Project README file

This is **YOUR** Readme file.

Note - You will submit a PDF version of this readme file. This file will be submitted via Canvas as a PDF. You may call it whatever you want, and you may use any tool you desire, so long as it is a compliant PDF - and for us, compliant means "we can open it using Acrobat Reader".

Project Description

Your README file is your opportunity to demonstrate to us that you understand the project. Ideally, this should be a guide that someone not familiar with the project could pick up and read and understand what you did, and why you did it.

Specifically, we will evaluate your submission based upon:

- Your project design. Pictures are particularly helpful here.
- Your explanation of the trade-offs that you considered, the choices you made, and *why* you made those choices.
- A description of the flow of control within your submission. Pictures are helpful here.
- How you implemented your code. This should be a high level description, not a rehash of your code.
- How you *tested* your code. Remember, Bonnie isn't a test suite. Tell us about the tests you developed. Explain any tests you *used* but did not develop.
- References: this should be every bit of code that you used but did not write. If you copy code from one part of the project to another, we expect you to document it. If you *read* anything that helped you in your work, tell us what it was. If you referred to someone else's code, you should tell us here. Thus, this would include articles on Stack Overflow, any repositories you referenced on github.com, any books you used, any manual pages you consulted.

In addition, you have an opportunity to earn extra credit. To do so, we want to see something that adds value to the project. Observing that there is a problem or issue *is not enough*. We want something that is easily actioned. Examples of this include:

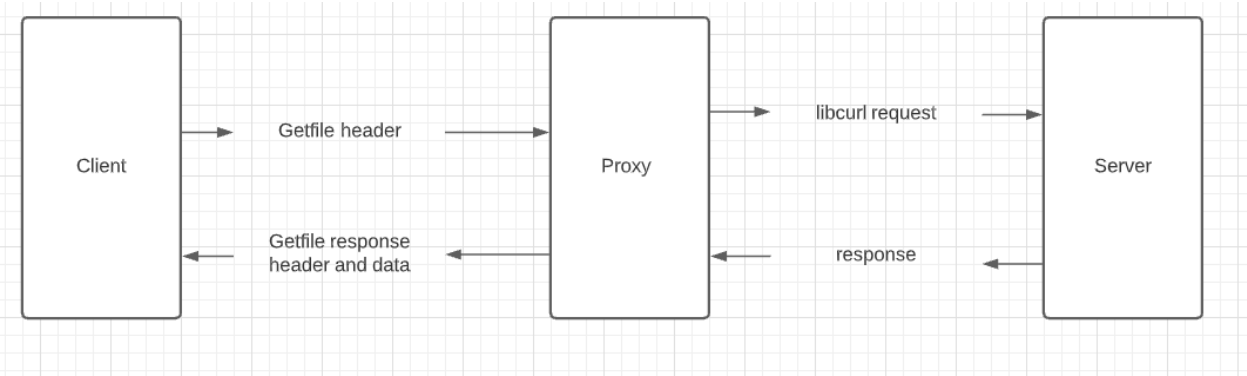
- Suggestions for additional tests, along with an explanation of *how* you envision it being tested
- Suggested revisions to the instructions, code, comments, etc. It's not enough to say "I found this confusing" - we want you to tell us what you would have said *instead*.

While we do award extra credit, we do so sparingly.

Part 1 - Proxy Server

For part 1, we converted the client and server process to contain a proxy. The proxy acts as a middle man between the client and server, where it receives requests from the client and then requests the server for those requests and then takes the response from the server and it sends it back to the client. So instead of retrieving the files from the disk as in the past implementations, this time it fetches from a github repo. To handle the fetching from a github repo, the proxy uses the libcurl easy interface. The libcurl easy interface supports http requests and writes the data it gets back to a struct using a callback. Once we get the data back, all that needed was to use gfs_send to send the data back to the client. Code was implemented with a lot of help from the libcurl easy interface documentation. It was pretty well documented, and through that I was able to write the proxy needed to act as the middle man between client and server. The most important part for the interface was having a struct and a callback function available for the request. It needed something to write to and a function that will write to that struct. The struct was simple, a char field for it to store the actual response and then a size field that'll store the size of that response. The size was later used to send from the proxy back to the client. The libcurl interface also has a geteasyinfo function that lets you get relevant information that you need about the request such as the response code of the call. That came in handy in determining which status was to be sent back to the client.

For testing, I just manually ran it multiple times, each time with a different varying number of requests and threads. As long as the client or proxy didn't hang it was in a good spot. The proxy was to be used in a multithreaded environment, so it shouldn't break or stall if multiple threads are running at the same time which it didn't. It should also be able to handle multiple requests coming in and not just quit after one request, which it did as well.



References

For the proxy, I relied very heavily on the libcurl easy interface documentation available at https://curl.se/libcurl/c/. I was able to get more info on the methods I needed just be reading through this site and seeing the options available.

