Author: Yufeng Xing Date: 2021/2/17 Update: 2021/2/18

Project README file

This is **Yufeng Xing's** 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:

Project Design

1. Echo Server & Client

For a **client**, a socket can be created by,

- **Step 1**. created with the `socket()` system call
- **Step 2**. connected to the address of the server by `connect()` system call
- **Step 3**. send and receive data by the `read()` and `write()` system call
- **Step 4**. close the socket by the `close()` system call

For a **server**, a socket can be created by,

- **Step 1**. created with the `socket()` system call
- **Step 2**. bind the socket to an address using the `bind()` system call
- **Step 3**. listen for connections with `listen()` system call
- **Step 4**. accept a connection with `accept()` system call
- **Step 5**. send and receive data by the `read()` and `write()` system call
- **Step 6**. close the socket by the `close()` system call

We should also implement the IPv4 and IPv6 features for our server and client.

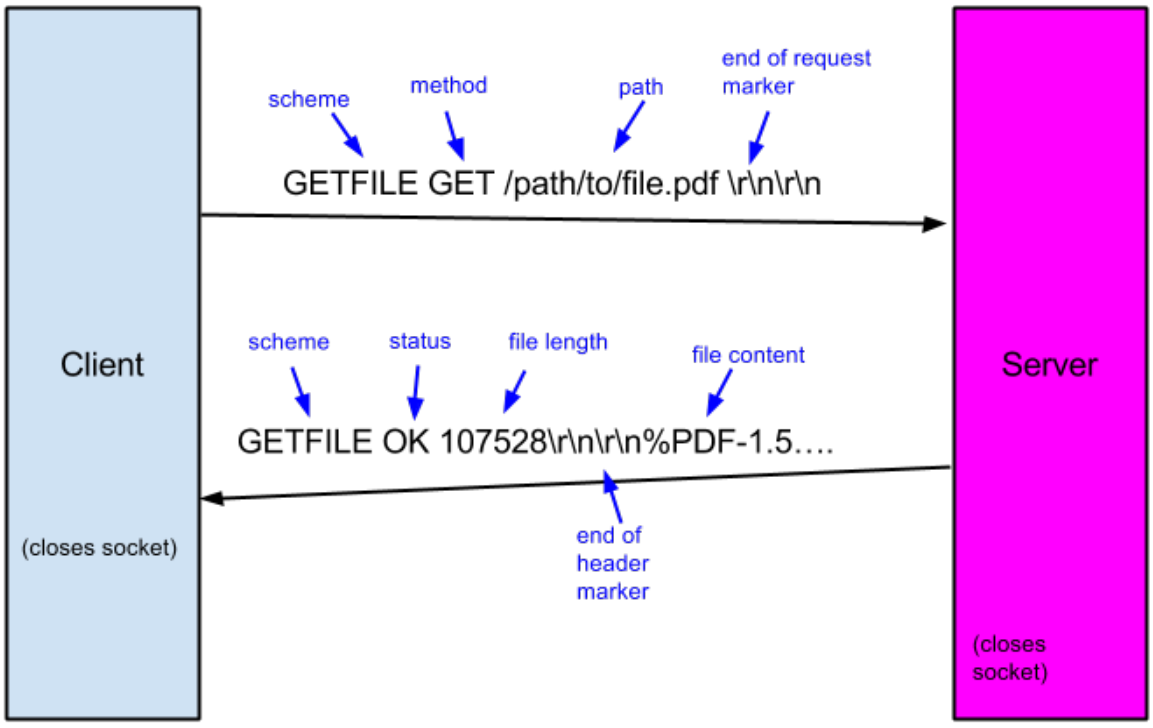
2. Transfer Server & Client

Suppose the server has a file and each client connected to this server will receive a copy of this file. In this example, we have to do the following steps,

