GetFile Client

Project Design: Choices and Tradeoffs

My GetFile Client design follows this workflow: 1) establishing a socket connection to a server, 2) sending a GET request, 3) parsing the response header, and 4) receiving and storing any file data received.

Client Request Structure

First is the Client Request Structure, `gfcrequest_t`, which is a structure that holds all the necessary information for each request, including the server address, port, file path, callback functions, and even information necessary for data processing, like the length of the file, the number of bytes received, and the request status. By packing all of this information into one struct, I could more easily keep track of all of the information important to the ongoing request.

Header Parsing

Next is header parsing: this assignment involves a custom protocol mutually understood by the client and server, with a very specific format for the request and response headers.

While designing my client, I decided to create a buffer that would accumulate the response header over time, until it found the `r\n\r\n` string, which indicates the end of a well-formed header. By doing this, my client could accommodate a server that sends a response header in chunks. Once the buffer has accumulated the entire header, I used `strtok` to quickly parse it and ensure the necessary pieces are there.

I also decided to return early after parsing the header if the header is found to contain any status other than `GF_OK`; returning early is a more efficient use of time, since there is no point in moving on to the data processing step if there is an error or if the file is not found.

Processing Payload

Since the server might send part of its payload with the header, my client first processes any part of the payload that might have arrived with the header. Then, my client uses a loop to receive the payload inside of a buffer with a `recv` call.

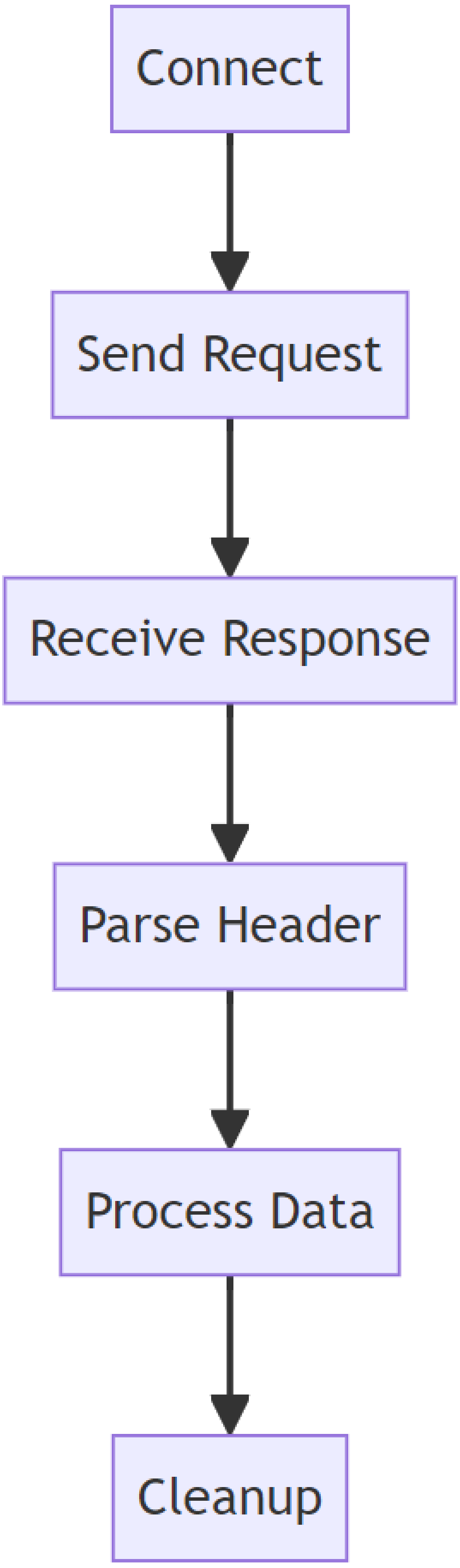
I chose to use a buffer to receive the data in chunks; by receiving in chunks, the client is able to process much larger files without overflowing the buffer data structure.

