulo-100 are used to determine the position in the array to access in the server, and whether to perform a read or write operation. In particular, if the number is less than 95, a read is requested by prepending "r" to the position in the array to read, and sending this string to the server. Otherwise, a "w" is prepended to request a write operation. The client thread then reads the response string from the server (the array element), prints it to console, and terminates.

4. A mutex (pthread_mutex) is used in the server to lock accesses to the global array of strings. Before reading or writing to the array, the mutex locked. Immediately after fetching the contents of the array at the specified index, the mutex is unlocked. In the alternate server implementation (main_server_slow.c), five mutexes are used, instead of one, to allow for more concurrency and to lock only certain portions of the array. Details of this are elaborated in the Performance Discussion section of this

report.

5. Overall execution times are recorded in both the server and client using the GET_TIME() function from "timer.h". Graphs of the times for various server implementations are attached to this report.

Testing and Verification:

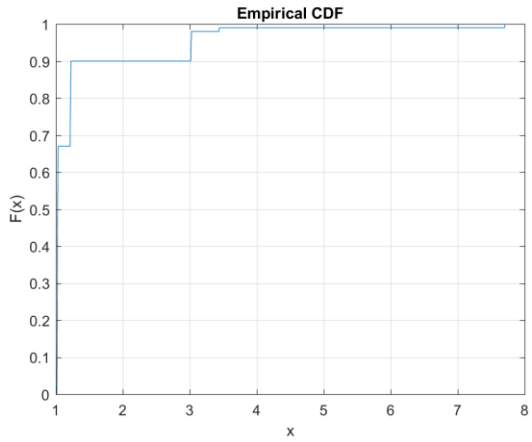
Correctness of the program was done through the use of debugging print statements. We printed the initial array of strings before any clients communicated with the server. We printed the response each client received from the server on the client side. We finally printed the array of strings after all the server-client communication threads had joined. We were then able to cross reference what the clients reported had been done to the state of the array as reported by the server in the end.

We also ran the program through a debugger (in Eclipse) to ensure that the order of execution was as expected between array reads and writes and that the print statements lined up correctly between the server and client. We found that our implementation was correct in this regard.

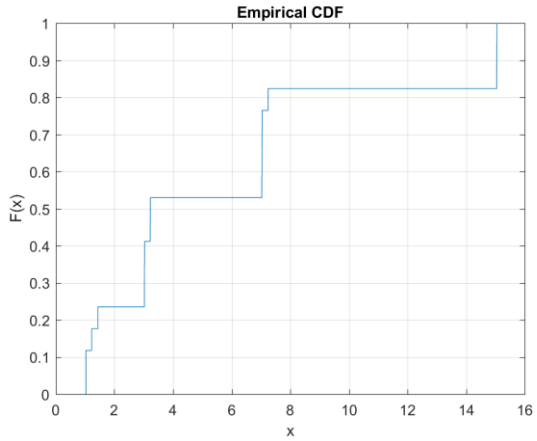
Performance Discussion:

Several versions of the server were created in order to determine the most efficient implementation. Below are CDF graphs from MATLAB depicting running times for the various implementations.

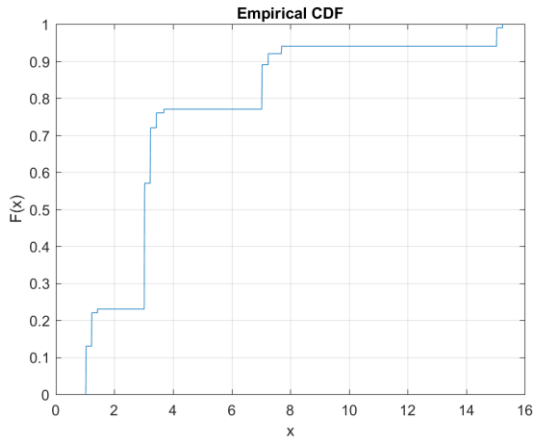
<u>Implementation using 1 Mutex for the entire array:</u>



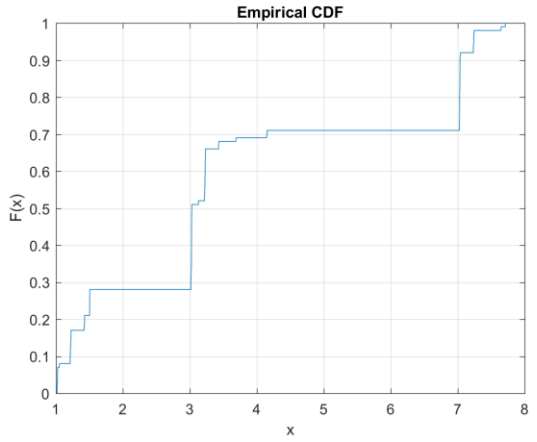
Implementation using 1 Read-Write Lock:



Implementation using 5 mutexes, splitting the array into 5 different sections:



Implementation using 100 mutexes; one for each element in the array:



