PA3: Threading and Synchronization

Introduction

In this programming assignment, you will be integrating multithreading to increase efficiency and improve on 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 would affect 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.

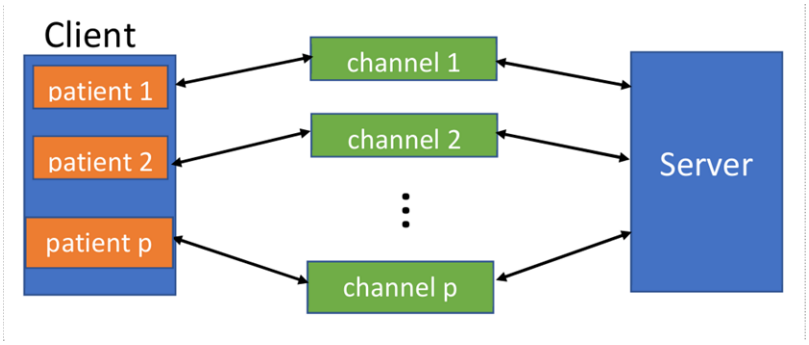


Figure 1: First approach to scaling PA1 performance - one thread per patient

However, there remains 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 for us to design a mechanism that would allow the w number of worker threads to read the request objects being produced by the p patient threads. We can use a buffer which can be thought of as a 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.

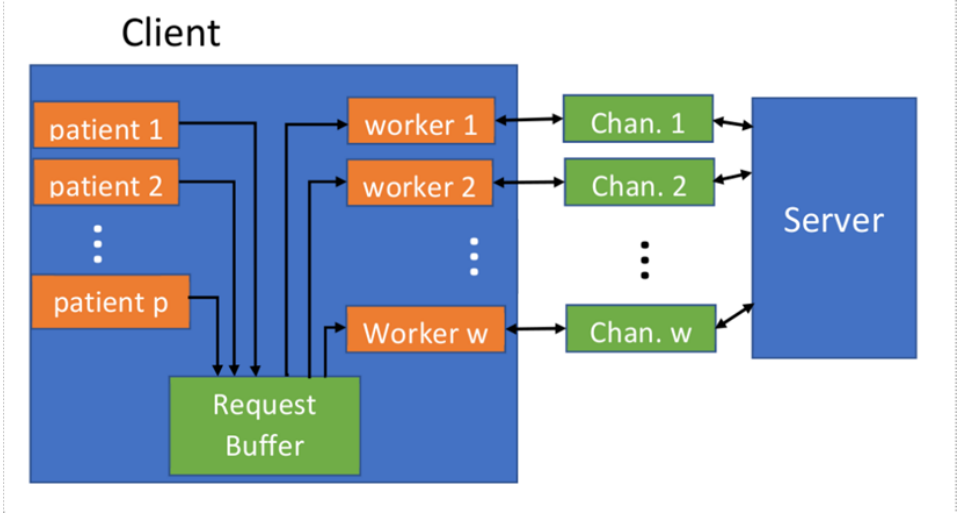


Figure 2: Second try with a buffer - number of worker threads w is now independent of number of patients p

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. Overflow can happen when the patient threads are much faster than the worker threads, while underflow can happen during 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.

