p1_README

Warm-up: Sockets

Design and Flow

Overview

The Echo Client-Server and File Transfer will be grouped together in this single section.

A quick note, although the project README explicitly references `send` and `recv`, this entire project uses `write` and `read` instead.

- `write` and `read` behave essentially the same as `send` and `recv` with the flags parameter set to `0` [1, 2].
- Since not much complexity regarding data transmission is required (in this implementation anyway), it is much more convenient to call `read` / `write` than to have `recv` / `send` with the flag parameter of `0`.

For both warm-ups, the following workflow is followed [3]:

1. Server-side (Socket, Bind, Listen, Accept, Read/Write)
2. Client-side (Socket, Connect, Read/Write)

The only key difference between the Echo Client-Server and File Transfer would be the need to loop while `read()`-ing or `write()`-ing, as there is no guarantee that a large file will be transferred all at once. The following pseudocode demonstrates such a strategy:

```
while (1) {
        /* loop indefinitely */
        bytes_read = read();
        if (bytes_read == 0) {
                break;
        }
```

```
        handle_bytes_read(bytes_read);
}
```

The above code shows the `read()` function encapsulated within an infinitely looping `while` loop. This is due to the fact that the caller of `read()` does not know how much data to expect; this is true of `transferclient.c`.

Similarly, the `write()` function will also be encapsulated within a loop that ensures all bytes will be sent out. It is generally considered good practice to wrap `read()` and `write()` in loops, as these functions make no guarantee that all bytes will be sent or received at any given point in time.

Note that the `while` loop terminates when `bytes_read` is equal to `0`. In the usage of `read`/`write`, the value of `0` will be returned when the associated connection is closed [2]. Therefore, this is used to inform the `transferclient` (in this case) that no more bytes are expected.

In later parts of this project, the expected number of bytes is made known ahead of time. As such, the following pseudocode can be used:

```
while (total_bytes_received < bytes_expected) {
        curr_bytes_received = read();
        if (curr_bytes_received == 0) {
                break;
        }
        handle_bytes_read(curr_bytes_received);
        total_bytes_received += curr_bytes_received;
}
```

The above is adapted from Beej's Guide to Network Programming [4]. This guide itself is used for the majority of the warm-ups and will be referenced accordingly in the Code References of Warm-up: Sockets.

The line below is required to allow the same socket address and port to be reused. This is primarily important for submission to Gradescope:

```
setsockopt(sockfd, SOL_SOCKET, (SO_REUSEADDR|SO_REUSEPORT), &(int){1},
sizeof(int));
```

Testing

1. Testing the Echo Client-Server is rather straightforward. The server should echo any input string without any differences.
2. Testing the File Transfer involved generating random data within a text file, then verifying that integrity is not compromised when sending data from the server to the client. Text files of varying sizes were generated using a Lorem Ipsum generator [5].

Code References

A majority of the warm-ups referenced Beej's Guide to Network Programming [4]. The portions referenced will be compiled in this section. The exact code used will be shown here for ease of reference.

1. Setting up the socket to accept IPV4 and IPV6 addresses). This references the `getaddrinfo()` section of Beej's Guide to Network Programming.

```
/**
 * echoclient.c :: main
 * transferclient.c :: main
*/
struct addrinfo hints;
struct addrinfo* server_info;

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;

char portstr[6];
sprintf(portstr, "%d", portno);
getaddrinfo(hostname, portstr, &hints, &server_info);

int sockfd = socket(server_info->ai_family,
                                        server_info->ai_socktype,
                                        server_info->ai_protocol);

freeaddrinfo(server_info);
```

2. Binding the socket (server-side). This references the `bind()` section of Beej's Guide to Network Programming.

```
/**
 * echoserver.c :: main
 * transferserver.c :: main
 */
struct sockaddr_in6 server_addr6
struct sockaddr_in6 client_addr6;
socklen_t addr_len = sizeof(struct sockaddr_in6);
memset(&server_add6, 0, sizeof(server_addr6));
server_addr6.sin6_family = AF_INET6;
server_addr6.sin6_addr = in6addr_any;
server_addr6.sin6_port = htons(portno);
```

Part 1: Getfile Protocol

Design and Flow

Overview

The `gfclient` and `gfserver` implementations share a few similarities. However, due to the inability to edit any custom shared library between the two, functions that could serve the same purpose in both the client and server had to be repeated in the individual `gfclient.c` and `gfserver.c` files. A notable example would be the auxiliary function `match_regex`, which is repeated word-for-word in both files.

