Programming assignment 3

Threading and Synchronization

Due date

Nov 10, 2023 11:59 PM

Academic Integrity

The following actions are strictly prohibited and violate the honor code. **The minimum penalty for plagiarism is a grade of zero and a report to the Aggie honor system office.**

- Uploading assignments to external websites or tutoring websites such as Chegg, Coursehero, Stackoverflow, Bartleby, etc.
- Copying code from external websites or tutoring websites such as Chegg, Coursehero, Stackoverflow, Bartleby, etc.
- Copying code from a classmate or unauthorized sources and submitting it as your work.
- Sharing your code with a classmate.

Keywords

Multithreading, producer-consumer relationship, race conditions, overflow, underflow, mutual exclusion.

Introduction

In this programming assignment, you will integrate **multithreading** to increase efficiency and improve the runtime of PA1.

While preparing your timing report for PA1, you may have noticed that transferring over multiple data points (1K) took a long time to complete. This was also observable when using filemsg requests to transfer over raw files of extremely large sizes.

This undesirable runtime is largely attributed to being limited to a single channel to transfer over each data point or chunk in a sequential manner.

In PA3, we will take advantage of **multithreading** to implement our transfer functionality through multiple channels in a concurrent manner; this will improve on bottlenecks and make operations significantly faster.

Why Threading?

Notice that the server calls usleep(rand % 5000) upon receiving a datamsg request. This leads to a random processing time for each datamsg request. Since the requests are sequential, one delayed request-response affects the processing time for all subsequent requests. If we wanted to transfer over each data point of the 15 files in the BIMDC directory through datamsg requests, we would have to make multiple requests for each file sequentially; naturally, this would result in a long time to execute.

One way to collect data faster from these files is to use **15 threads** from the client side, **one for each patient**. Each thread would create its own request channel with the server and then independently collect the responses for each file as well. Since the server already processes each channel in a separate thread, you can get at most 15-fold speed over the sequential version. This technique is shown in Figure 1. Note that file requests can be sped up similarly.

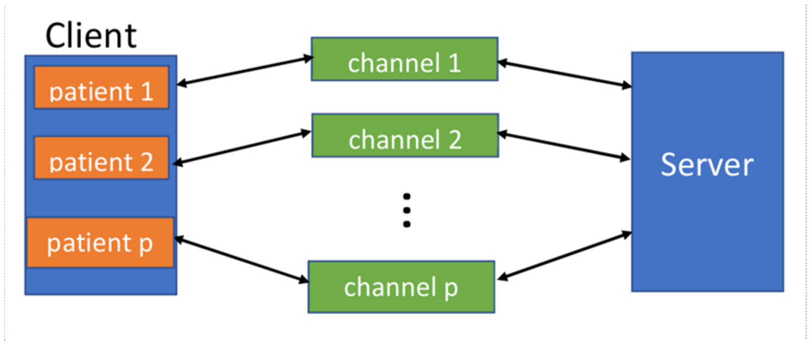


Figure 1: First approach to scaling PA1 performance - one thread per patient

However, there are two major issues with this approach:

- The speedup is always limited to p-fold, where p is the number of patients.
- Since each request takes a random time, some patient threads may take longer to complete transfers than others.

To avoid these roadblocks, we have to make the number of threads processing these requests independent of the number of patients, p.

The standard solution to this problem is to separate the tasks of producing these requests (i.e. creating the datamsg/filemsg object) and processing them (i.e., sending them to the server). We can think of this model as a **producer-consumer relationship**. We use p threads to produce the request objects (one thread per patient), and use w number of “worker” threads that read (or consume) these requests and send them to the server. The theoretical speedup factor is now w, which is independent of p, and if w >> p, we have achieved significant speedup. This way the number of patients p can be decoupled from the number of threads (w) that would be in charge of communicating with the server.