Returning Status

At this point, if the server stops sending bytes, or the connection closes, my client does not immediately return an error. Instead, it moves onto the return logic. My client checks if the bytes received matches the length of the file specified in the response header; if yes, it returns a 0, and if not, it returns a -1 to indicate an error. In either case, `GF_OK` status is left alone to indicate that the request went through, even if the payload did not fully process for some reason.

Flow of Control

The worfklow of the client can be seen in this diagram:



The client starts by creating a request structure that contains all the information necessary for the request. It then configures the request with specific details such as server address, port, file path, and callback functions.

Next, it performs the actual request, establishing a connection to the server using the specified address and port by opening a socket and binding it to the server.

If the connection is successful, the client sends a formatted request for a specific file. The client then enters a loop to receive the response from the server. It looks for the header part of the response to determine the request status and length of the file.

If the request was sucessful, the client starts receiving the file data, which is passed to a write function specified by the caller. It compares the bytes received to the expected file length to know when the file transfer is complete.

After the file transfer is comoplete or if an error occurs, the client performs cleanup, which includes closing the connection to the server and freeing dynamicaly allocated memory.

If at any point an error occurs, it sets an appropriate status within the request structure and returns early where necessary.

Testing

I used the makefile to test for compilation errors, and used `make all_noasan` and `valgrind` to check for memory leaks.

After that, the main way I tested my code was using Miguel Paraz's test files in his `6200-tools` repo, linked here. For this client, I used his file `gftestserver.py`, which provides a number of different responses for the client for testing purposes.

The main edge cases in the test file that were useful were:

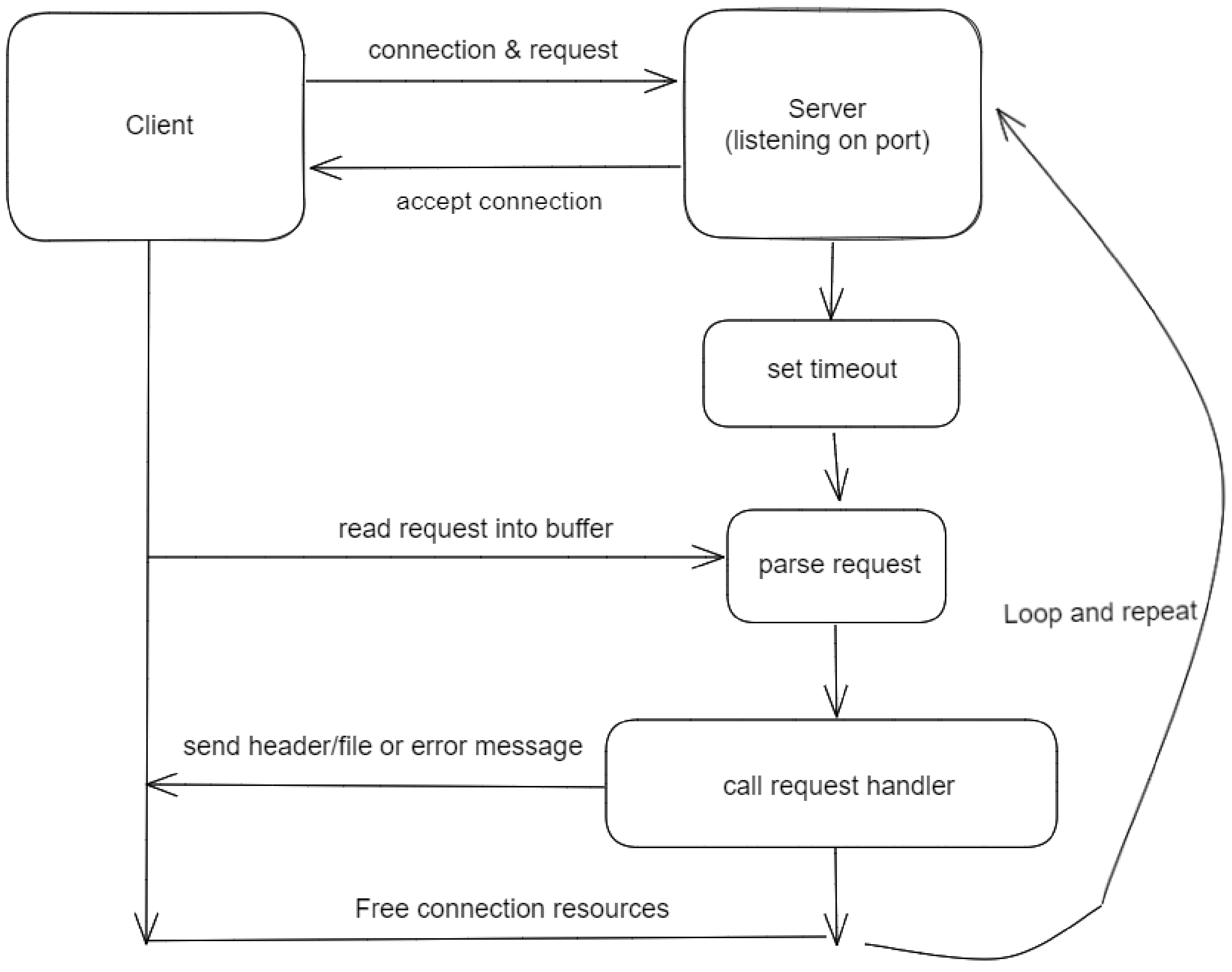
- Sending the header in multiple pieces with a small payload
- Sending a slow response with enough time to interrupt (test how the client responds when the server connection is interrupted)
- Sending a non-numerical file length
- Sending a bad character in the middle of the end of header indicator string
- Sending the end of header indicator string in the middle of the payload
- Sending an incomplete header and closing the connection
- Sending FILE_NOT_FOUND, ERRROR, and INVALID statuses
- Sending an invalid scheme or status
- Sending a large file (>2GB)
- Sending a random sized file with random-sized buffers