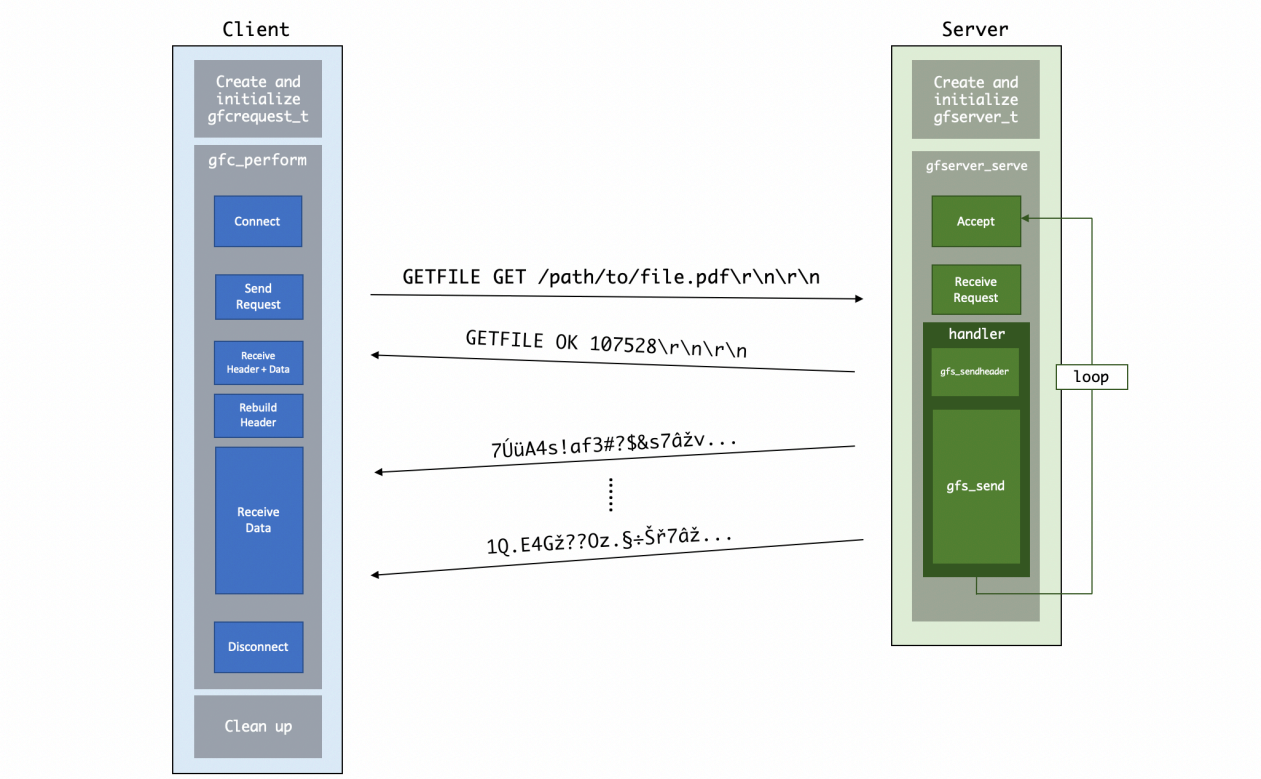
- **Step 1**. Server reads file. Because this file is stored in the server, the server should first use the function `read` to read from this file.
- **Step 2**. Server sends file. Because we have to maintain the file sending process, we have to use `send` to send the data to the server.
- **Step 3**. Client receives file. Because we have to maintain the file receiving process, we have to use `recv` to send the data to the server.
- **Step 4**. Client writes file. In the end, we have to write the data we have received in the buffer to a local file by the `write` function.

3. Part 1

Although the diagram provided in the instructions is a good one to decribe what we are going to implement for this part.



We can actually make it even better with more informations provided,



Let's start from the Getfile server. We can learn from the `gfserver_main.c` file at the beginning. The service is constructured by the following steps,

- **Step 1**. `gfserver_create` for creating a `gfserver_t` structure with the server information
- **Step 2**. Initialize the crearted `gfserver_t` structure by setting the handler callback, the port, the maxpending parameter, and the handler arguments (set to NULL in this case)
- **Step 3**. forever loop the serve based on the gfs information

Then, for the Getfile client, it follows the steps below,

- **Step 1**. `gfc_create` for creating a `gfcrequest_t` file at the beginning with the client connection information.
- **Step 2**. Initialize the crearted `gfcrequest_t` structure by setting the server (hostname), workload file path, port, write callback fuction, and the argument for this callback function.
- **Step 3**. Call the `gfc_perform` to send the request for getting the target file. Then receive data from the server.
- **Step 4**. Loop the step 1 to step 3 until all the workloads are finished.

