PA#4: Threading and Synchronization

Due: 4/9/20 Thursday at 11:59pm

Introduction

In this programming assignment, you would increase the efficiency of your PA2. You may have noticed that PA2 takes a long time collect all the requests for just 1 person. Transferring files is even slower. You would incorporate multiple threads to make these operations faster. However, that must be done carefully so that we eliminate bottlenecks instead of needlessly using threads.

Threading for Faster Programs

In PA2, we have several bottlenecks. First, the communication channel is sequential for each request, which can take a random amount of processing time from the server side. As a result, one request being delayed would affect all subsequent requests. Notice in the `BIMDC/` directory that there are 15 files, one for each patient. If you are to collect data for say p of these patients, an efficient approach would be to use p 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 person.

The server already processes each channel in a separate thread. As a result, you can get 15-fold speed over the sequential version assuming $p = 15$. Figure 1 describes this technique.

However, there is a big problem: even if there are adequate hardware resources (e.g., CPU cores), you cannot use more than 15-fold speedup. To avoid this roadblock, we have to make the number of threads some how indpendent of the number of patients.

The standard solution to this problem is using a buffer where the p patient threads would push requests. Now, there would be w threads, called worker threads, each communicating with the server independently of each other through a dedicated request channel. As a result, you can now decouple the speedup factor from the number of patients. Assuming that the communication between the client and the server is the main bottleneck, you can now acheive w times speedup, which is indpendent of p and potentially $w >> p$. Figure .2 demonstrates this technique.

For this technique to work, you need a special and more compilcated buffer between the patient threads and the worker threads. First, the buffer needs to be thread-safe; otherwise simultaneous access from the producers (i.e., patient threads) and consumers (i.e., worker threads) would lead to 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,

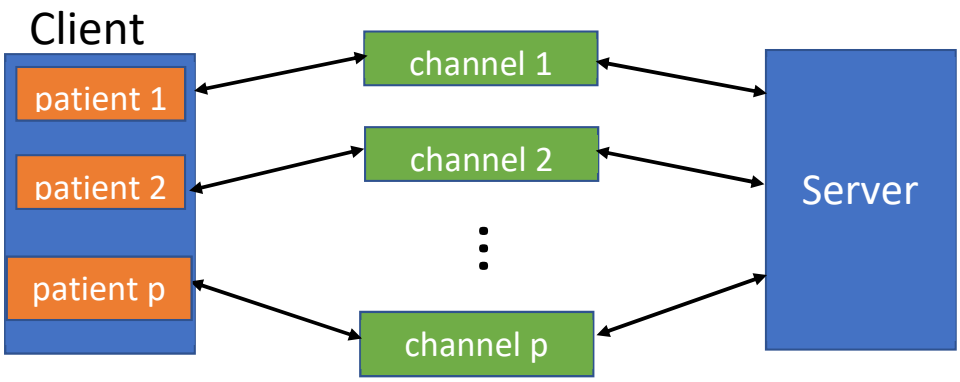

Figure 1: First try to scaling PA2 performance - one thread per patient.

