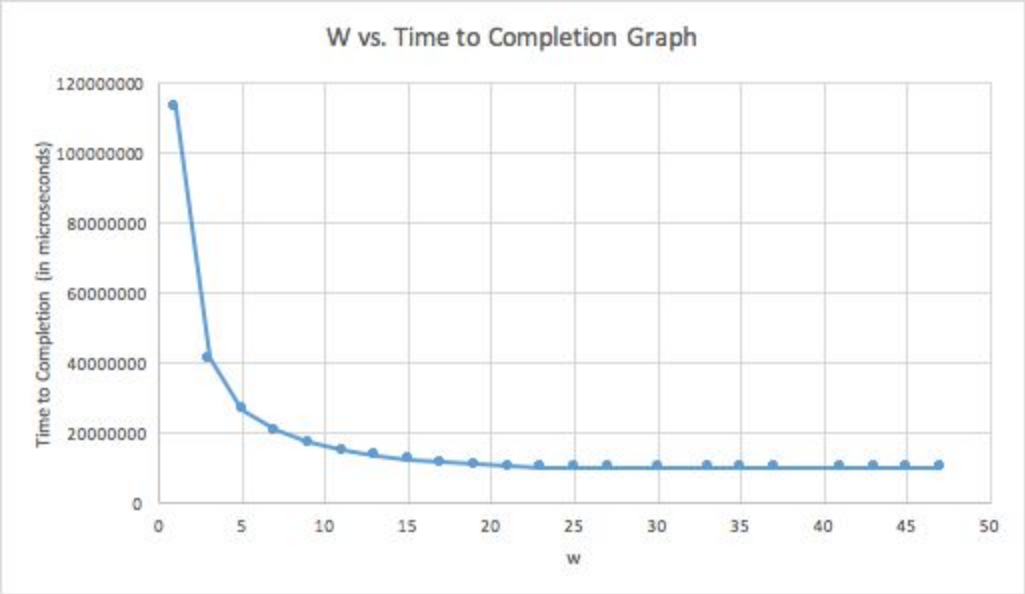
Partner 1: Frank DiRosa
Partner 1 UIN: 223008529
Partner 2: Briana Martinez
Partner 2 UIN: 124001108

MP5 MULTITHREADING REPORT

The objective of Machine Problem 5 is to implement multithreading processes to increase the efficiency and execution time of a client program that makes requests to a dataserver. In the original code supplied, the client program ran synchronously with no threading. In the old code, the first job was to simply create all of the required variables for the objective. Next, the program looped to put data into the request buffer. After that, data was taken out of the request buffer and sent to the dataserver one at a time. Finally, all of the statistics and histograms were populated and the code exited.

To increase the efficiency of the program, multithreading was added to this design. Initialization in the new code now creates new ‘Params’ structures that will be passed in as arguments to the thread functions. Static member variables of this Params structure includes: the data being pushed into the request buffer and the thread locks being used. Params also stores pointers to variables of data that are shared among the different threads, this includes: the request buffer pointer, a pointer to the number of threads and data entries, and the frequency vectors. In the next step of the code, 3 threads are created to populate the request buffer (One for John, Jane, and Joe). These threads execute the ‘request buffer function’ which does a pushback to the request buffer for every new data entry (there are n new data entries for each thread). To avoid corruption, mutex locks are placed when the request buffer class is inserting into its shared data (critical section), followed by an unlock. When all of the requests are inserted, the threads join back to the main. This leads into the next section of the code that executes ‘w’ worker threads. To do this, the threads are created in a while loop and asked to execute the worker thread function. Within the worker thread function, requests are taken out of the request buffer and the statistics vectors are incremented accordingly. Mutex locks/unlocks are added when reading/writing from the shared data memory to ensure that synchronization between the threads is not corrupted. When the worker threads reach the end of the request buffer they will see a ‘quit’ statement and exit. The threads are then joined back with the main function. Next, the new code is to print out the statistics and histograms gathered. Finally, any newly allocated memory that was initialized for the Params structures are deallocated.



From the graph above, it can be seen that as w increases, in general, the time to completion decreases. This can especially be seen with the low numbers of w because the time to completion reduces dramatically with the increase in the number of threads. However, once w hits a certain point it starts to level out because the number of threads the system has to handle outweighs the benefits of using multiple threads. From the data we collected off of the linux.cse.tamu.edu server, this happens at approximately 23 threads. After 23 threads, the program no longer benefits from the addition of more threads because the overhead of managing threads in the kernel outweighs the benefits of adding them.

When running the multithreaded code with only 1 worker process, the result is essentially a synchronous program that executes nothing in parallel. Observing the graphed data points above, the effectiveness of asynchronous calls in comparison to synchronous execution is obvious- the original synchronous program may run MUCH slower than the updated code. Essentially, the updated code base is able to execute multiple instructions (that do not deal with critical sections of data) in parallel so there is no down time for the program.

Using the linux.cse.tamu.edu server, the maximum number of threads that the system allows was found to be 47. If the program tried to create more than 47 threads, the system created files called fifo_control and fifo_data for threads that it could not create. It also printed the error message “p_create failed: Resource temporarily unavailable” for each of the threads past the limit. In regards to the client program when it tried to create more than the maximum number of threads, the client program would not execute and had to be forced closed.