All that is left is to design a mechanism that allows the w number of worker threads to read the request objects being produced by the p patient threads. We can use a buffer that can be thought of as an STL::queue to implement this mechanism. **The p patient threads push the requests onto the buffer, and the w worker threads pop each request**. Each worker thread can now communicate with the server independently through its own channel. Figure 2 demonstrates this technique.

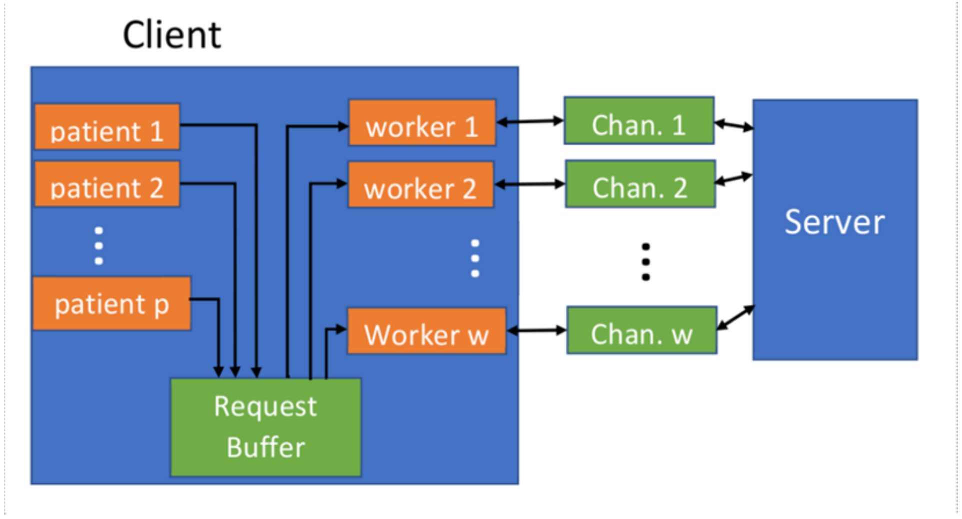


Figure 2: Second try with a buffer - number of worker threads w is now independent of the number of patients p

For this technique to work, you need a special and more complicated buffer than just an STL queue. First, the queue must be thread-safe; otherwise, simultaneous accesses from the producers (i.e., patient threads) and consumers (i.e. worker threads) would lead to a **race condition**. Second, the buffer must be made "bounded" so that the memory footprint of the buffer is under check and does not grow to infinity. In summary, the buffer must be protected against race conditions, and **overflow** and **underflow** scenarios.

An overflow happens when the patient threads are much faster than the worker threads

An underflow happens in the opposite scenario.

The BoundedBuffer class is the perfect solution to all these problems.

Client Requesting Data Points

You can follow the below procedure to request data points for p patients:

- The p patient threads place datamsg requests into the **request buffer** (a BoundedBuffer object). Each thread places requests for one patient (i.e. first thread for patient 1, second thread for patient 2 etc.)
- The w workers threads pop requests from the request buffer, sends the requests to the server, collects the response from the server, and then puts the response in the **response buffer** - another BoundedBuffer object.
- The h histogram threads pop these responses from the response buffer and update p ecg histograms. A HistogramCollection object consists of p Histogram objects. A Histogram object keeps track of a particular patient's statistics (ecg values).

Note that multiple histogram threads would potentially update the same histogram leading to another race condition, which must be avoided by using **mutual exclusion**. This use of mutual exclusion has been implemented in the starter code (in the Histogram Class).

In order for the histogram threads to know which response is for which patient, the worker threads must make sure to prepend each data response with the respective patient no. You can do that by making a pair of the patient number and the data response (i.e., using STL::pair or a separate struct/class with 2 fields). Figure 3 shows the structure to follow.