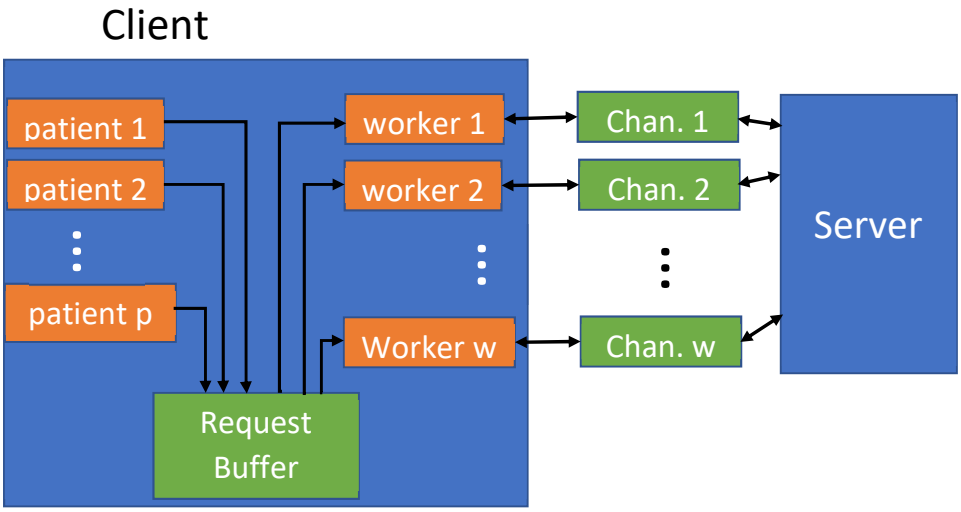

Figure 2: Second try with a buffer - number of worker threads w is now independent of number of patients p.

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 case. Use a *BoundedBuffer* class to solve all these problems . See the class lectures to find out more about BoundedBuffer.

Client Requesting Data Points

The workers threads in this PA work as follows. Each worker thread pop a request from the request buffer (i.e., the buffer between the patient threads and the worker threads) that contains `datamsg` objects, sends it to the server, collects the response, and puts the response in the patient's (the patient no is extracted from the data message request) histogram. 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. 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 (range 1 to a few hundred). For instance, the following command is requesting 15K data items each 2 persons (i.e, patients 1 and 2) using 100 threads, and using a bounded buffer of capacity 50 requests.

```
prompt $>
```

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 $p + w$ threads in this design: p patient threads and w worker threads. All these threads must be running simultaneously for your program to work. You cannot just have a huge request buffer where all requests would fit. Make sure to test your program using $b = 1$ indicating the bounded buffer size is 1 - your program should work perfectly fine (of course a bit slower) with this. Smaller b value along with high p, n increase concurrency and thus manifest race condition bugs that are not visible otherwise. 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 reqeusts the size of the file using a special `filemsg` (like PA2). After that, instead of sending the `filemsg` requests directly to the server, the client starts a thread that puts makes all the requests for this file and pushes those to the request buffer. The worker threads pop those requests from teh buffer, send those out to the server through their dedicated request channels, receives the response, and writes those responses into the appropriate locations of the file. 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 other w threads for the workers, who keep taking from the request buffer and sending those to the server.

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 enough so 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 3 command line arguments: w for the number of worker threads, m for maximum buffer size 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.

```
prompt $>
```

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 `dataserver` 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., $m = 1, w = 1$, or $m = 1, w = 5000$).

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. You are not allowed to use other primitives either.

Condition variable is an excellent way for asynchronously waiting for certain event/condition. We have seen in class the alternative of such asynchronous wait and notification is some busy-spin wait which is either inefficient or wasteful in terms of CPU cycles. POSIX provides `pthread_cond_t` type that can waited upon and signaled by one or more threads. In PA4, we can use this type to guard both overflow and underflow from happening.

You will need one condition variable for guarding overflow, and another one for guarding underflow. Each producer thread (i.e., request threads) waits for the buffer to get out of overflow (i.e., buffer size is less then the maximum) encoded in condition # 1 and then pushes an item. It also notifies the consumer threads (i.e., worker threads) by signaling condition # 2 that data is now available. This wakes up all waiting consumer threads (if any) one at a time. The logic for the consumer threads is similar with the difference that consumer threads wait on condition # 2 and signals condition # 1. Please check the man pages and the L11 slide set for reference.

Assignment

Given Code

The given source package includes the files from PA2 (i.e., `dataserver.cpp, client.cpp, common.h/.cpp`, and `FIFOreqchannel.cpp/.h`). In addition, it now includes `Histogram.h/cpp` and `HistogramCollection.h` both fully implemented for you. 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 nice `print()` function to output the histogram values together. Finally, the package contains a template for the `BoundedBuffer` class (.h/.cpp) that you have to fill out and use that properly in the `client.cpp` file.