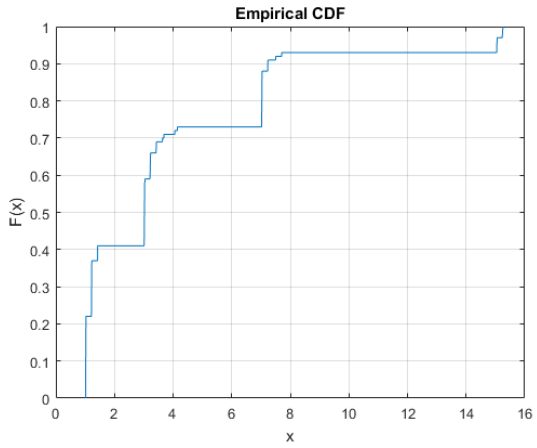
Implementation	Average Time over 100 Runs (seconds)
1 Mutex	1.31927
5 Mutexes	4.06231
100 Mutexes	3.78543
Read-Write Lock	5.92568

Overall, our results clearly indicate that using one mutex for the entire array (of 100 strings), for both reads and writes, is the fastest solution, averaging at ~1.32 second per program execution when using 1000 threads. We found the greatest increase to efficiency when we removed the overhead caused by context switches. To achieve results significantly faster than this, the number of active threads must be limited to 4 threads (one for each CPU core), instead of spawning 1000 threads right away. The results from this test were ~100x faster than the 1000 thread implementation, while still retaining 1000 requests to the server. This speed up occurs because the amount of context switching between threads significantly decreases by reducing the number of threads from 1000 to 4. The overhead from context switching when using 1000 threads is very large, and this skews the execution time drastically. For the rest of the discussion we will ignore the effect of context switches and focus on other domains of multithreaded process efficiency.

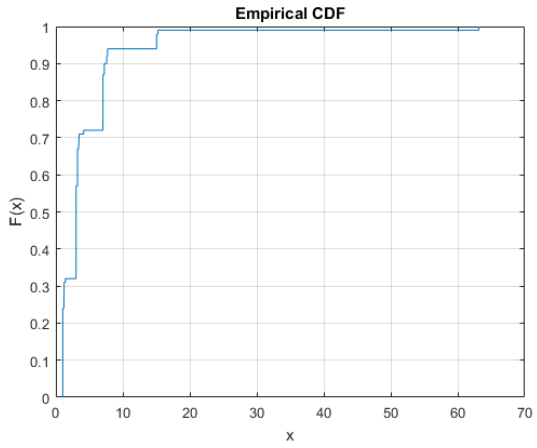
The other implementations that were tested include: a read-write lock (from the lecture notes) instead of a pthread_mutex; 5 mutexes instead of 1; and 100 mutexes instead of 1. All of these implementations were slower than the original implementation that used just one mutex for the entire array. For the 5 mutex and 100 mutex solutions, the average time increased from ~1.32 seconds (using 1 mutex) to ~4.06 seconds and ~3.79 seconds, respectively. Increasing the number of mutexes was intended to reduce contention versus the single mutex, but introduced a new layer of complexity with a global array of mutexes, and this likely significantly increased the number of L3 cache operations. The read-write lock solution likely faced a similar issue; the added complexity from the read-write lock cluttered memory and increased the overhead from false sharing. The average time using the read-write solution was approximately ~5.92 seconds. Some iterations in the tests were fast, but many of them were *very* slow (~7 or ~15 seconds) due to a large number of contentions for the lock. It is likely that the implementation of pthread_mutex_lock() and pthread_mutex_unlock() are expensive, and it seems that due to the large number of threads (1000), the performance became worse by introducing more mutexes.

Performance also decreased as the array size (number of strings) increased. Below are CDF graphs from MATLAB depicting running times for two different array sizes, as well as a summary chart containing average times for a few runs.

Base implementation (1 Mutex) using 10 strings:



Base implementation (1 Mutex) using 1000 strings:



Implementation	Average Time over 100 Runs (seconds)
Array Size 10 (10 Strings)	3.9750
Array Size 10 (100 Strings)	1.31927
Array Size 10 (1000 Strings)	4.5658
Array Size 10 (10000 Strings)	6.1276

Our results indicate that the best performance occurs when the array size is set to 100 (populated with 100 strings). Having less than this (10 strings) resulted in a large amount of contention to read/write the array, thus slowing down the average running time. A larger number of strings in the array (1000, 10000) was also slower than the original 100 string array size. These results may have been an anomaly; they seem to be inconsistent with what is expected. Using an array size of 100 should not have such a large speed increase over all other sizes. Regardless, the performance decrease with the increasing array size is probably a result of increased false sharing between threads.

Conclusion and Experience:

In the end, this lab has demonstrated to us the proof that S(P) > p truly is a fallacy. We see that as a larger portion of the code runs parallel that there is a sublinear speedup. In fact, most likely due to a large number of reasons, we saw a general slowdown in speedup which would indicate there was a great contention for the resourced array. We've witnessed the loss of efficiency brought by the overhead of false sharing within the cache. We found that attempting to optimize such contentions was quite difficult and the performance gain almost negligible. We believe that before you begin optimizing efficiency in cache it is very important to ensure that a program is rid of any needless context switching as this is very costly.

References:

https://mortoray.com/2011/12/16/how-does-a-mutex-work-what-does-it-cost/