4. Part 2

In this part, we would like to modify the `..._main` and `..._download` code files to implement a multithreading system. For the boss thread of the server,

- **Step 1**. Pick a task
- **Step 2**. Lock the shared queue
- **Step 3**. Put the task information in the queue
- **Step 4**. Broadcast to the workers to work on the task
- **Step 5**. Unlock the shared queue
- **Step 6**. Loop step 1 to step 5 forever

For the worker threads of the server,

- **Step 1**. Lock the shared queue
- **Step 2**. Wait until a signal from the boss
- **Step 3**. Grab a task from the queue
- **Step 4**. Unlock the shared queue
- **Step 5**. Get and send the file by the task information
- **Step 6**. Loop step 1 to step 5 forever

For the boss thread of the client,

- **Step 1**. Pick a request path
- **Step 2**. Lock the shared queue
- **Step 3**. Put the request information in the queue
- **Step 4**. Broadcast to the workers to work on the request
- **Step 5**. Unlock the shared queue
- **Step 6**. Loop step 1 to step 4 until all the requests are finished
- **Step 7**. Lock the remaining threads global variable
- **Step 8**. Check if remaining threads > 0, wait for a join condition variable
- **Step 9**. Join/terminate the worker threads

For the worker threads of the client,

- **Step 1**. Lock the remaining request global variable
- **Step 2**. Check whether there are requests left
- **Step 3-1**. If no request left, unlock and break the loop and broadcast a join condition variable, **end of the thread**, remaining thread minus 1 (remember to lock this variable)
- **Step 3-2**. If requests left, prepare to send the request, remaining request minus 1
- **Step 4**. Unlock the remaining request global variable
- **Step 5**. Lock the shared queue
- **Step 6**. Wait until a signal from the boss
- **Step 7**. Grab a request from the queue
- **Step 8**. Unlock the shared queue
- **Step 9**. Send the request and receive from the server
- **Step 10**. Loop step 1 to step 9 until the end of this thread

Trade-offs & Choices

1. About `\r\n\r\n` Flags

The most ideal choice for locating an rnrn flag is to check whether or not this flag is at the end of the header. However, the C language doesn't provide us a method to check the last few characters. Based on this problem, I use a trade off method of string `strstr` to check for this flag. However, there can be many other problems even though it can pass the current tests. I will talk more about this in the **Suggestions** part.

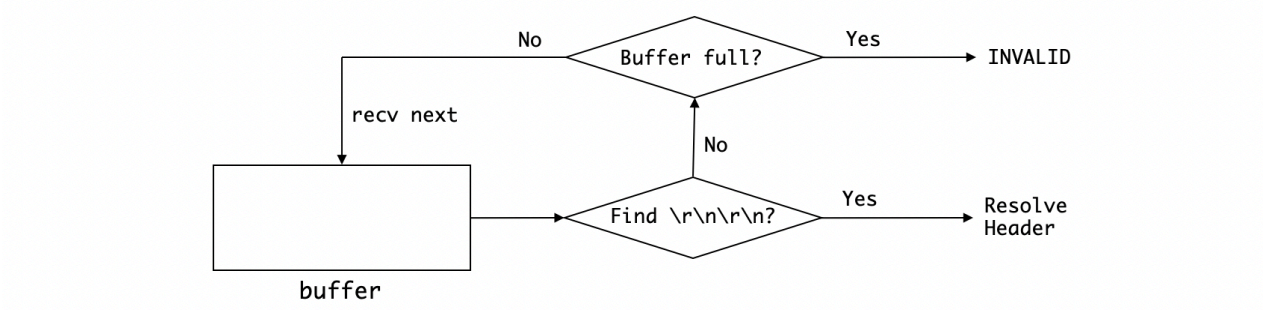
2. About the Client Mutex

As far as I am concerned, the client part for the mtgf is quite complex because we have to maintain the terminating sequence of the worker threads in this case. I choose to use 3 mutexes and it seems a little biot complex. I am not sure are there any better solutions but it seems a must for me to use these three mutex. I tried to use the item number of the shared queue to trace when to break the loop of a worker. However, I can either lock the whole shared queue or lock nothing, and this will cause some concurrency problems for me. So I finally chose to use a separate global variable of remaining requests for dealing with this problem.