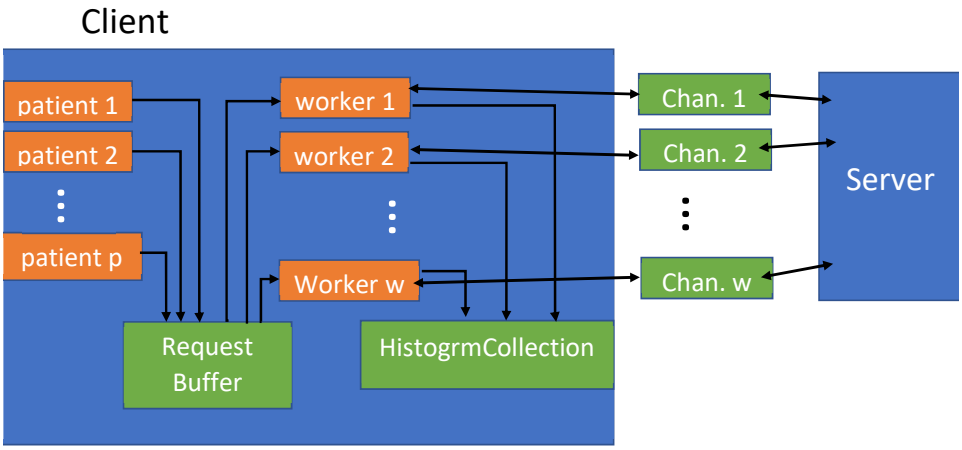

Figure 3: Structure of PA4 for data requests.

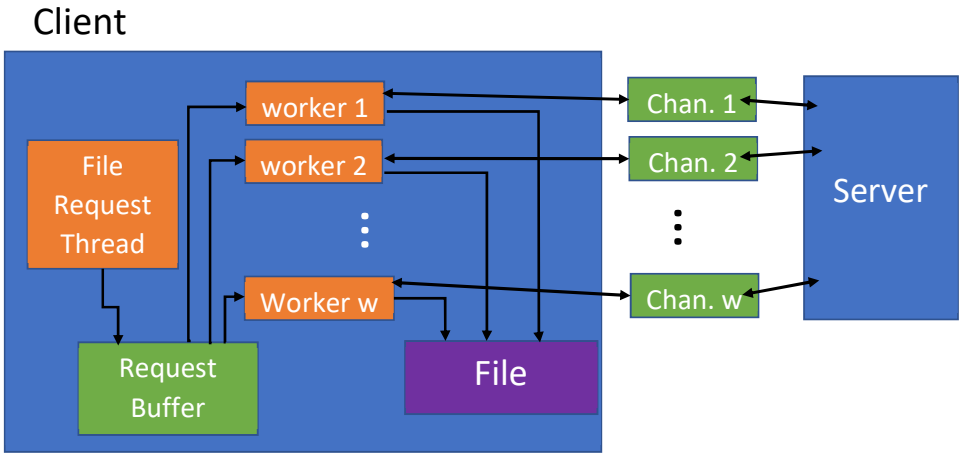

Figure 4: Structure of PA4 for file requests.

Your Task

Your code must also incorporate the following modifications compared to PA2:

- Your client program should accept all the command line arguments: n, p, w, b, m, f. 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 and w worker 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
- 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.
- Your program should include a functional `BoundedBuffer` class that is thread-safe and guarded against over-flow and under-flow.

- The `dataserver` should accept another argument m for buffer capacity which should be passed along from the `client`.

Bonus

You have the opportunity to gain bonus credit for this Programming Assignment. To gain this bonus credit, you must implement a real-time histogram display for the requests being processed.

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 $n = 15$K. Discuss how performance changes (or fails to change) with each of them, and offer explanations for both. Do we see linear 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. Do we see a linear scaling? Why or why not?

What to Turn In

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

Rubric

1. BoundedBuffer class (20 pts)
    - Your program cannot have a `Semaphore` class. Having one would result in 20 lost 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
    - Binary files (15 pts)
    - Text files (10 pts)
5. Report (20 pts)
    - Should show plots of runtime under varying n, b, w, m.
    - Having $w < 500$ will result in 5 lost points. You must find a platform that supports 500 threads or more.
    - Requiring $b = 3 * n$ will result in 15 lost points, because that indicates that you did not need any overflow or underflow checks.
6. 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