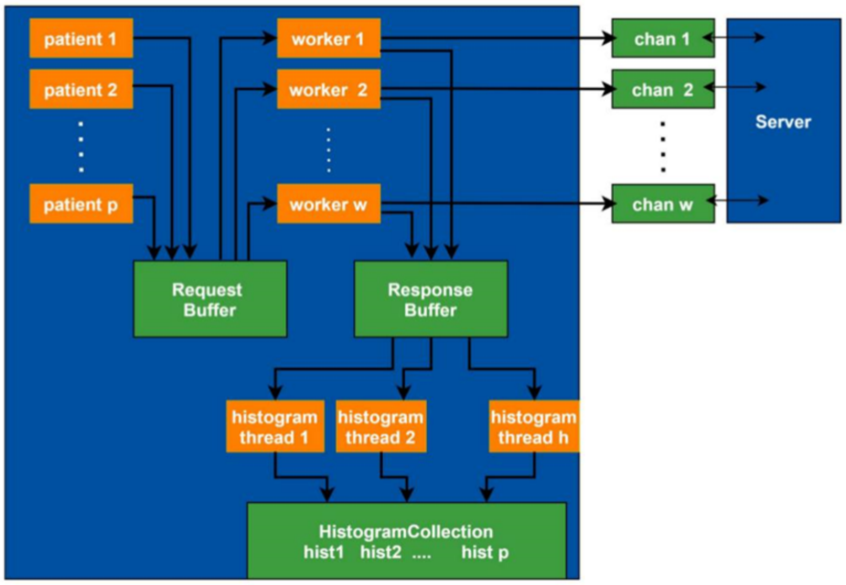


Figure 3: Structure for data requests

When requesting data messages, the client program should take in 4 command line arguments: n for number of data points requested per file (range - [1, 15K]), p for number of patients (range - [1, 15]), w for number of worker threads (range - [50, 500]), h for number of histogram threads (range [5, 100]), and b for bounded buffer size in number of messages (range - [10, 200]).

For instance, the following command is requesting 15K ecg data points for the first 10 patients using 500 worker threads and a request buffer of size 10. It will use a separate 10-message response buffer to collect the responses and 5 histogram threads to make the 10 patient histograms for ecg values.

```
$ ./client -n 15000 -p 10 -w 500 -b 10 -h 5
```

Default values of each of these variables are instantiated in the starter code. Note that all these command line arguments are optional, which can cause their default values to be used.

Notice that there is a total of p + w + h threads in just the client program: p patient threads, w worker threads and h histogram threads. All these threads must be running simultaneously for your program to work; otherwise, the request and/or the response buffers will stall after reaching their bounds (buffer full) or after running dry (buffer empty).

You cannot just use a large request buffer where all requests would fit. Make sure to test your program using a small request/response buffer size (e.g., b = 1); your program should work perfectly fine, albeit a bit slower.

Smaller b values, along with high p, w, n, h, increase concurrency and thus manifest race condition bugs that are not visible under easier circumstances. Make sure to stress-test your program under these settings.

Note: When asked to plot histograms for n data points and p patients, you are expected to plot the first n ecg1 values for each person. You should NOT transfer and use the ecg2 values to update your histograms.

Client Requesting Files

You can follow the procedure below to request a file:

- Client queries file size from the server, like in PA1.
- Client starts one thread that makes several filemsg request objects for each chunk of the file and pushes these requests onto the request buffer.
- The w number of worker threads pop these requests from the request buffer and send the requests to the server. The worker threads receive the response and write each chunk of the file to the appropriate location. Each of the w worker threads communicates with the server using its own dedicated channel.

Note that unlike requesting data messages, there is no response buffer in this case, only a request buffer. Also note that while the program is running, there are w + 1 total threads working simultaneously: 1 thread for making the requests and pushing them to the request buffer, and the rest w worker threads who keep reading these requests from the request buffer and processing them. The structure is shown in Figure 4.