The `gfclient` and `gfserver` files both define a number of constants. Again, much of this is repeated. The constants will be delineated here. Note that the maximum sizes are occasionally padded by an arbitrary amount. For example, although the maximum valid header size is 53, the constant `MAX_RESPONSE_HEADER_SIZE` is defined as 60.

> The additional padding hardly incurs a large memory cost and simply serves as a more defensive stance.

- `#define SCHEME_SIZE 8`
  - "GETFILE" is the only correct scheme + 1 for `'\0'`.
- `#define METHOD_SIZE 4`
  - "GET" is the only correct method + 1 for `'\0'`.
- `#define MAX_FILEPATH_SIZE 4097`
  - 4096 on Unix systems + 1 for `'\0'`.
- `#define MAX_FILELEN_SIZE 25`
  - Up to 64-bit decimal unsigned value (as per project requirements)
  - This would be the length of `18446744073709551616` + 1 for `'\0'`.
  - This value comes up to 21 but is arbitrarily increased to 25.
- `#define MAX_STATUS_SIZE 16`
  - The maximum length of an element of {`OK`, `FILE_NOT_FOUND`, `ERROR`, `INVALID`} + 1 for `'\0'`.
  - This value comes up to 15 but is arbitrarily increased to 16.

- `#define MAX_RESPONSE_HEADER_SIZE 60`.
  - Add `SCHEME_SIZE`, `MAX_STATUS_SIZE`, `MAX_FILELEN_SIZE`, and 4 (length of `\r\n\r\n`).
  - This value comes up to 53 but is arbitrarily increased to 60.
- `#define MAX_REQUEST_HEADER_SIZE 4113`
  - Add `SCHEME_SIZE`, `METHOD_SIZE`, `MAX_FILEPATH_SIZE`, and 4 (length of `\r\n\r\n`).
- `#define TIMEOUT 4000`
  - Arbitrarily set timeout to 4 seconds. This is admittedly influenced by one of the test cases written by cparaz3 in their Python test suite [6].

The maximum request/response header sizes informs the server and client respectively when to terminate. If the number of bytes received exceeds the maximum possible length of a valid header but the terminator sequence `\r\n\r\n` is still not received, there is really no point in continuing as the header is already guaranteed to be invalid.

Helper functions `assert_valid_gfs` and `assert_valid_gfr` are implemented to check if an input argument of type `gfserver_t**` or `gfcrequest_t**` is `NULL`. These assert functions are called at the start of every function that takes in `gfserver_t**` or `gfcrequest_t**` as an argument. It was noted later in course discussions that this is not necessary, as it is guaranteed that non-`NULL` values will be passed to the relevant functions. This was left in the implementation anyways, as it serves as a defensive layer of checking.

Both the `gfclient` and `gfserver` utilise a `struct` that is passed around by the user (the getter/setter functions require a `gfcrequest_t` and a `gfserver_t` `struct` respectively, for example). These could be thought of as "instances" of a client request or an active server; these instances therefore must contain the necessary "attributes" or information for the library functions to act on. For both implementations of `gfclient` and `gfserver`, all user `GET`-able and `SET`-able information will of course be stored in their respective `struct`s. The only other information deemed necessary would be the socket file descriptor itself.

gfclient

The `gfclient` implementation performs a user request in these three main steps:

```
client_socket_connect();
build_and_send_request();
handle_server_response();
```

- Quite simply, a connection is made, a request is sent out, and an incoming response is handled.
- The subsequent sections will take a greater look at the individual functions delineated above.

client_socket_connect

This function simply encapsulates socket-related code. More specifically, the `socket` and `connect` functions are called within `client_socket_connect`. This section is similar to that of the warmup, and will not be rehashed here.

build_and_send_request

This function builds a request of the format `GETFILE GET <path>\r\n\r\n` and loops until all request bytes have been sent out. The notion of looping until all bytes have been sent is already covered above.

handle_server_response

This is where the bulk of the server code lies. The figure below preemptively shows the idea behind handling **header data with the possibility of trailing content data**. More details will be given when the pseudocode is discussed:

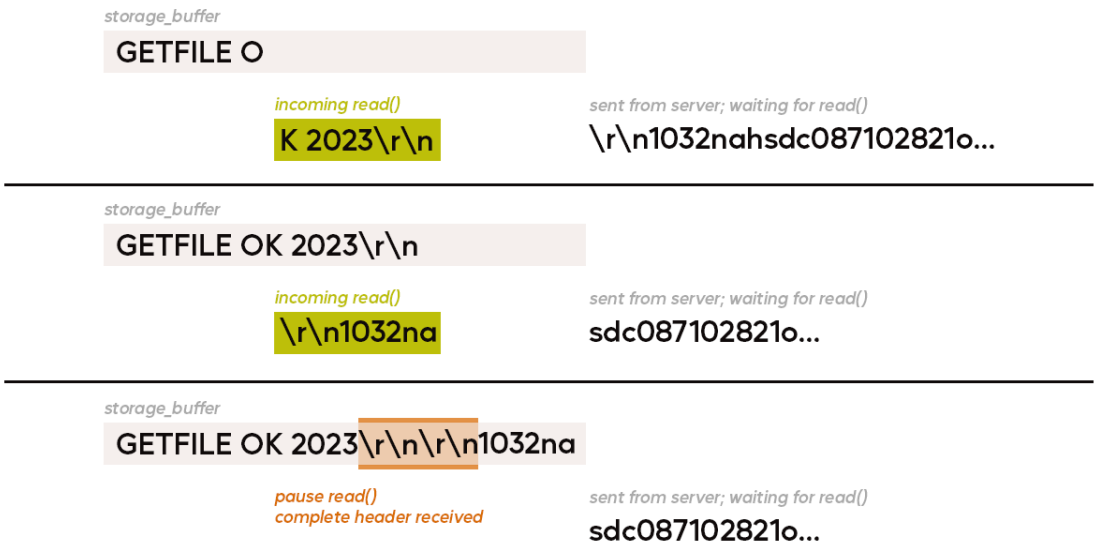

Figure 1: Hypothetical scenario demonstrating the possibility of content data trailing behind the header terminator sequence `\r\n\r\n`. It is impossible for the client to demarcate the exact point where the header stops without first receiving data indiscriminately.

The pseudocode for handling an incoming server response is given below, though note that the actual code implementation breaks this up into smaller helper functions:

```
/* read incoming header */
header_buffer.init();
while (!header_buffer.is_terminated || header_buffer.len < max_header_len)
{
        /* loop till a header terminator encountered OR the max header
length reached */
        incoming_data = read();
        if (incoming_data == 0) { break; }
        header_buffer.append(incoming_data);
        if (header_buffer.is_terminated) { break; }
}

/* assert valid header */
regex_match(header_buffer, header_regex_pattern);

/* extract header data */
status = header_buffer.status;
if (status != "OK") { exit(); }
```

```
header_data = header_buffer.header_data;
expected_content_len = header_buffer.expected_content_len;
trailing_content_data = header_buffer.trailing_content_data;

header_handler_func(header_data); /* set by the user */
content_handler_func(trailing_content_data) /* set by the user */
content_received = trailing_content_data.len

/* receive remaining content data */
while (content_received < expected_content_len) {
        incoming_data = read();
        if (incoming_data == 0) { break; }
        content_handler_func(incoming_data); /* set by the user */
        content_received += incoming_data.len;
}
```

Comments:

- While reading for an incoming server response, the first item encountered should be the response header. The client will loop continuously until:
  - (i) a proper header terminator sequence (`\r\n\r\n`) is encountered,
  - (ii) the maximum possible header length is exceeded,
  - (iii) the server closed the connection (`incoming_data == 0`), or
  - (iv) some error is encountered (not shown above)
- The client will also timeout if the server has not sent any data for **four seconds** but none of the conditions above have been met (further details on this choice under the Testing section of Part 1).

  > `read()` returns a 0 only when the connection is closed. Otherwise, it blocks indefinitely unless a non-blocking socket is specified. As such, there is no way to tell if a server has stopped sending data but refuses to close the connection properly. A timeout is used to circumvent the possibility of blocking on an ill-intentioned server.

- An assertion for a valid header is first made with regex pattern matching. This ensures that downstream code can work with the assumption that the header **must** be valid -- no extra checks will be required after this point.

> Regex pattern matching is used for its ease and relative strictness. However, note that the regex pattern is compiled with the `REG_EXTENDED` flag, which allows the use of word groups in C.

- As mentioned above, there is a possibility that content data may be included when trying to read the entire response header (termed: "trailing content data"). This is handled by simply separating the response buffer into a header and content section.

  > Initially, the intention was to store the trailing content data into a separate buffer and process it alongside the remaining content data. In other words, read up to the header terminator sequence and simply funnel the rest of the incoming data into a separate `content` buffer, for instance. This would result in a clean separation between parsing the header and content section.
  >
  > However, this proved to be much more involved than expected -- `strcpy` needs to be used to funnel the already-read trailing content data into a separate buffer, which then needs to be appended with the remaining content data. This would also introduce yet another design decision: how much content data should be read before calling `content_handler_func`?
  >
  > This idea was scrapped in favour of the easier (but admittedly dirtier) method of handling the header, trailing content, **then** the remaining content data.