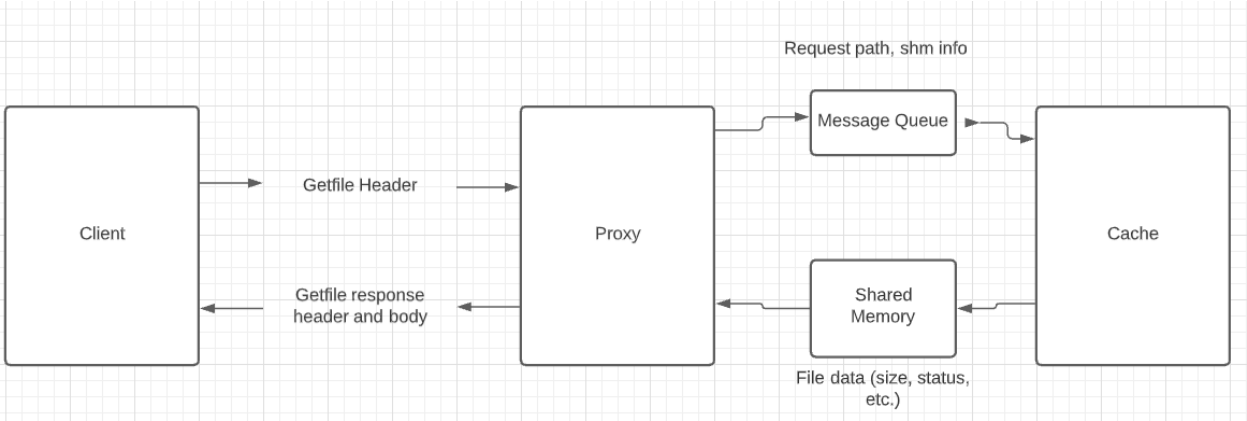
Part 2 - Shared Memory

For part 2, again we leveraged the proxy but this time a little differently. Instead of doing a http request, we did cache requests. In addition to doing cache requests, IPC was implemented between the proxy and cache to talk to each other and send information back and forth. Message queues and shared memory-based IPC were both used. Message queues was used to handle the basic information about the requests, so the the proxy would push onto the message request a path along with the information needed to open the shared memory IPC later when it got the file. The cache would get the request from that message queue and push onto another message queue, the getfile response header information, so the status of the file and the file size. Once the proxy and cache had those information, it would open the shared memory if necessary (meaning in only cases where the status was GF_OK otherwise it wouldn't open the shared memory). With the shared memory open, semaphores was used for synchronization between the proxy and cache. The cache would first write to the shared memory before the proxy can read it. The read sempahore was initialized with a value of 0 and write with a value of 1, so the read semaphore would block first allowing the write to go first and write to the shared memory. Each chunk of data that was read from the cache into the shared memory, the proxy would use gfs_send to send that chunk to the client. Once the shared memory transfer was complete, the shared memory segment was enqueue back into the queue, making it available to be picked up by another thread.

Since there were choices available as to which shared memory api to use, I decided to go with the POSIX message queues and shared memory IPC. I decided to go with this since POSIX was newer compared to SysV and simpler to use. I decided to one queue, that would contain information pertinent to each request. So in my case, that would be the request path along with the shared memory name. I used unnamed semaphores because it was easier to initialized them at the beginning in webproxy without needing to create and pass more names for semaphores. It was also cleaner that way for me. I created a struct for the mmap process that contained the read and write semaphores, the status of the request, total size of the file, the number of bytes in the current chunk that is in the shared memory, and finally a data field for the data chunk. I had to ftruncate the size to be the size of the struct plus the number of bytes allowed in the segment, allowing enough room in the data field of the struct to contain segsize bytes of data. The cache would then first write to the shared memory the file get status and size, which the proxy will then read and determine what to do with the status. If the status is good, it'll send GF_OK, if the file was not found it'll send FILE_NOT_FOUND and so on. If the status was ok, it'll go on to send the data from the cache and the proxy will read that corresponding data. As mentioned before, write semaphore was given an initial value of 1 because it should write first while the read blocks. Outside of the loop once it finishes it waits for the proxy to post to the write semaphore again, allowing the cache to clean up the semaphores. All the shared memory segments were created in the webproxy setup and put into a queue, that would later allow the handle_with_cache function to pick up a request only when there was a segment free and available. Once a segment was done, we had to add it back to queue to be picked up by another thread for another requests. The only time they are fully destroyed are when the webproxy and cache are terminated. As mentioned before, the proxy just requests the file

from the cache, so if it doesn't exist in the cache we just respond with the FILE_NOT_FOUND status.

For testing, similarly to part 1, I tested with multiple segment sizes, different number of segments, threads, and requests. I started testing with all of the three simplecached, gfclient_download, and webproxy with the same number of segments and threads and then just made sure the service didn't stall or hang with a higher number of requests. Even with a 100 requests it was pretty instantenous. The next step was to make sure that with varying number of segments, segment size, requests, and threads that it worked fine. Through that I was able to uncover a bug that was causing it to hang on the webproxy side where the shared memory segment would just hang even after receving the first chunk. Weird part was that it doesn't happen consistently on my local machine, but did on gradescope. So I had to go back and double check if I was incurring any race conditions. Then I just went through the same tests I did before to double check if the fix worked.



References

For the POSIX shared memory and message queues and semaphores, I relied on a couple different resources. The most notable one was https://man7.org/conf/lca2013/IPC_Overview-LCA-2013-printable.pdf, which gave a great outline of which functions were available for the message queues, shared memory, and semaphores. Other than that I used tutorial pages and the man pages to better understand the functions needed for the IPC.

Improvements

I feel like for this assignment, there's opportunity to combine these two parts into one bigger assignment. With the idea of the cache and using IPCs to communicate between processes, we can do something about the cache misses. So whenever the cache doesn't contain the file, we can make a http request similar to part 1 to get the file and then put it into the cache. With the cache, there's opportunity to expand that too, maybe have students implement a LRU policy or a different simple cache policy on the cache. With those additions, in my opinion it would make the assignment more interesting and not feel as repetitive. It would also have us better understand how it truly works together.