Once I was passing all of these local tests, I submitted to GradeScope which revealed the last two issues with my code: I had to make my client IPv6 compatible, and also had to NOT set the request status to GF_INVALID if the server closes the connection early (instead, check if all the bytes were received first).

Getfile Server

Flow of Control

The flow of control of the server, as well as its interactions with the client, can be seen in this diagram:



My GetFile Server consists mainly of a server structure (`gfserver_t`), initialized by `gfserver_create()`, that contains important information about the server, such as the port, the handler callback function, and the max pending connections it allows (`max_npending`). The main function, `gfserver_serve`, creates a socket and binds it to a port. The server then starts to listen for incoming connections.

The server then loops infinitely, waiting for client connections. For each connection:

- A socket is accepted, and a timeout is set using a `timeval` struct
- The request is read into a buffer and parsed
- If the request is invalid, `gfs_sendheader` is called with `GF_INVALID` to indicate an error
- The request handler is called if the request is valid
- The handler function is responsible for sending the file or an error message to the client using `gfs_sendheader()` and `gfs_send()`
- The resources associated with the connection are freed The server continues to listen for new connections.

Design Choices and Tradeoffs

Server Structure

The foundation of the server is the `gfserver_t` struct. I chose to add the bareminimum attributes to the struct to keep the server lightweight and efficient; the port, the max_npending connections, the handler, and the handlerarg. The handler callback function adds modularity to the code, as any handler can be passed to the server.

Request Buffer and Fixed Timeout

As client requests don't always come all at once, I chose to use a request buffer (similar to the buffer in the client implementation) to accumulate the request over time. The accumulation terminates once it finds the `\r\n\r\n` string, which signifies that we have the full header, and the server is then able to move onto parsing the header.

However, this created an issue: if the client sends corrupt or malformed headers without the proper ending string, the server would hang, waiting indefinitely for the client to complete the header.

So, in order to avoid that, I chose to set a fixed timeout duration for each client connection. If the client took more than a certain amount of time (in my server's case, 5 seconds) to send the complete header, the server can just assume the header is malformed, and close the connection.

