PA#4: Threading and Synchronization

Introduction

In this programming assignment, you would increase the efficiency of your PA3. You may have noticed that PA3 takes a long time to collect 1K data requests per person. It would take much longer to collect all requests in a single file. Transferring raw files through file messages, although faster than using data messages, is still quite slow because of single channel usage at a time. In PA3, we learned how to create multiple channels, which the server processes independently and in parallel. However, we could not take full advantage of the parallelism because the client was single-threaded. In this PA, you would incorporate multiple threads to make these operations significantly faster.

Threading for Faster Programs

The primary bottleneck in PA3 for collecting data points is the fact that we must collect data points one after another. This problem is worsened further by the random processing time of each request (i.e., emulated using a `usleep()` call) by the server. And since the requests are all sequential, one delayed request-response would affect all subsequent requests. Notice in the `BIMDC/` directory that there are 15 files, one for each patient. 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 a person. Since the server already processes each channel in a separate thread, you can get atmost 15-fold speed over the sequential version. This technique is shown in Figure 1. Note that file requests can be spedup similarly.

However, there is a big problem: even if there are adequate hardware resources (e.g., CPU cores), you are getting only p-fold speedup, where p is the number of patients. You can write more complicated code to divide each patient's data k-ways and obtain pk-fold speedup. However, even with such scheme, since each request takes a random time, some patient threads may take much longer compared to the other ones. This is called the problem of "stragglers", where one or more threads take significantly longer time to complete a given task. The underlying reason for this problem is the fact that we try to divide the workload equally among threads often unsuccessfully due to the performance variations occuring during the runtime. To avoid this problem we must let the threads divide the work among themselves as they see fit.

The standard solution to this problem is to separate the tasks of *producing* these requests and *processing* them (i.e., sending to the server), and using a buffer in between these tasks. This way the number of patients p can be decoupled from the number of threads that would

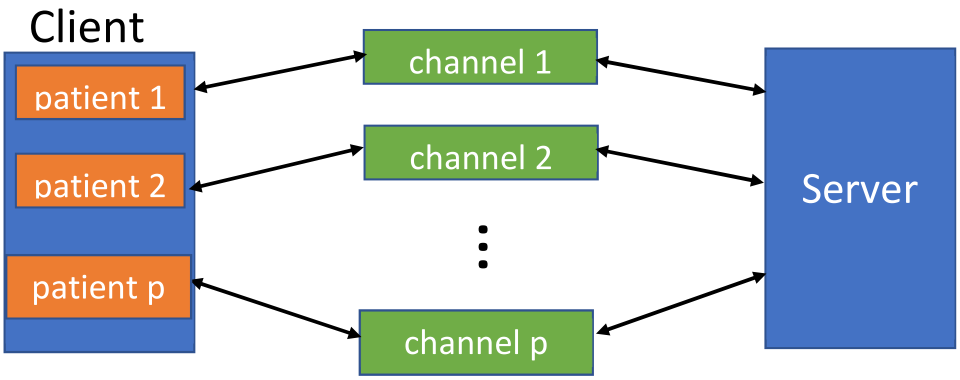

Figure 1: First approach to scaling performance - one thread per patient.