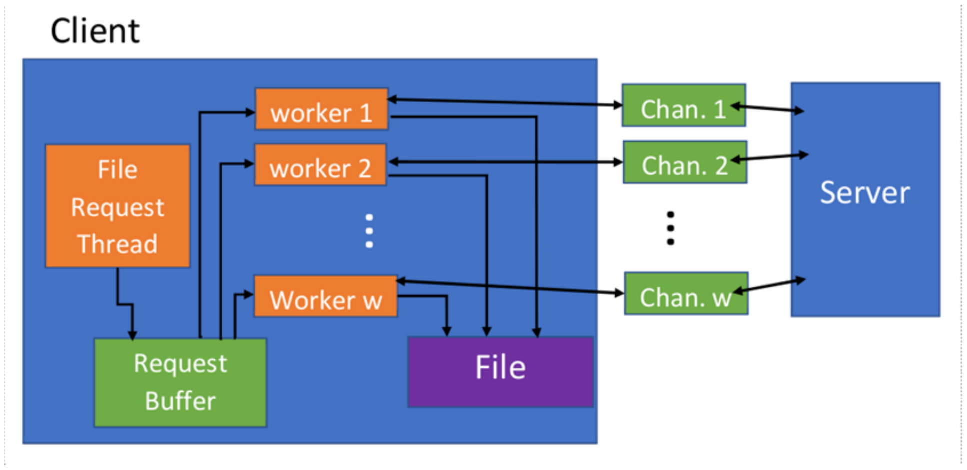


Figure 4: Structure for file requests

Note that in this design, file chunks can be received out-of-order (earlier chunks arriving later or vice versa). You must make your worker threads robust such that they do not corrupt the file when they are writing to it simultaneously. You will find the function fseek(...) useful here. There is a specific mode for opening a file that would make this possible.

When requesting a file, the client would take 4 command line arguments: w for the number of worker threads (range - [50, 500]), b for request buffer size (range - [10, 200]), m for buffer capacity (range - [64, 4092]) to keep the file content in memory, and f for file name.

The following example command asks to transfer the file “file.bin” using 100 worker threads and a buffer capacity of 100 messages. The three arguments (w, b, and m) are optional.

```
$ ./client -w 100 -b 100 -f test.bin -m 256
```

It is important to note the difference between the b and the m flag. The b flag indicates the size of the request buffer (a BoundedBuffer object), whereas the m flag indicates the maximum capacity that the server can respond to the client through the communication channels.

Implementing the BoundedBuffer

BoundedBuffer must use an **STL queue** of items to maintain the First-In-First-Out order. Each item in the queue must be type-agnostic and binary-data-capable, which means **you cannot use std::string**. Either **std::vector<char>** or some other variable-length data structure would be needed.

The BoundedBuffer class needs 2 synchronization primitives: **a mutex** and **two condition variables**.

You should not need any other data structures or types. Use std::mutex from the standard C++ library as your mutex and std::condition_variable from the standard C++ library as your condition variable. You will need one condition variable for guarding overflow and another one for guarding underflow.

The following procedure shows how the producer and consumer threads coordinate interaction through the BoundedBuffer:

- **Each producer thread waits for the buffer to get out of overflow** (i.e., buffer size is less then the maximum) before pushing a request item. It also notifies the consumer threads (i.e., worker threads) through the condition variable guarding underflow that data is now available. This wakes up all waiting consumer threads (if any) one at a time.
- **Each consumer thread waits for the buffer to get out of underflow** (i.e. buffer size is non-zero) before popping an item. It also notifies the producer threads through the condition

variable guarding overflow that there is space to push requests. This wakes up all waiting producer threads (if any) one at a time.

HistogramCollection and Histogram Classes (Only for data point transfers)

The Histogram Class represents a histogram for a particular patient defined by the number of bins between a start and end value. This start and end value should be defined such that it covers the range of the ecg values for any patient.

For example, if we wanted to instantiate a Histogram for a particular patient with 10 bins, of range -2 (start) to 2 (end) we would:

Histogram* histogram = new Histogram(10, -2, 2);

A HistogramCollection object stores each Histogram object (for p patients). In the starter code, you will see Histogram objects being added to a HistogramCollection object using the add(...) function. You will need to use the update(...) function to update the Histogram corresponding to person p with an ecg value. The print function would print the p Histograms to stdout if you implement your functionality correctly.