Adding the buffer and the timeout may cause some good but slow requests to time out. On the other hand, it can also cause the server to process client requests slower, since each request may take up to 5 seconds. However, overall I think these costs are wroth it in order to enable the server to parse client requests that come in pieces, which is a relatively common occurrence. The timeout number can be increased or decreased as well according to our needs.

Header Parsing

I used `strtok`, just like in the client, to parse the request header. Here, I chose to check for a number of possible edge cases, like paths that are too long, that don't contain a `/`, that are only a `/`, and more. Malformed and incomplete requests generally get hit with a `GF_INVALID`, and the valid requests are left to the `handler`, which magically handles attempting to retrieve and send the file.

Testing

Similar to the client, I used the makefile to test for compilation errors, and used `make all_noasan` and `valgrind` to check for memory leaks.

I also used the default tests provided in `gftestclient.py` in Miguel Paraz's `6200-tools` repo.

The main edge cases in the test file that were useful were:

- Sending a complete request for a non-existent file
- Sending an incomplete request
- Sending corrupt headers
- Sending invalid scheme and method
- Supporting PATH_MAX and failing when path is greater than PATH_MAX
- Sending request in tiny pieces
- Sending invalid paths

- Closing connection before receiving the response

To the test file, I added one more test that tests for an existing file, just to make sure the server will return a 200, `GF_OK`, and the correct file length to the client.

Once I passed all these local tests, my code passed the Gradescope tests without much trouble.

Multi-Threaded GetFile Client

Flow of Control

The Multi-Threaded GetFile Client first parses command line arguments to configure the server address, port, the number of threads, and number of requests to be served.

The client then initializes a pool of worker threads using `pthreads` to process multiple download tasks concurrently. The client also initializes a shared task queue. The client enqueues the download tasks into a shared queue, with each task specifying the file path to download; it keeps track of the number of active requests in a global variable called `active_requests`.

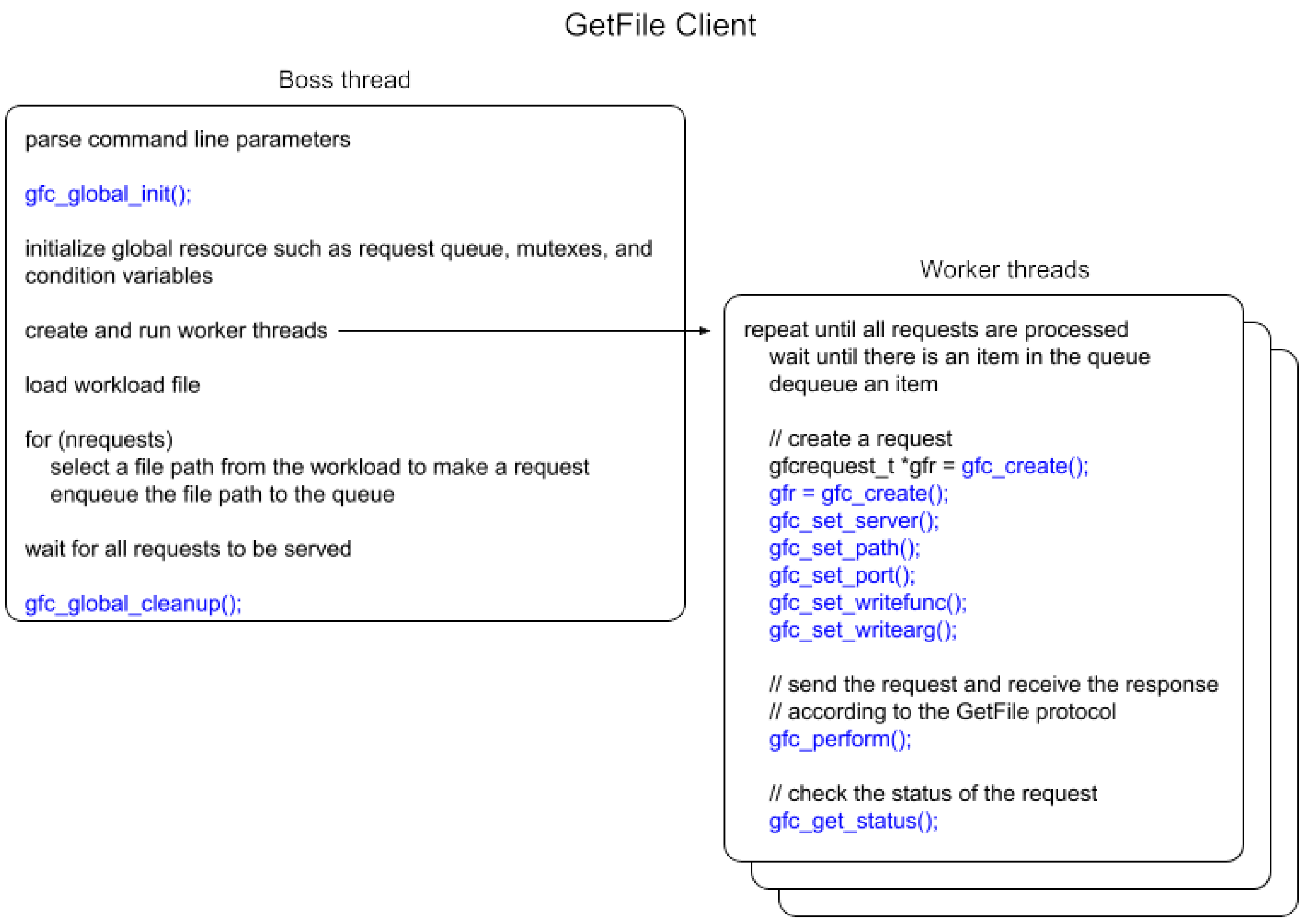
The worker threads work in a continuous loop. While the request queue is empty, the worker threads wait for the a request to be queued. The boss thread uses a mutex to queue requests, and signals a worker thread when a request is queued. A worker thread can dequeue the task, unlock the mutex so other threads can start processing other requests at the same time, and then start working on the task.