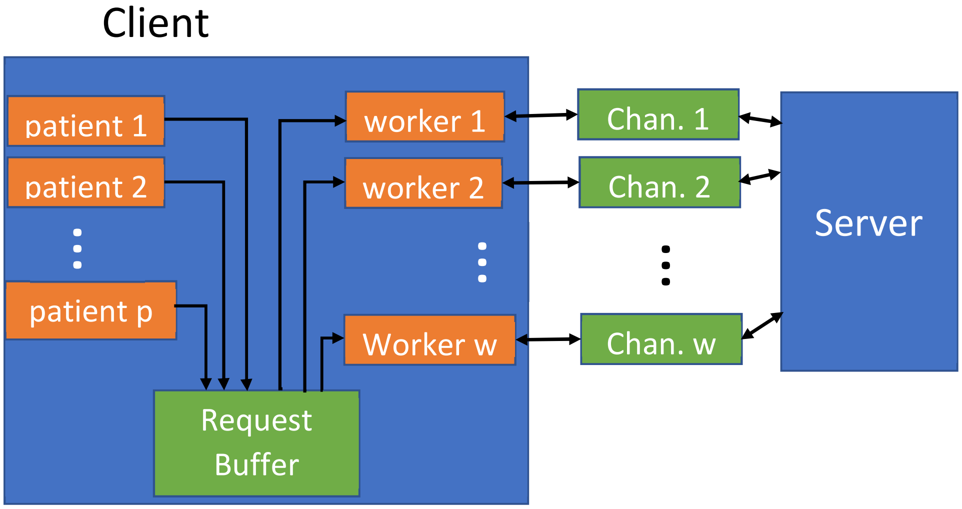

Figure 2: Second try with a buffer - number of worker threads w is now independent of number of patients p.

be in charge of communicating with the server, simplifying the scaling issue. That means now there are p `patient` threads pushing requests on to the buffer, which can be thought of an `STL::queue`. Then, there are w `worker` threads, each communicating with the server independently of each other through a dedicated request channel. As a result, the theoretical speedup factor is now w, which is indpendent of p and can lead to significantly more speedup with $w \gg p$. Figure 2 demonstrates this technique.

For this technique to work, you need a special and more compilcated buffer than just an `STL queue` between the patient threads and the worker threads. 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", "overflow" and "underflow". Overflow can happen when the patient threads are much faster than the worker threads, while underflow can happen under the opposite scenario. As described in the "Synchrnonization" lecture, the `BoundedBuffer` class is the perfect solution to all these problems. Nice thing is, you are already given the BoundedBuffer pseudocode in class and a template for its implementaion in the starter code.

Client Requesting Data Points

The workers threads in this PA work as follows. Each worker thread pops a request from the request buffer (i.e., the `BoundedBuffer` object between the patient and the worker threads), sends it to the server, collects the response, and then puts the response in the response buffer - another `BoundedBuffer` object. In addition, there are h histogram threads who pop these responses and update p ecg histograms, one for each patient.

There is a histogram per patient that keeps track of that patient's statistics. Note that multiple worker threads would potentially update the same histogram leading to another race condition, which must be avoided by using mutual exclusion.

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

Figure 3 shows the structure.

When requesting data messages, the program should take 4 command line arguments: n for number of data items (in range $[1, 15K]$), p for number of patients (range $[1, 15]$), w for number of worker threads (try between $[50, 5000]$), and b for bounded buffer size in bytes (acceptable range $[100, 1000000]$). For instance, the following command is requesting 15K ecg data for the first 10 patients using 500 worker threads and a request buffer of size 1KB (i.e., 1024 bytes). It will use a 1KB response buffers to collect the responses and then $h = 5$ histogram threads to make the 10 patient histograms for ecg values.

```
$./client -n 15000 -p 10 -h 5 -w 500 -b 1024
```

Note that all these arguments are optional, meaning that they can be omitted, causing their default values being 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 or after running dry.

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., $b = 100$) - your program should work perfectly fine, may be 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.

Client Requesting Files

You can also run the client for file requests. First, the client queries the file size just like PA3. After that, instead of sending the requests directly to the server, the client starts a thread that makes and puts all the requests on to the request buffer. The worker threads pop those requests from the buffer, send those out to the server through their dedicated request channels, receive the responses, and write those responses into the appropriate locations of the file. Note that unlike requesting data messages, there is no response buffer in this case. The structure is shown in Figure 4. Note that while the progrm is running, there are $w + 1$ threads working simultaneously: 1 thread for making the requests and pushing them to the request buffer, the rest w worker threads who keep taking from the request buffer and process them.

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. There is a specific mode for opening a file that would make this possible.

When requesting a flie, the client would take 4 command line arguments: w for the number of worker threads, b for request buffer size, m for buffer capacity to keep the file content in memory, and f for file name. The first two are optional and the last argument (i.e, "-f") is mandatory. The following example command asks to download the file "file.bin" using 100 worker threads and using a buffer capacity of 256 bytes.

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

Capacity m indicates the maximum number of bytes from the file that can be sent back from the server in each response. Note that the `server` needs to know how this value, which would be passed from the `client` as a command line argument through the `exec()` function.