Flow of Control

1. Part 1

First, let's discuss about the server. By the comments in the gfclient, we can know that the request of the header should be shorter than 256 bytes, and because we have used a buffer of 630 bytes (this is the value set exactly by the warmups), we can make sure that our buffer could contain this header properly. However, we should carefully deal with the broken header problem becasue the send/recv frunction may separate the header. I choose to loop for receiving the result until we meet the `\r\n\r\n` marker.



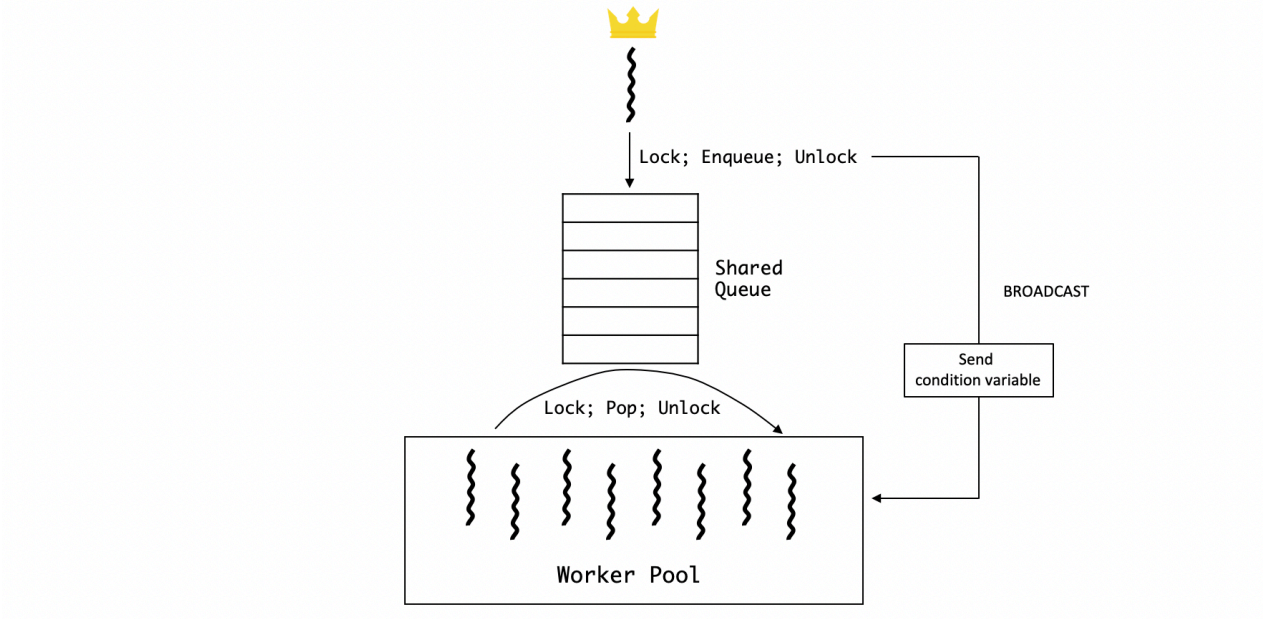
There will be a test for testing the server should properly handles a malformed request (path). It is quite easy to forget that a standard path for our server should begin with a '/' so that it can be mapped to the path of localhost based on the table in `content.txt`.

The client is much more complicated than the server. Let's see the `gfclient_download.c` file at the first glance. Similar to the server, function `gfc_create` is called to create a `gfcrequest_t` structure (gfr). Then this structure is initialized by setting the port, server, path, writefunc callback and the argument for this callback. The `gfc_perform` is then called to send the data. If the data is successfully send, we can then go to request for the next file. The path of the files that we have to request is stored in the file `workload.txt`.

The client should first send a download request based on the `workload.txt` file and then the server will respond to the client. If the status is OK, that means we can then prepare to download the data of this file. However, if we receive a not OK status (i.e., ERROR, INVALID, or FILE_NOT_FOUND), we should return an error -1, print the this status, and then continue requesting for the next file.

2. Part 2

For the server, `nthreads` numbers of threads are created as workers. The parent threads for all these threads then becomes the boss of these threads (because it can start from a different program counter). The `handler` are used to enqueue the items and after each enqueuing, it would broadcast a send conditional variable because the queue is not empty and the workers can start working. We have to use a mutex to lock this shared queue in case of any concurrency problems. At the beginning of each worker, the queue should be locked at first and then they have to wait for the send conditional variable to continue. Note that we must use `while(steque_isempty(queue))` for this because we need to check a second time in case of a race condition. When the worker receive the send conditional variable, it can then continue to send the content of the files as we wish. Note that we have to call `content_get` to convert paths.