Whenever a worker completes a task, it decrements the `active_requests` variable. If this hits 0, the worker signals the main thread that the requests are all done.

Once the boss thread gets this signal, it queues signal termination tasks to the workers (also known as a poison pill). Each worker eventually picks up a poison pill, which causes them to free resources and exit.

The boss thread then joins the threads, cleans up resources such as allocated paths and the shared queue, and returns 0.

Below is the instructor-provided high level implementation diagram of the boss and worker threads for the multi-threaded getfile client (link to original file here: https://docs.google.com/drawings/d/1a2LPUBv9a3GvrrGzoDu2EY4779-tJzMJ7Sz2ArcxWFU/edit):


GetFile Client
Boss thread
parse command line parameters
gfc_global_init();
initialize global resource such as request queue, mutexes, and condition variables
create and run worker threads
load workload file
for (nrequests)
select a file path from the workload to make a request
enqueue the file path to the queue
wait for all requests to be served
gfc_global_cleanup();
Worker threads
repeat until all requests are processed
wait until there is an item in the queue
dequeue an item
// create a request
gfcrequest_t *gfr = gfc_create();
gfr = gfc_create();
gfc_set_server();
gfc_set_path();
gfc_set_port();
gfc_set_writefunc();
gfc_set_writearg();
// send the request and receive the response
// according to the GetFile protocol
gfc_perform();
// check the status of the request
gfc_get_status();

Design Choices and Tradeoffs

Boss-Worker Pattern

The Boss-Worker Pattern was prescribed for us in the assignment. This model allows a central boss thread to control the tasks for the worker threads.The boss thread can become a bottleneck for more complicated/work intensive applications though, and another bottleneck is the mutex surrounding the queue (only one model can dequeue a task at a time, though the mutex is immediately unlocked after the dequeue). On the plus side, it's scalable, easy to understand (boss and worker is very intuitive and simple), workers can divide up tasks, and also quite fault tolerant as different workers can pick up any task.