You should vary m and w and report the variation in runtime that you experience for files of different sizes. Make sure to test your program using boundary conditions (e.g., $m = 30, w = 1$, or $m = 30, w = 5000$). You need the buffer capacity m at least big enough for the name of a new channel or any other messages exchanged between the client and the server.

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 a variable-length datatype to support different types of data of varying length. You might be tempted to use `std::string`. However, that will not do because it does not support binary data, which we will need to handle. Since each item can be of You can use a dynamically-allocated array of type `char` or `void` would do. However, coding that would be a little complicated because of maunal handling for allocation, deep copy and deallocation. Therefore, the preferred datastructure in this case is `std::vector<char>`. Because it is variable-length, stack-allocated, deep-copied using just the equal operator `=`.

The rest of the implementaion should be straight-forward from the class lecture. You are given the `Semaphore` class and a template for the `BoundedBuffer`. You just need to fill the functions (i.e., constructor, destructor, push and pop) and add necessary variables in the template `BoundedBuffer.h`.

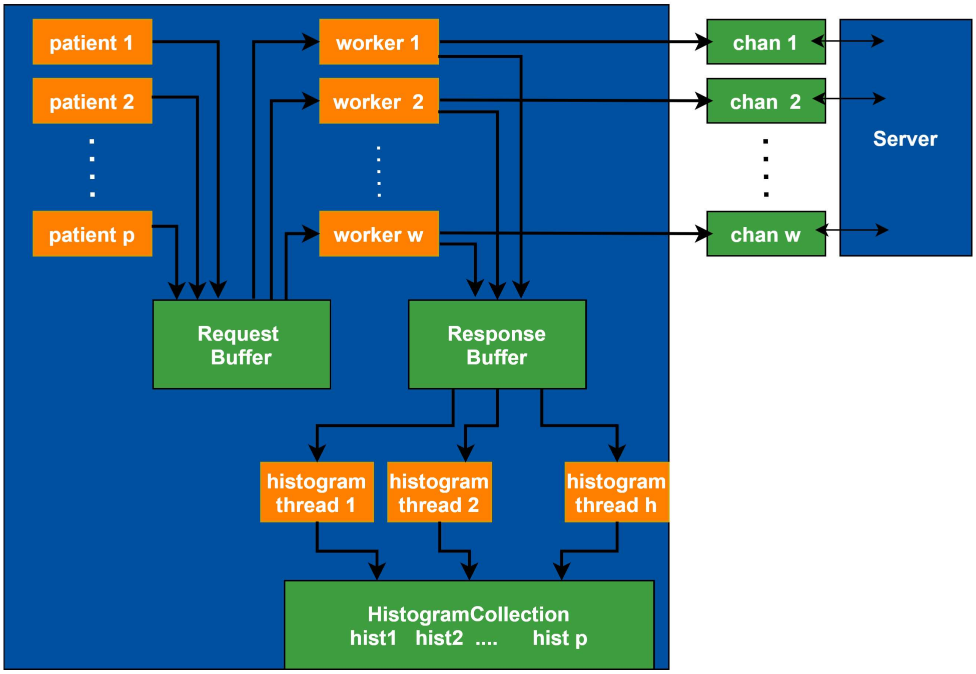

Figure 3: Structure of PA4 for data requests.

Assignment

Given Code

The given source package includes the files from PA3 (i.e., `server.cpp, client.cpp`, common.h/.cpp, `FIFOreqchannel.cpp/.h`, `Semaphore.h` and a partially filled `BoundedBuffer.h`). In addition, it now includes `Histogram.h/cpp` and `HistogramCollection.h`. The `Histogram` class encapsulates histogram management functionality. Note that you need to create a `Histogram` object for each patient, resulting in p (i.e., $p \in [1, 15]$) Histograms. All these histograms are added to the `HistogramCollection` object which maintains the list and provides a `print()` function to output the histogram values together. You may need to modify the `Histogram` class for thread-safety.

Finally, you need to implement the `BoundedBuffer` class and use it properly in the `client.cpp` file.