Getting started

1. Go to the assignment's GitHub classroom: https://classroom.github.com/a/WXx8eWSm
2. Create a repository for your project.
3. Watch the start video: https://www.youtube.com/watch?v=HbRhkJiIgjo

Code Requirements

The starter code includes files from PA1 (i.e., server.cpp, client.cpp, common.h/.cpp, and FIFOreqchannel.h/.cpp). In addition, it now includes Histogram.h/.cpp and HistogramCollection.h/.cpp. Do not modify the Histogram-related classes.

The package also contains a template for the BoundedBuffer class (.h/.cpp) that you have to fill out and use properly in the client.cpp file.

Your code must also incorporate the following modifications compared to PA1:

- Your client program should accept all the command line arguments: n, p, w, b, m, f, and h. Based on whether the f argument was provided, the client chooses to request data or a file. All the other arguments are optional.
- Start all threads (e.g., p patient threads, w worker threads, and h histogram threads) and wait for the threads to finish. Time your program under different settings and collect runtime for each setting.

- For data requests, your client program should call HistogramCollection::print() function at the end. If your program is working correctly, the final output should show a histogram of n data points for each person.
- Your program should include a functional BoundedBuffer class that is thread-safe and guarded against overflow and underflow.
- The server should accept another argument m for buffer capacity, which should be passed along from the client.

Rubric

1. Implement BoundedBuffer class [40 pts]
    a. Your implementation cannot have a Semaphore class. Using Semaphores will result in a loss of 15 points. You must instead use std:mutex and std:condition_variable.
    b. A unit tester has been provided to verify the correct implementation.
2. Implement data requests and reporting [30 pts]
    a. This involves implementing the patient threads, the histogram threads, and the datamsg component of worker threads. Results will be verified by correct counts on the histograms displayed on the terminal.
3. Implement file requests and transfers [30 pts]
    a. This involves implementing the file threads and the filemsg component of worker threads.
    b. Both CSV files and binary files will be tested, with varying message capacity.
    c. diff will be run on the original file in the BIMDC/ directory compared against the file in the received/ directory.

QA

1. For condition variables, are notify_one() and signal() the same thing? it says signal on the lecture slides but notify_one on cppreference website.

notify_one is used by cpp for signal (which unblocks one of the threads that was blocked by the condition variable on wait). notify_all is used for broadcast (which unblocks all threads that were blocked by the condition variable on wait). You must use notify_one in this PA.

2. Since there is only one histogram per patient, does that mean we will only use the ecg 1 or 2 values? Or will each histogram handle both values?

You should only be using ecg1 to update your histograms.

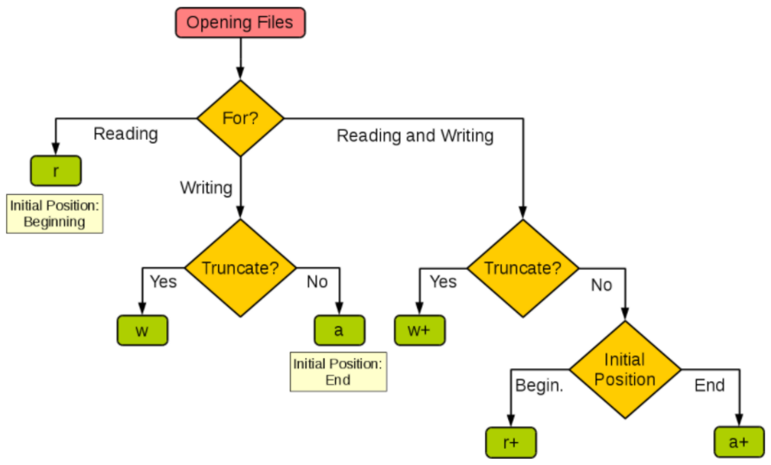
3. Should I release the lock before I notify or notify before I release the lock?