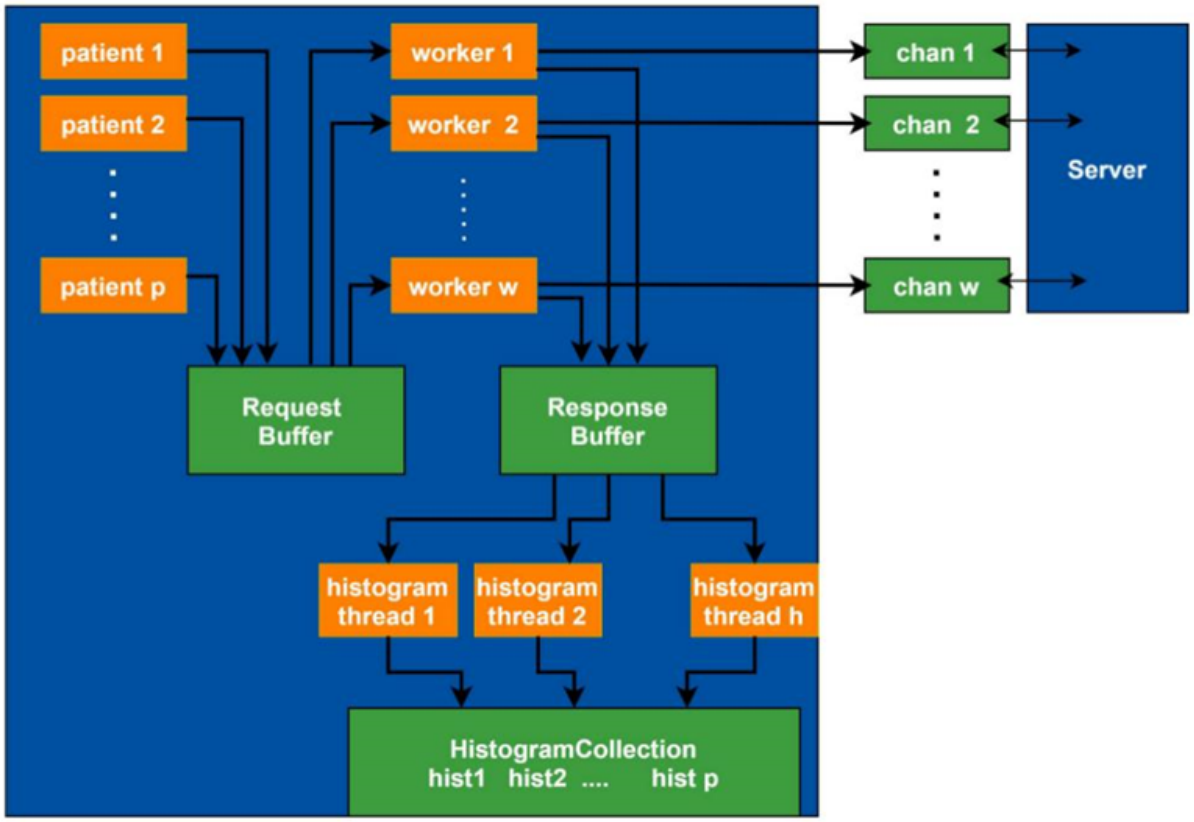


Figure 3: Structure for data requests

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 huge request buffer where all requests would fit. Make sure to test your program using 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 below procedure 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 communicate 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.