For the client, in order to properly terminate (join) all the worker threads (child threads), we decided to use three mutexes. One mutex is for locking the shared queue (mu), another is for locking the number of the remaining requests (`remain_rqst` with mutex name `mrqst`), the last one is for locking the number of the remaining threads (`remain_threads` with mutex name `mthrd`). `remain_rqst` is for telling the other workers whether we have a request remaining or not. If there are no requests remaining, the idle threads can be terminated immediately. However, the boss thread can only join once for all the workers because it can not figure out which threads ends first. So the other workers have to wait for the join until all the threads come to ends. This also means that the `remain_threads` parameter will be 0. So when this parameter becomes 0, we can join all the workers.

Code Implementation

In order to make our client worker for both the IPv4 addresses (i.e. `127.0.0.1`) and the IPv6 addresses (i.e. `::1`), we have to use **addrinfo** structure and **gethostbyname** for resolving the hostnames.

Now, let's see how the function `getaddrinfo` works for us. To use this function, we usually have to create three variables `hints` , `res` , and `p` . The `hints` variable is an instance of the `addrinfo` structure, while `res` and `p` are two pointers that can be used to point towards a `addrinfo` structure. They are defined by,

```
struct addrinfo hints, *res, *p;
```

Before we use the `getaddrinfo` to resolve the hostname, we have to specify the values of the `hints` structure. We want to specify `ai_family` to `AF_UNSPEC` because we want to resolve both the IPv4 and the IPv6 addresses for this hostname. Also, we have to use the TCP transformation, so we have to use the stream sockets.

```
memset(&hints, 0, sizeof hints); // initialize hints with 0s
hints.ai_family = AF_UNSPEC; // AF_INET or AF_INET6 to force version
hints.ai_socktype = SOCK_STREAM; // TCP transformation
```

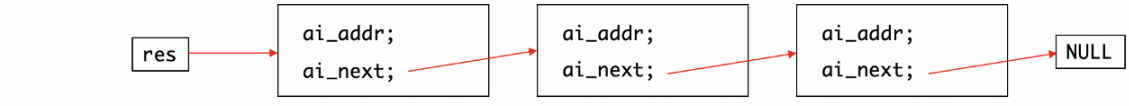
Then we can call this magic function `getaddrinfo` to resolve the hostname,

```
char *hostname = "localhost";
status = getaddrinfo(hostname, NULL, &hints, &res);
```

The return value `status` of this function is the status after resolving the hostname. `0` means that the hostname is resolved successfully. If the returned value is not zero, we can use the function `gai_strerror` to print the detailed error information.

```
printf("%s", gai_strerror(status));
```

After resolution, the `res` pointer will be pointing towards a `addrinfo` structure that stores the address information. Because this structure has an element `ai_next` , which is a pointer pointing to the next `addrinfo` structure, we actually have a linked list (or maybe we can call it a lined structure) as a result. If we loop this structure and retrieve `ai_addr`s until we meet a NULL pointer, we can get all the address information of this given hostname.



Let's see how we can open a file by the **file descriptor**, which is a fundamental way of file I/O. Commonly, it is good to use `fopen` instead of `open`. The synopsis of the `open` system call is,

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>int open(const char *path, int access, int mode);
```

where,

- `path` is the path of the file we would like to open
- `access` is the access mode for this file. Usually, we have three macros to specify for this argument, `O_RDONLY` (read-only access mode), `O_WRONLY` (write-only access mode), or `O_RDWR` (read-write access mode). We can also use or operator `|` to specify more features. For example, `O_CREATE` means to create a file if the file is doesn't exist. What's more, `O_TRUNC` means that we will clear the file before we write to it. So here are some corresponding rules,