Generally, it shouldn't matter which way you choose to go. But, for the purposes of minor optimizations - if you notify before you release the lock, it can lead to the notified thread being blocked again because the notifying thread still holds the lock. Therefore, unlocking before notifying is the preferred method to use.

4. Are there any tips for where to find the specific way of opening a file that will allow multiple threads to write to it without corruption?

In the file request thread, you would allocate the file in memory (by opening in write mode then using a seek operation to touch all bytes that need to be allocated). The seek operation in the file request thread just tells the kernel how large the file will be.

Then in the worker threads, when you process one of the file message requests from the request buffer, you would open the file in update mode, then **use the offset to set the seek location to the appropriate place to write the response**. As long as the file message requests are well-formed and you reposition to the correct byte to start writing at, you shouldn't encounter a conflict.



Follow this diagram in order to help you determine the update mode that you will use.

Note: Truncate means that you will empty the contents of the file and then write. We don't desire this.

5. How would a worker thread know when it needs to stop reading from the bounded buffer?

You would push an equal amount of quit messages as there are worker threads to the

request buffer after joining the data/file threads. Then when the worker thread reads a quit message, it would send it along its channel, then end its execution. You can now call join on the worker threads.

6. How do I convert a vector<char> to a char* for popping from the BoundedBuffer? And similarly, how do I convert char* to a vector<char> for pushing into the BoundedBuffer?

Use vector::data() to help you copy over the contents of vector<char> to char*.

For converting char* to vector<char>, you can call the vector constructor in a certain way that would allow you to do this. The reference link here will be useful (you can look at the example code for hints).

7. I see different ways to use the condition variable: the busy waiting in the while loop as well as the predicated version using lambda expressions. Which one should I use?

Either one works just fine, it is up to you to choose which style you want to implement. You can find more information here

8. My program breaks when I call join(). What is the reason for this?

It's usually because your functions are stuck in an infinite loop and have not completed their functions because of not receiving the QUIT_MSG and terminating upon receiving it.

9. I'm running into some problems with the filemsg transfer. Any tips on potential points I've overlooked?

Make sure you are calling fclose(...) in your file_thread_function after opening it and using seek to touch all bytes necessary. This allows the system to flush all data in the disk intended to be written to the file.

Code Organization

BoundedBuffer

Any thread calling BoundedBuffer::push must acquire a shared mutex to prevent race conditions (as does any thread calling BoundedBuffer::pop). It must also use the condition variable to avoid situations where the queue size is greater than the max capacity. How do we use the condition variable along with the mutex? Condition variables accept std::unique_lock, which tracks the locked status of the mutex. Calling the unique_lock constructor with a shared mutex as its parameter will immediately lock the mutex. Any thread calling BoundedBuffer::pop can then use a new unique lock object passing in the shared mutex in its constructor.

Client

The ideal code structure behind PA3 can be broken down into the below control flow:

1) Is the client tasked with a multiple data point transfer?

If so, start the patient, worker, and histogram threads simultaneously to enable concurrency.

   a) The patient threads run a loop terminating after the nth iteration

b) The worker threads run a loop that keeps processing requests from the request buffer till indicated by a QUIT_MSG that all requests have been processed. The worker threads can cast the popped buffer to a MESSAGE_TYPE to determine the type of request.

c) The histogram threads run a loop that keeps processing pairs/structs containing the person and the corresponding ecg value from the response buffer till all n values of each of the p patients have been processed; the histogram threads break upon some special pair/struct member value.

How will you organize your code such that the worker threads will receive its QUIT_MSG only AFTER all datamsgs have been popped out? Similarly, how will you ensure that your structure ensures that all histogram threads receive their special pair value only and only AFTER all legitimate pair/struct values have been processed?

