Nathaniel Trujillo
13 November 2021
CSCE 313-598

PA4 Report

The patient threads are implemented by using a function that will create n amount of datamsg objects for p patients. Each of these datamsg objects are pushed to the bounded request buffer for the worker threads to read from. These threads are simply creating requests to be fulfilled by workers. The file thread operates in a similar fashion where it creates a bunch of filemsg objects for the worker threads to carry out. The size and bytes are all determined by getting the file size of the requested file and doing the correct calculations and loops to make a request for everything.

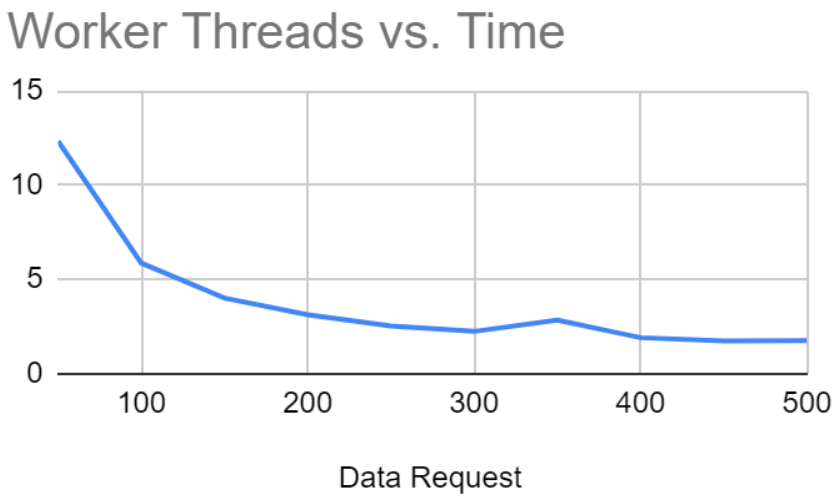
Both the worker threads and file request threads communicate with the server through their own dedicated channels outside of the control channel to prevent any corruption or segfaults.

The histogram was implemented by making a few changes to the code that was provided. The only thing changed on the histogram was an addition of mutex and locks on the update function for individual histograms. This addition was to prevent any race conditions when multiple histograms were used to update the same histogram. As far as the histogram function it was a simple implementation that popped from the bounded response buffer, converted the buffer to an object and read the person and value response. This information was then added into the histogram using an update histogram function that I implemented in the histogram collection class. The histogram function will loop as long as it does not read a quit message from the response buffer. The quit message is put into the response buffer after the worker threads are complete. The histogram function will read a quit message then push it back to the response buffer and terminate. This is so that all other threads can read the quit message and also terminate.

The worker threads were implemented in a similar manner as far as how it receives quit messages and terminates. The difference however is that worker threads will pop from a bounded request buffer and determine if it is a file message or a data message. This is done by using a simple boolean value passed into the function from main. If the function determines the message to be a datamsg then it will convert the buffer into a datamsg object and request its information from the server. Once it gets the information from the server it will create a reply object and store the value and person of the response. This object is then pushed onto the response buffer for the histogram to read. In the case that the worker function determines the message to be a file message it will request the file chunk from the server and write that chunk onto a file using fopen(), fseek(), and fwrite(). fopen() uses a “r+b” permission so that it can handle binary files and also write to the file without deleting any information. fseek() will determine where to write the file chunk to and fwrite() will write the chunk to the file. After the chunk has been written to the file it will simply close the file and keep iterating through all the file requests until it comes

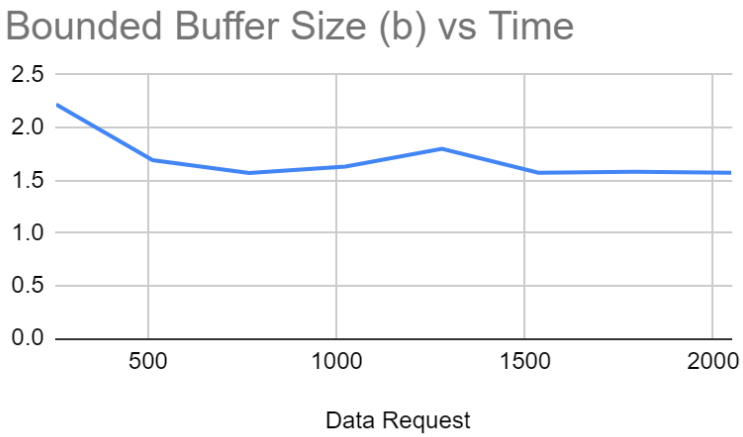
across a quit message that was put there by main(). It will push the quit message back onto the request buffer and terminate. This is so all worker threads know to quit.

Threads are synchronized using mutex and condition variables. The bounded buffer class contains these to help prevent any race conditions from happening when accessing the shared object. It will lock before executing a push or pop and unlock after completing the execution. The condition variables are signaled when an object is able to be pushed or popped depending on the capacity of the queue. These variables prevent overflow and underflow to the queue.