```
r == O_RDONLY
r+ == O_RDWR | O_TRUNC
w == O_WRONLY | O_CREATE | O_TRUNC
w+ == O_RDWR | O_CREATE | O_TRUNC
a == O_WRONLY | O_CREATE
a+ == O_RDWR | O_CREATE
```

- The `mode` argument can only be used when `access=O_CREAT` and it is used to specify the future accesses of the newly created file. The commonest macros for this argument are `S_IRUSR` (means user has read permission), `S_IWUSR` (means user has write permission), and `S_IXUSR` (means user has execute permission). We can also use or operator `|` to specify more than one value.

For example, if we want to open and write to a file named `test.txt` by `open` (create one if doesn't exist), we can use,

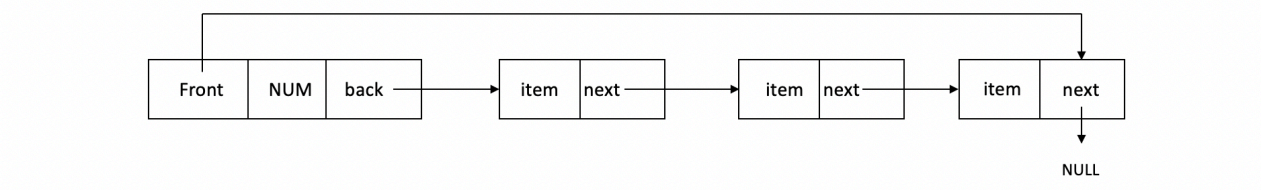
```
open("test.txt", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
```

This function returns a file descriptor (datatype int) for us to use. This file descriptor can be useful for the following write and read operations.

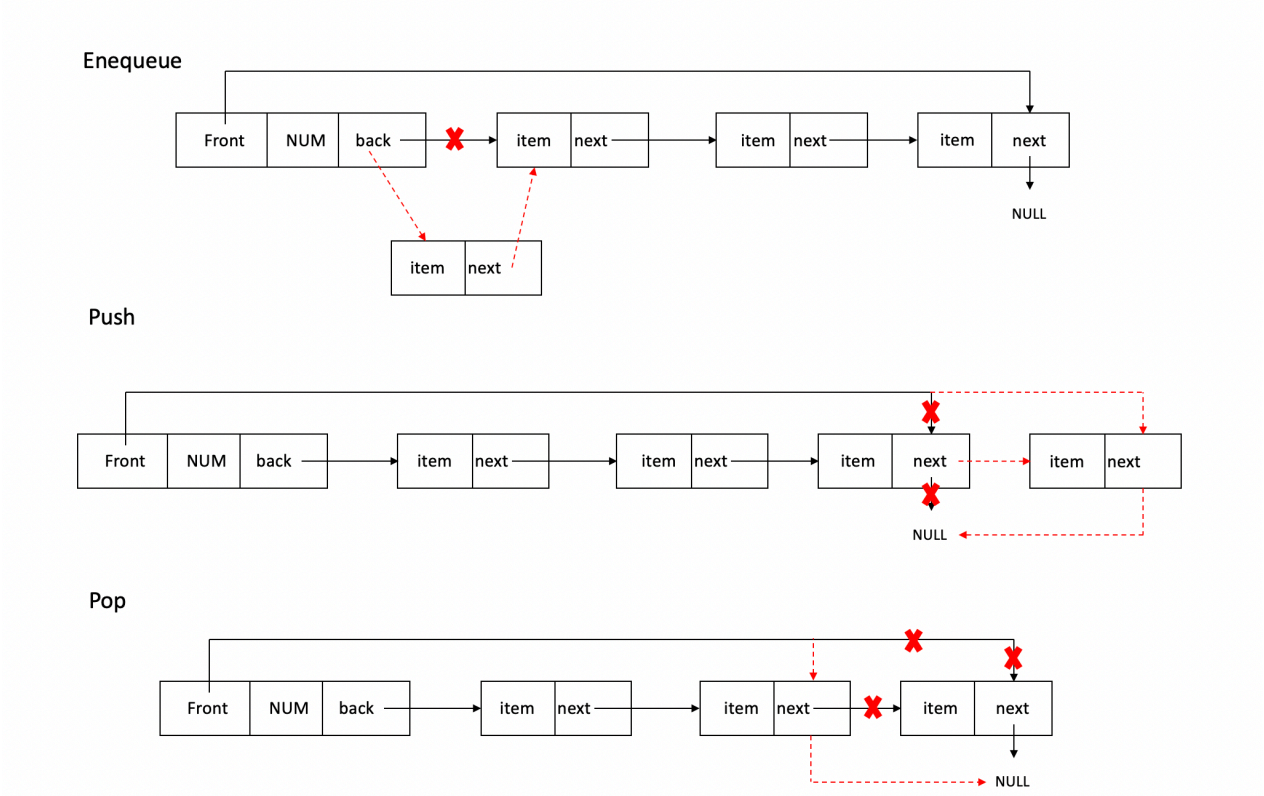
```
int fd = open("test.txt", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
```

We are also willing to talk about the multithreading gfserver and gfclient. We are going to use the boss-worker model for creating the multithreading feature. Several thread can work together for a better performance.

The `steque` data structure is very convinent to use as the shared queue in the boss-worker model, but we have to understand how the `steque` works in the first place. Here's a diagram with three elements,



The steque can work as either a stack or a queue because it provides both the enqueue function and the push function. The difference between the enqueue and the push is that the enqueue add the last item to the back, while the push function add the last item to the front. So if we insert new elements with enqueue, this will be a queue. Or if we insert new elements with push, this will become a stack. The folllowing diagram shows how these three functions work.



Testings

To test the compiled file locally, basically we have two methods.

* Method 1

We can use netcat command to test our files. Here is a list of the tests I have used for this project.