Global Mutex and Conditions

To maintain integrity in the shared resources, I used a `queue_mutex` to control access to the shared queue. Since the `request_queue` itself can be accessed globally, without a mutex, the program could suffer from race conditions as multiple workers try to dequeue tasks at once. So, any change to this shared queue must flanked by locking and unlocking the mutex.

However, in order to wake up workers or put them to sleep based on the mutex, I needed a condition; thus, I created the condition, `queue_cond`, to help synchronize the threads. When a worker checks the `request_queue` and it's empty, it waits on the `queue_cond`, sleeping until the queue has tasks. When a new download task is enqueued, I use `pthread_cond_signal(&queue_cond)` to signal one waiting worker thread. I also broadcast this signal when queueing the termination signal/poison thread to make sure all worker threads terminate.

I also created one more condition, `all_done_cond`, which is used by worker threads to notify the boss thread when all active download requests have been processed. The main thread waits on this condition after enqueueing all tasks. Once `active_requests` reaches zero, the worker who discovers this signals the `all_done_cond`. The boss can then clean up all threads and resources.

Poison Pill

I saw this term often come up on Piazza and did some research into it. I chose to use this method because it was simple and intuitive, and also allows for a graceful shutdown, as workers only pick up the poison pill from the queue like any other task once they're done with their own tasks. For my poison pill, I use a special `enqueue_termination_signal` method to queue an empty string as a task. The choice of using the empty string was arbitrary; it could be any uniquely identifiable task. I used a separate method to queue the poison pills as my normal queueing method would increment `active_requests`, which I did not want to happen for the termination signals (this would cause `active_requests` to never hit 0 and would cause the client to hang).

Testing

My testing for the multi-threaded client was much less methodical. I again used `make all_noasan` and `valgrind` to check for memory leaks. I also used Miguel Paraz's `gftestserver.py` just to set up a server to test client request against. But the majority of my testing came from `printf` statements that I used throughout the code. I printed out every task that was queued, every thread that was initialized, what request they dequeued, and whether `active_requests` was being properly incremented and decremented. I struggled a lot with the termination signal/poison pill part of the code, but I used print statements to make sure the worker threads were receiving the signal, and then handling the signal properly. After the poison pill successfully caused all the threads to exit, I was able to join all the threads in my boss thread, and clean up all leaks.