Notice that the std::thread::join() function causes the calling thread (in this case - the main thread) to suspend execution till the thread on which join is called completes execution. There are two things we need to keep in mind when instantiating our threads:

- We do not want the main thread to finish execution before all our instantiated threads have completed execution. Solution? Call join on all threads instantiated.
- At the same time, if we call join on the workers and histogram threads, an undesirable situation arises - if all our patient threads have completed, main resumes execution (after all patient threads have been joined), and calls join on the worker threads; the main thread will be waiting for worker threads to complete, but the worker threads are waiting on the condition variable that the request buffer size be non-empty. This causes the program to hang. How do we resolve this situation?
  - Hint: We know that all patient threads have pushed their requests into the request buffer if the main thread has resumed execution after calling join on all patient threads. If we could use our main thread to push in QUIT_MSGs into the request buffer after all patient threads have completed, we can ensure that worker threads can see these QUIT_MSGs and terminate only after all datamsgs have been processed because of the FIFO property of std::queue! Each worker thread must receive the quit message, so you must push in the right number of QUIT_MSGs. You must follow a similar design for histogram thread termination, except in this case the histogram threads have to break only after all worker threads have pushed to the response buffer.

If not a data point transfer (file transfer case):

Start the file thread and worker threads simultaneously to enable concurrency.

a) Within the file thread function, use a new channel to obtain file size and push the appropriate number of filemsgs (according to the “m” buffer capacity value) to the request buffer

b) The worker threads would determine the type of request and use its own independent channel to communicate and receive the response of the filemsg from the server.

You must follow the above-mentioned design implementation to ensure that all worker threads break.

Test-plan Development

Debug small portions of your code incrementally.

You can start off with your BoundedBuffer implementation.

Some common issues in relation to the BoundedBuffer are:

- Condition variables have been given the inverse condition
- std::system_error - this is due to calling lock on an already locked mutex. As mentioned earlier, when you feed in a mutex to the unique_lock constructor, it automatically locks the mutex.
- BoundedBuffer test cases pass, but when values of a popped out datamsg are accessed, only garbage values are being read. Some possible reasons include:
    - vector::data is a pointer, so calling sizeof(vector::data) would ALWAYS result in 8 bytes because a pointer stores an address.
    - In the past, students have fed the address of a pointer to memcpy which has resulted in a double pointer being passed in as the argument. You should only be passing in the address of non-pointers.
    - Using memcpy to convert from vector<char> to char * ;memcpy copies till null termination (many of which exist in a datamsg/filemsg). Therefore, not all sizeof(datamsg/filemsg) bytes will be copied over which results in a garbage value being read.

Implement each push and corresponding pop, and test whether the popped elements can be accessed correctly (i.e. Patient-Worker relationship, Worker-Histogram relationship separately). For example, if you test your program after completing the patient and worker thread functionality without the histogram functionality, your code might end up working fine, but you will not be able to test it successfully; the worker threads will wait on the histogram threads to pop out messages from the response buffer, which means that they will not be able to pop out from the request buffer and the client will give you the illusion that it is incorrectly hanging. If you are unable to test for a single push-pop relationship due to not having other required parts like join complete, you can consider incrementing and printing out the value of some global count variable (protected by a mutex) that keeps track of the # items popped out.

While coding your client, you may run into some of the following issues:

- Server could not read anything: The server, upon its call to cread(...) read 0 bytes which resulted in this error. This is likely because of some incorrect message sent to the server or a wrong sizeof() call, which could possibly break the connection. Have you declared any one of your MESSAGE_TYPE variables as m? This could result in the value of "m" the buffer capacity changing. Are you sending a QUIT_MSG across all new channels established?

- Client is requesting a chunk bigger than the server's capacity: Have you hardcoded the buffer capacity to MAX_MESSAGE? If the command line specifies a flag changing the "m" buffer capacity, a discrepancy arises between the capacity that the server expects and MAX_MESSAGE.