Your Task

Your code must also incorporate following modifications compared to PA3:

- 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 setting and collect runtime for each setting. You need to wait for all threads

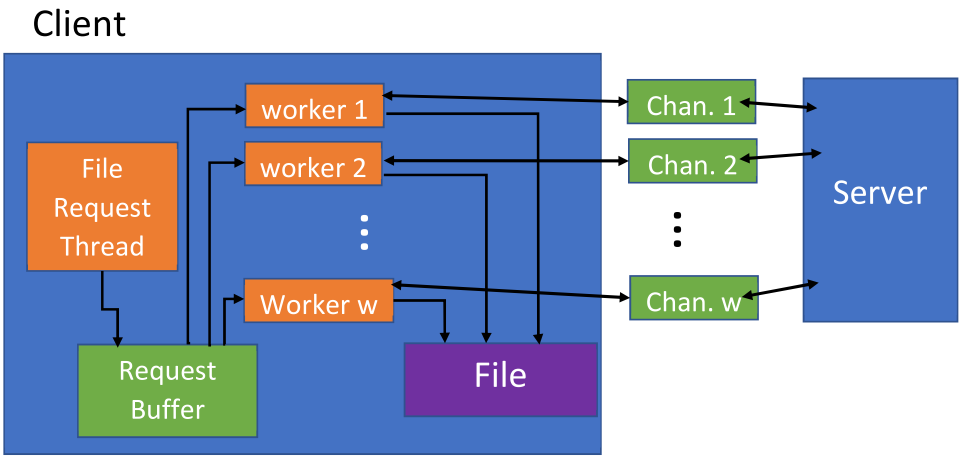

Figure 4: Structure of PA4 for file requests.

- For data requests, your client program should call `HistogramCollection::print()` function at the end. If you program is working correctly, the final output should show n for each person.
- The `server` should be launched from inside the `client` using `fork()` and `exec()` and the $-m$ argument must be passed along to the server so that the server uses the same communication buffer size.

Bonus

Write a signal-handler function that clears the terminal window (system("clear") is an easy way to do this) and then displays the output of either the `HistogramCollection::print()` function or show how much of the file have been received so far in the form of percentage or fraction.

In main, register your signal-handler function as the handler for SIGALRM (man 2 sigaction). Then, set up a timer to raise SIGALRM at 2-second intervals (man 2 timer create, man 2 timer settime), so that your handler is invoked and displays the current patient response totals and frequency counts approximately every 2 seconds. To do this, you will need to make sure that your signal handler is given all the necessary parameters when it catches a signal (man 7 sigevent). When all requests have been processed, stop the timer (man 2 timer delete).

If you have succeeded, the result should look like a histogram table that stays in one place in the terminal while its component values are updated to reflect the execution of the underlying program. You can use global variables for the bonus part.

Note that this is an example of asynchronous/real-time programming where the program performs certain operations based on the clock instead of in synchrnous manner. Such technique is useful when a program itself is busy doing its main work, while the asynchronous part is in charge of dealing with real-time events (e.g., printing something every few seconds, wrap-up computation after a deadline expires).

Report

1. Data Requests: Make two graphs for the performance of your client program with varying numbers of worker threads and varying size of request buffer (i.e. different values of w and b) for $n = 15\text{K}$. 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?

2. File Request: Make two graphs for the performance of your client program with varying numbers of worker threads and varying buffer capacity (i.e. different values of w and m). Discuss how performance changes (or fails to change) with each of them, and offer explanations for both.

What to Turn In

- The full solution directory including all cpp/h files and a `makefile`
- Completed report

Rubric

1. `BoundedBuffer` class implementation using the given `Semaphore` class.(15 points)

2. Cleaning up fifo files and all dynamically allocated objects (10 pts)

3. Correct counts in the histogram (25 pts) for data requests

4. File Reqeusts (25 pts) with multiple threads
   - Text files (10 pts)
   - Binary files (15 pts)

5. No global variables are allowed (except for the bonus part). (10 points)

6. Report (15 pts)
   - Should show plots of runtime under varying n, b, w, m, h.
   - Having maximum $w < 500$ will result in 5 lost points. You must find a platform that supports 500 threads or more.

7. Bonus: using timers to display the counts (10 pts)
   - If your implementation uses a separate thread instead of a signal handler, you only get 5 bonus pts. You should also make sure that the threads that do not relate to this signal handling should ignore the signal and not wake up.