Multi-Threaded GetFile Server

Flow of Control

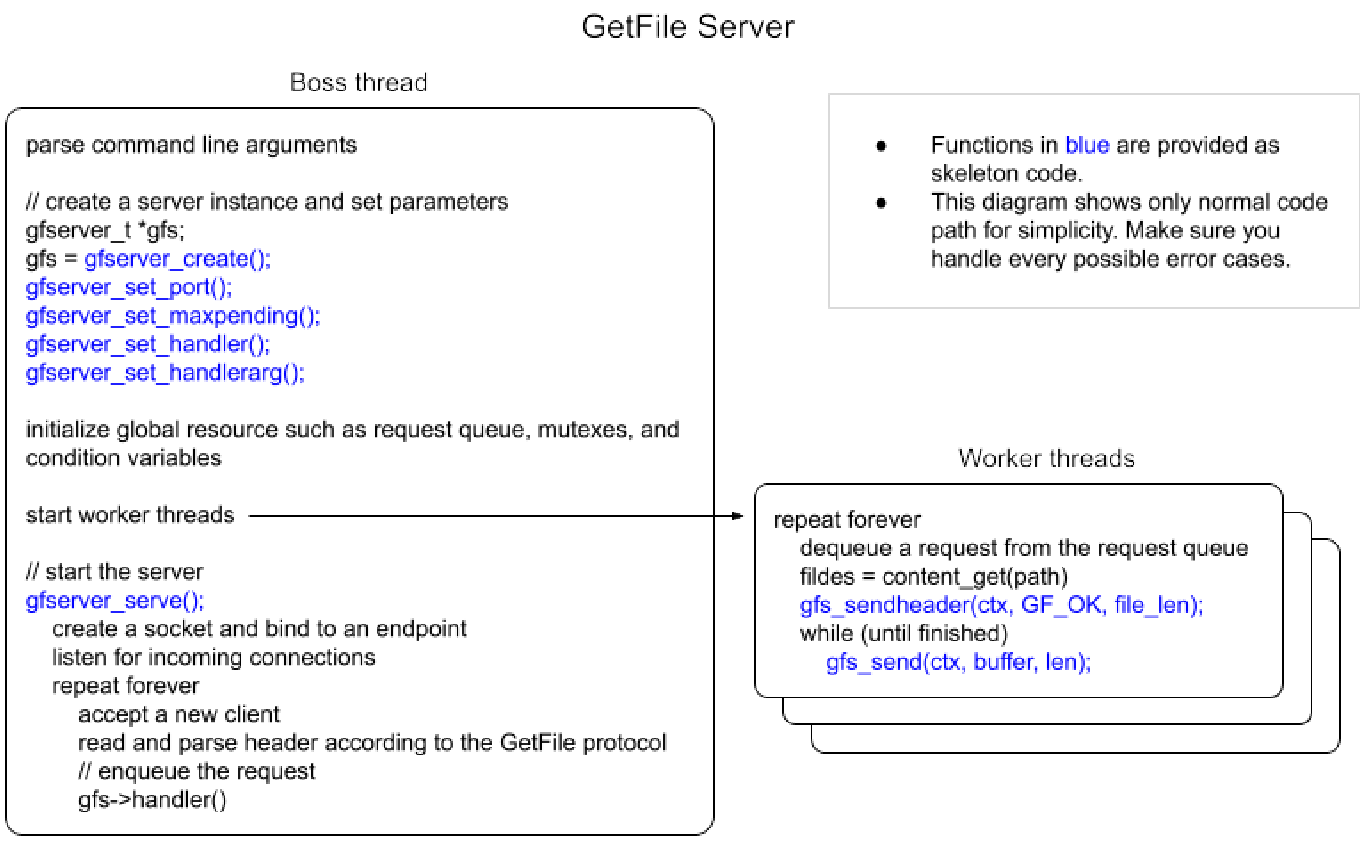
The Multi-Threaded GetFile Server first initializes the server with `gfserver_create`, and sets various configurations, such as the port, the number of threads (`nthreads`), max pending threads, the handler callback, and the handler arg. It calls `content_init` which parses the content the server can send.

The boss thread initializes a pool of threads (I used an array) and a global task queue (again using the `steque` library, and protected by the `queueMutex`). The boss then starts the worker threads (creating `nthreads` number of threads), and then calls `gfserver_serve`, which makes the server run and listen forever.

The `gfs_handler` handles the get requests from the clients, allocating memory for the request and copying the path to ensure each request has its own copy. The handler also locks the mutex, enqueues the request, unlocks the mutex, and then signals a thread with the `queueCond`. It also transfers ownership of the context from the boss thread to the request object.

Each worker thread, similar to the client implementation, waits for the `queueCond` in an infinite loop. When the queue is no longer empty, it can dequeue a request and unlock the `queueMutex`. It then gets the file descriptor of the content with `content_get` and duplicates it for exclusive use by the thread. If all is well, it sends a `GF_OK` header with the file length, and then starts sending bytes of the file in chunks until all the bytes are sent. It then cleans up that particular request, and continues the loop, ready to pickup another request.

Below is the instructor-provided high level implementation diagram of the boss and worker threads for the multi-threaded getfile server (link to original file here: https://docs.google.com/drawings/d/1a2LPUBv9a3GvrrGzoDu2EY4779-tJzMJ7Sz2ArcxWFU/edit):



```
parse command line arguments

// create a server instance and set parameters
gfserver_t *gfs;
gfs = gfserver_create();
gfserver_set_port();
gfserver_set_maxpending();
gfserver_set_handler();
gfserver_set_handlerarg();

initialize global resource such as request queue, mutexes, and
condition variables

start worker threads

// start the server
gfserver_serve();
    create a socket and bind to an endpoint
    listen for incoming connections
    repeat forever
        accept a new client
        read and parse header according to the GetFile protocol
        // enqueue the request
        gfs->handler()
```