Oftentimes you may run into the below issue specific to file transfers when your program hangs/finishes in between for no apparent reason. A common issue is using the control channel in their file thread function to get the file size. Upon running GDB, you may notice a SIGPIPE error as well.

Consider the scenario: the file thread is scheduled, it uses the control channel to write a filemsg asking for the file length from the server. It is then context switched out - the main thread is scheduled, which uses the control channel to send NEWCHANNEL_MSGs to the server to create the new channels for each worker thread. The main thread then calls cread(...) but undesirably reads both the response meant for the file thread and the name of the new channel.

Common memory issues indicated by AddressSanitizer:

- Heap use after free: You are trying to access an object on the heap that has already been deallocated.

- Heap buffer overflow: You're trying to access the object at some area/index beyond its allocated capacity.

- Memory leak: First ensure that you have a delete (deallocation) for every new (allocation) [This excludes memory on the heap that is cleaned up by the destructor]. Note that the char* passed into BoundedBuffer::push is copied over to vector<char>. Therefore, if the object you use as this parameter is allocated on the heap within the client, you must deallocate it once used. You must also use the [] operator when deleting arrays on the heap. Also ensure that you are deallocating any dynamic memory before you break from a loop, if necessary.

Note: Address Sanitizer clearly indicates the line at which the error originates. A simple CTRL+F for client.cpp within your terminal when debugging an ASan message can help you isolate the location at which the error occurred in the client.

Coding Dos and Don'ts

Dos

You can follow any one of the below options to pass in parameters into your threads:

To avoid passing in a copy of BoundedBuffer, FIFORequestChannel and HistogramCollection objects, we must take these parameters as pointers or references. You can use one of the following options to achieve this:

If the object is declared on the heap or we have a pointer to an object on the stack, then our parameter must accept a pointer, and we can pass in this pointer as is.

If the object is declared on the stack (and is a non-pointer):

If the parameter accepts a reference - you must wrap your object with a std::ref()

If the parameter accepts a pointer, you can simply pass the address of the object.

Don'ts

The following points are crucial to keep in mind when instantiating threads:

- If you instantiate your objects/variables on the stack within some form of control structure (loops or if/else) and then pass it on as parameters to the thread, these parameters will go out of scope and the threads will not be able to use them anymore.

- If you instantiate a thread and then push it into the vector later, this vector will be pushing a copy of the thread rather than a reference to the thread. You must either push it during the thread instantiation or use std::move to store the actual instantiated thread in the vector and not a copy.

Some common misconceptions with the thread functions include:

- Do not break your worker threads on the condition that the queue size is empty: Consider this scenario - There are a 1000 number of data points to be pushed for each of the 10 patients, and the BoundedBuffer capacity is 20. In some run of the client, the patient threads start running and together fill the request buffer with 20 messages. Now this is the max capacity, so the patient threads have to wait on the condition variable and go to sleep. Meanwhile the scheduler at some point schedules worker threads that pop out each of the 20 messages (and considering that the amount of worker threads >> amount of patient threads), they pop out all 20 items for the BoundedBuffer even before a single patient thread was scheduled and could push in a message. In this case we have the worker threads waiting on the condition variable that the BoundedBuffer size is 0, because there exists more requests to be pushed in by the patient threads when they get the chance to be scheduled. The worker threads break when the BoundedBuffer queue size is 0, but there are still further requests that are put in once the sleeping/ready patient threads get scheduled.

- Do not rely on setting a global boolean variable to indicate the breaking condition (e.g. some boolean variable set to true to indicate all workers threads have completed which the histogram threads can check to break) - Let's say all worker threads are done executing - you're relying on the main thread getting scheduled as soon as all worker threads are complete and setting the boolean variable to true. However, what if the scheduler runs the histogram threads such that the histogram threads have completed popping out all messages and are not context switched out before the main thread has set the variable to true? This means that they are forever waiting on the condition variable because the queue size is 0, and they can never again check for the boolean variable value after the main thread does get scheduled and updates this value.