```
Normal header, received all at once with no file content
$ nc -l -p 30605 -c "echo 'GETFILE FILE_NOT_FOUND\r\n\r\n'"

Invalid header, received all at once
$ nc -l -p 30605 -c "echo 'GETFILE INVALID\r\n\r\n'"

Malformed header
$ nc -l -p 30605 -c "echo 'GETFILE INVALID\r\n\r'"

Malformed header
$ nc -l -p 30605 -c "echo 'GRABFILE OK 1000\r\n\r\n'"

Properly formed header sent with content
```

```
$ nc -l -p 30605 -c "echo 'GETFILE OK 10\r\n\r\n0123456789' && sleep 10"

Properly formed header sent with content (sleep allows you to see if your
server or client exit on their own)
$ nc -l -p 30605 -c "echo 'GETFILE ERROR\r\n\r\n' && sleep 10"

Properly formed header sent in two separate messages
$ nc -l -p 30605 -c "printf 'GETFILE ER' && sleep 3 && printf 'ROR\r\n\r\n'"
```

* Method 2

The TAs and the others students also provided some half tests or binary executable files for local testings. You can download these binary files for local testing. Note that the user may be able to execute the downloaded file, so make sure to change the mode to 777 before you run the binary program,

`$ chmod 777 {filename}`

Debuggger

As we test the program, there can be many bugs, to be honest. Here are some most common ones that I have met during this part and I think it may be helpful if I make a summary here.

1. ERROR: AddressSanitizer: SEGV on unknown address 0x000000000000 Hint: address points to the zero page.

- Reason: this bug occurs if we READ or WRITE to the memory we don't have access to. The instructor's answer on piazza helps and here's a copy "That's a null pointer dereference. If you have a structure pointer, and it's zero, accessing a field will be something close to zero... like 16 (0x10).".
- Debugger 1: find the following information like `#1, created by gfc_perform (gfclient.c:{line})` in the stderr, go to this line to see what's wrong.
- Debugger 2: if you meet this problem in the `gfclien.c` file, it can be caused if you specify a NULL value to the write function. So think about remove the statement like `set_writearg` because this value will be provided in the `gfclien_download.c` file.

2. ERROR: LeakSanitizer: detected memory leaks

- Reason: this bug is because you malloc a memory for a variable but didn't free it before exit the process
- Debugger: find the following information like `#1 0x55a5943d42d5 in .../workspace/gfserver.c:{line}\n` in the stderr, go to this line to see what's wrong.

3. Failed to send: Broken pipe\ngfc_perform: Bad file descriptor\ngfc_perform returned an error -1 (gfc_get_status shows 3)

- Reason: This error is caused when you are trying to send stuff to the client after it closes the connection.
- Debugger: Think about finding the error in the `gfs_send` function. You may send some information with the contxt that you have aborted.

4. Failed to send: Connection reset by peer\ngfc_perform: Bad file descriptor\ngfc_perform returned an error -1 (gfc_get_status shows 3)

- Reason: This error is caused when you are trying to receive stuff from the server after it closes the connection.
- Debugger: Go to find out what error caused the server disconnected.

5. AssertionError: Expected a response with INVALID as the status to a malformed request.

- Reason: This is caluse by the malformed request test. In this test, the scheme, method, path of the GETFILE request can be wrong.
- Debugger: Carefully handle all the malformed situations including path not start with '/', not a GETFILE scheme, not a GET method, no \r\n\r\n marker in the end, and etc..

6. AssertionError: The client is taking too long (probably hung)

- Reason: this is one of the commonest error and this is cause by a client that doesn't exit properly. You may pass your local tests but
- Debugger 1: I have found that the script written by Tony Mason is useful. So I save a copy here,

> (1) Buffer handling issues. I've seen cases where people fail to account for data following the header; thus the client discards bytes that are valid data and then waits for the data to arrive, but since the server already sent it, the client waits for the server.
>