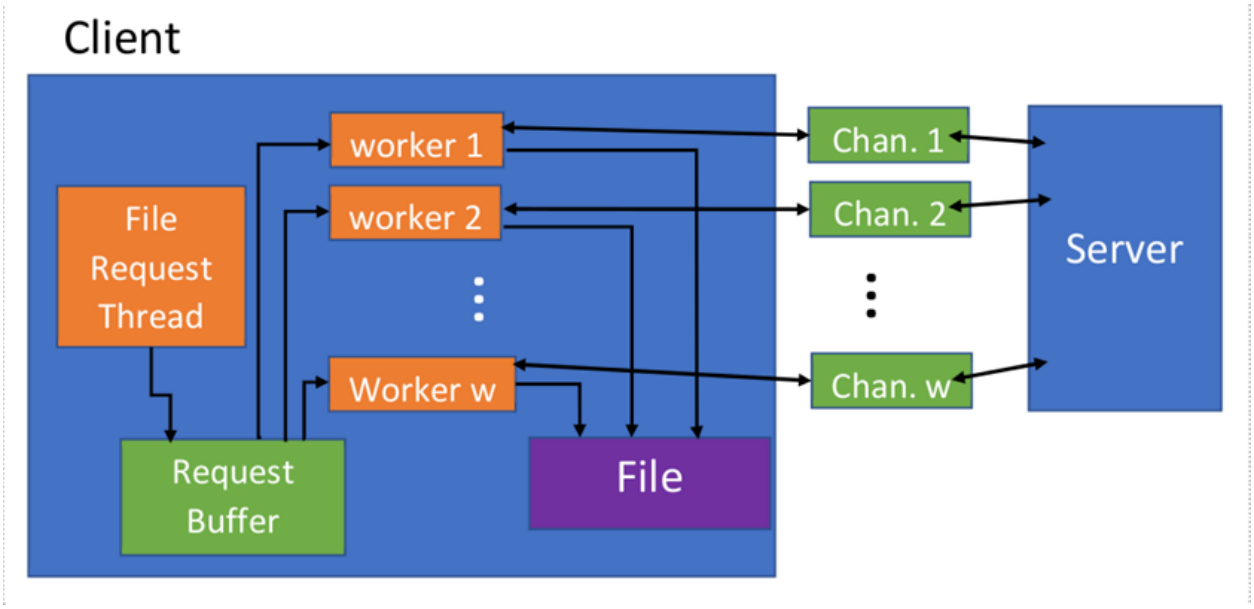


Figure 4: Structure for file requests

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.

BoundedBuffer class should need 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.

Assignment

Given Code

This assignment will be hosted using the **GitHub Classroom** platform. You must visit **https://classroom.github.com/a/K6fUWJx4** to create a repository for your project.

Code Requirements

The given source package 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. No modifications need to be made to 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.

Report (at least 1 and ½ pages)

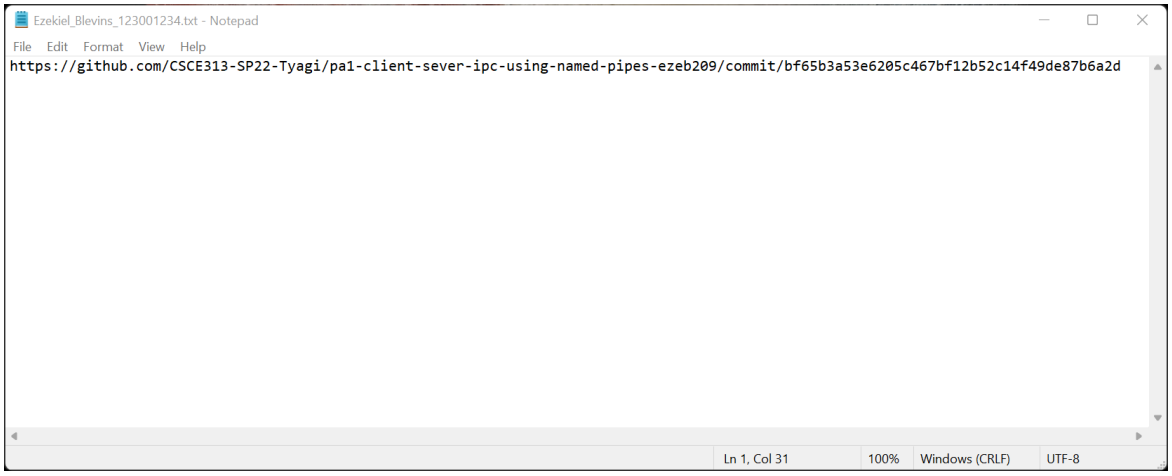
1. **Design**: Describe your implementation design. How did you implement the histogram and the worker threads? How did you synchronize threads to avoid race-conditions?

2. **Data Requests**: Make at least two graphs for the performance of your client program with varying numbers of worker threads **w** (try [50, 500]), varying size of bounded buffer **b** (try [10, 200]), varying size of histogram threads **h** (try [5, 100]) for patients **p** = 15 and ecg data count **n** = 15K. Discuss how performance changes (or fails to change) with each of them, and offer explanations for both. Do we see scaling on any of the parameters?

3. **File Request**: Make at least two graphs for the performance of your client program with varying numbers of worker threads **w** (try [50, 500]) and varying buffer capacity **m** (try [64, 4092]). Discuss how performance changes (or fails to change) with each of them, and offer explanations for both. Do we see a scaling? Why or why not?

What to Turn In

Once you have completed the assignment and submitted your final code to GitHub Classroom, on Canvas, turn in a text document named *FirstName_LastName_UIN.txt* containing the link to the commit hash that you are submitting (PA1 example below) and your report named *PA3_FirstName_LastName_UIN.pdf*.



Rubric

1. Implement BoundedBuffer class **[35 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 correct implementation.
2. Implement data requests and reporting **[28 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 to the terminal.
3. Implement file requests and transfers **[22 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.
4. Report **[15 pts]**
    a. Details described above.