- Functions in blue are provided as skeleton code.
- This diagram shows only normal code path for simplicity. Make sure you handle every possible error cases.

```
repeat forever
    dequeue a request from the request queue
    fildes = content_get(path)
    gfs_sendheader(ctx, GF_OK, file_len);
    while (until finished)
        gfs_send(ctx, buffer, len);
```

Design Choices and Tradeoffs

Boss-Worker Pattern

Same as for the multi-threaded client. The implementation of this pattern for the server is actually even a bit simpler, since the instructors did not require us to handle `SIGINT` or `SIGTERM`, so the boss thread doesn't have to join the pthreads or coordinate them to exit with a poison pill.

Global Mutex and Conditions

The way a mutex manages the shared queue and also uses a queue condition to signal workers when a new task is queued is exactly the same as my client implementation. It worked well and was simple to understand, so I chose to use the same pattern here.

Duplicating the File Descriptor

I ran into major issues as my threads' file transfers seemed to interfere with one another; for example, thread 1 would return file000.txt as having 1080 bytes, but then thread 2 would return file000.txt a having 0 bytes, and thread 3 would return file000.txt as having 2060 bytes. So, in order to avoid such an issue, I decided to duplicate the `file_descriptor` returned by `content_get` using `dup`. Doing this would allow multiple threads to access the same file at once, as each thread would have their own exclusive copy of the file descriptor.

Reading Buffer

The part I struggled with the most was determining how to send the bytes of the file. I soon realized a smaller buffer size would not be large enough for big files, so I decided to send the bytes in chunks. I landed on a `8192` chunk size, which is quite large, but not so large it would take up too much memory. My server sends chunks until the total bytes sent hits the file length, at which point it's done.

Testing

My testing for the multi-threaded server was pretty much identical to my testing for the multi-threaded client. I again used `make all_noasan` and `valgrind` to check for memory leaks, and I also used Miguel Paraz's `gftestclient.py` to send some requests. I mainly used `printf` statements throughout the code again to debug my code. I printed out every task that was queued, every thread that was initialized, and what request they dequeued. The most difficult part to debug was the file transfer. I was able to debug my program by printing out how many bytes my server was sending to the client and how many bytes were remaining; I was able to figure out when the buffer was overflowing, when the workers were interfering with one another, and figure out when the files were successfully transferred.

References

- Heavily referenced Beej's network programming guide (socket programming section) for the basic structure of the client and server; I used this for the warmup exercises (echo and transfer client/server), which I referenced to set up the basic structure of the clients and servers in part 1 and part 2
- String constants in C: https://eklitzke.org/declaring-c-string-constants-the-right-way
- Accessing structure members in C: https://www.tantiauniversity.com/tu-documents/Notes/Notes_Engineering_Btech_3_Accessing_Structure_Members_in_C.pdf
- How to use malloc and free: https://www.educative.io/answers/how-to-use-malloc-and-free-in-c
- How to free a struct in C: https://stackoverflow.com/questions/13590812/c-freeing-structs
- How to send data using a buffer in C: https://www.geeksforgeeks.org/buffer-in-c-programming/
- How to tokenize a string using `strtok`: https://www.geeksforgeeks.org/strtok-strtok_r-functions-c-examples/
- High-level boss and worker thread implementation (provided by instructor): https://docs.google.com/drawings/d/1a2LPUBv9a3GvrrGzoDu2EY4779-tJzMJ7Sz2ArcxWFU/edit
- A simple boss-worker pattern code example using pthreads: https://github.com/bmartini/worker-threads
- Documentation on using `pthread_cond_broadcast` and `pthread_cond_signal`: https://pubs.opengroup.org/onlinepubs/007904975/functions/pthread_cond_broadcast.html
- Poison pill explanation (high level): https://java-design-patterns.com/patterns/poison-pill/#explanation
- I also read student and instructor tips in Piazza when I was stuck