> (2) Assuming the server will close the connection when the file is fully transmitted. **This is not a safe design**. If your server closes the connection before the client has received all the data, the pending data queue on the client will be flushed. This will work locally because there's really no lag - your client will be scheduled and run quickly. Equally frustrating about this one is that most students write their server to close the connection after they send the file, which means the interop servers won't catch this particular failure. **Web servers do not close connections immediately**. One reason is efficiency - connection set-up (especially over the internet) is an expensive operation. In fact, this is why we have TCP Fast Open (TFO) and even QUIC (Quick UDP Internet Connection) which tries to overcome TCP limitations by constructing a new (non-TCP) transport protocol over UDP (the *unreliable* datagram protocol). Every semester I remember (during Project 1) that we need to add a test for this case, where a client sends multiple requests on the same server. If we could inject a scheduling delay between client and server, so the client waited *before* doing another recv, we'd generate the pipe flushing behavior of TCP (though that also depends upon the specific implementation, so probably just two back-to-back requests would be enough).
>
> (3) Uninitialized variables. Since your code is not running in a mixed tenant environment, while Gradescope's is, you will find that behaviors in your environment (e.g., the order of execution and thus the contents of random stack locations) will differ substantially. This is one reason we run **valgrind** against your code, as it does a good

> job of identifying cases where you use uninitialized values for making decisions ("conditional jump on uninitialized value").
>
> (4) Thinking that binary buffers will be '\0' terminated. This is a **binary protocol.** Nothing here specifies that you will receive C style strings. It is certainly not required and there's definitely no reason to assume that an arbitrary protocol would favor the implementation model for strings from a single language.

- Debugger 2: Try to download the half-test from the piazza that are shared by the TAs and other students. It can be helpful because your client may not work for other server.

7. handler did not ctx to NULL

- Reason: ctx should be set to NULL after configuration because this is required by the `gfserver.o` file
- Debugger: In the end of the handler call, the ctx pointer should be set to NULL.

Suggestions for additional tests

I have to say that I think the rnrn flag is not tested properly. For example, if the server returns,

```
$ nc -l -p 30605 -c "echo 'GETFILE ERROR\r\r\n\r\n'"
```

If we use `strstr`, the test can pass and this will be treated as a correct header. However, this is invalid because `INVALID\r` is not a correct status. The true output should be an error of invalid header with status `INVALID`, however, in my case it is `ERROR` request. I think the Gradscope doesn't test for this case and there should be a test like this. Here's a false result,

```
Status: ERROR
```

The true result should be,

```
Status: INVALID
```

Suggested revisions to the instructions

In the instructions for the transfer warmup. It is said that,

> Beware of file permissions. Make sure to give yourself read and write access, e.g. `S_IRUSR | S_IWUSR` as the last parameter to open.

However, as I have talked about in the code implementation part, the `S_IRUSR | S_IWUSR` can only be useful if we use the `open` call. I find this confusing because the instructions doesn't force us to use the `open` call, which means we can also use `fopen` to read from the file. But I think this instruction implies that we should use `open` instead of `fopen` (I don't know whether `fopen` will cause some permission problems or not because I didn't implement this). But I think it can be clearer for us if the instructions specify which function to use for us.

Reference

- [1] Errno Manual
- [2] Bind Manual
- [3] getsockname Function Usage
- [4] round-robin DNS
- [5] fopen vs. open
- [6] Server and Client Example Code
- [7] Guide to Network Programming
- [8] Linux Sockets Tutorial
- [9] SO_REUSEADDR Debugger
- [10] Piazza