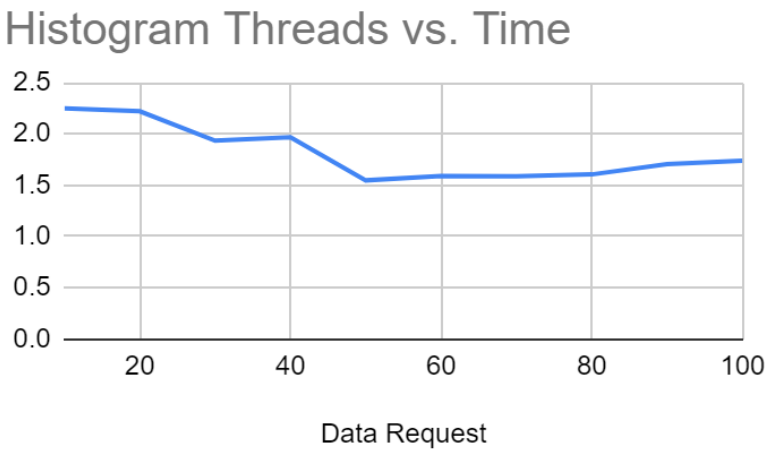


For the graph that illustrates the amount of worker threads vs time it can be seen that there is a dramatic effect where the time exponentially decreases as worker threads increase. This is because much of the heavy arithmetic exists within the worker thread so when the number of possible workers is increased so more are working at a time the execution becomes much faster. This graph we can see scaling since each of the workers can complete multiple data transfers and so it scales exponentially.

Nathaniel Trujillo
13 November 2021
CSCE 313-598

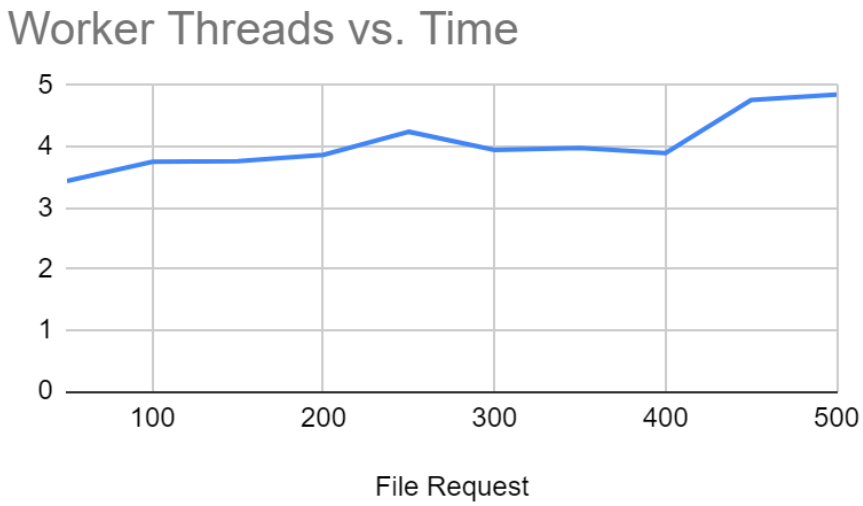


For the graph that illustrates buffer size vs time we can see that there isn't much of a difference going on as the bounded buffer size increases. This is likely because the amount of items that are in the buffer are mostly capable of being dealt with the worker threads and histogram threads. The buffer isn't maxed out for long periods of time and all of the requests get handled consistently. There is no scaling here however it can be seen that there is a slight increase in performance when the bounded buffer can hold more items.

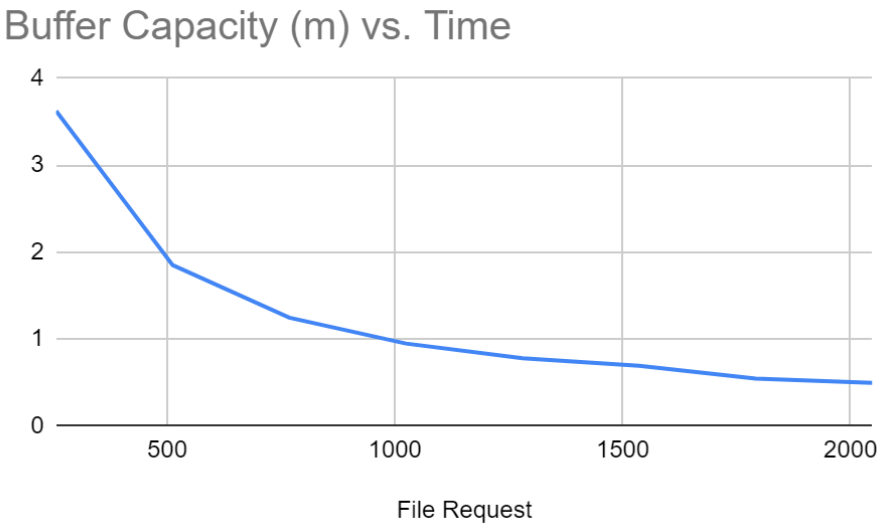


When experimenting with histograms vs time it can be seen that there is a slight increase in performance with more histogram threads. This is because more functions are popping from the response buffer and outputting. The difference is not as noticeable as worker threads because the histogram function itself is not a very time consuming function and is relatively quick already.

Nathaniel Trujillo
13 November 2021
CSCE 313-598



With the increase in worker threads vs time when a file request is being processed it can be seen that performance actually gets worse with more worker threads. This is likely because of the I/O handling with fopen(), fseek(), and fwrite(). When more I/O calls are being made the slower the program is going to execute. This is an expected behavior.



Performance exponential y increases as the buffer capacity between client and server increases. This is because less iterations are required to be made and less I/O requests are being made when the buffer capacity increases. Less I/O means faster execution.

Demo: https://youtu.be/O4OQ-_DsgWQ

Nathaniel Trujillo
13 November 2021
CSCE 313-598

PA4 commit git (also submitted on canvas):
https://github.com/CSCE-313-Tyagi-Fall-2021/pa4-threading-and-synchronization-n-trujillo/commit/977d09a67fc3dbedd23172a79cec49900c6dac07