- Once the trailing content data has been handled, reading and parsing the remaining content data is trivial, and follows concepts already established and covered in the warm-ups.
- It is possible that the server sends more content data than expected. For example, the server first establishes a file length of `4` but sends the content data `ABCDE`.

  > The client simply ignores any data received once it has read up to the expected length. This is achieved by parsing the minimum of the number of remaining bytes and the number of incoming bytes. Specifically, `MIN(curr_bytes_read, remaining_bytes)`.

- The client does not use `strtok` to distinguish content data from header data. This design choice in particular was informed by testing with cparaz3's Python

test server [6].

> If the server sends a perfectly valid response `GETFILE OK 2\r\n\r\n\r\n`, `strtok` does not correctly identify the last two characters `\r\n` as content data.
>
> In fact, during implementation, it was noted that `strtok` does not work with substring delimiters [7, 8]. Therefore, `strstr` alongside pointer arithmetic was used to properly split a string based on the `\r\n\r\n` delimiter.

gfserver

The `gfserver` implementation performs a server response in these three main steps:

```
server_socket_bind_listen();
while (1) {
        accept();
        read_and_parse_header();
}
```

- Briefly, the server listens for incoming client requests indefinitely. A valid request should contain only a request header.
- The subsequent sections will take a greater look at `server_socket_bind_listen` and `read_and_parse_header` in greater detail. `accept` is simply used to accept an incoming connection, similar to that of the warm-ups and thus will not be covered here.

server_socket_bind_listen

This function, as the name implies, performs the following socket-related code: `socket`, `bind`, and finally `listen`. The server then loops and waits for any incoming requests with `accept`.

read_and_parse_header

The pseudocode for handling an incoming client request is given below:

```
header_buffer.init();
while (!header_buffer.is_terminated || header_buffer.len < max_header_len)
{
        incoming_data = read();
        if (incoming_data == 0) { break; }
        header_buffer.append(incoming_data);
        if (header_buffer.is_terminated) { break; }
}

/* malformed header handling not shown */
regex_match(header_buffer, header_regex_pattern);
check_client_still_connected();
header_data = header_buffer.header_data;
header_handler_func(header_data);
```

A few comments:

- The overall flow is relatively similar to that on the `gfclient` end. Only one major difference will be pointed out here; the server checks if the client remains connected after a valid header is received.

  > This is to handle the event where the client closes the socket **after** sending a valid header. It is therefore not possible for the server to respond to the client accordingly -- the server should simply terminate this particular context.

- In the project requirements, it is stated that `gfserver` should still respond if the client's request was malformed or incorrect. In other words, if an incoming client request is invalid (for example), the server should respond with a header `GETFILE INVALID \r\n\r\n`. The handler should be called **only when the header is deemed to be valid** -- the handler then has the responsibility of sending an `OK` or `FILE_NOT_FOUND` header using `gfs_sendheader`.

  > It is stated that `gfs_sendheader` must **only** be called by the handler, though the server itself still needs to respond with an `INVALID` or `ERROR` header. In fact, since the set handler takes a path, it **must be the case that a path must at least exist in the client's response before calling the handler function**.

> This led to a lot of confusion (more in the Feedback section). How is it possible to send an `INVALID` or `ERROR` header when `gfs_sendheader` must only be called within a handler, which *must only* handle an `OK` or at the very least `FILE_NOT_FOUND` status?
>
> Over time, some of the class have arrived at one solution -- simply have another function that serves to send a response header and *name this something else*. In this implementation, a function `sendheader_private` is defined and `gfs_sendheader` simply calls `sendheader_private`. `sendheader_private` can then be called in other parts of `gfserver` where sending an `INVALID` or `ERROR` response header is required.
>
> This method feels extremely contrived, as it requires an additional function to do essentially the same task -- send a response header. In previous iterations of `gfserver`, I took the comment "only be called from within a callback ..." to be **directed to the user** of the `gfserver` library. In other words, the user of `gfserver` must only call `gfs_sendheader` within the handler, and is not allowed to call `gfs_sendheader` explicitly like `gfserver_serve` or `gfserver_create`.
>
> As such, whether or not `gfs_sendheader` can be called *within* the `gfserver` library should be the prerogative of the library implementer (i.e. myself). However, as of 12 September, the general consensus on both Piazza and Slack would be to simply have another function that handles sending headers as well. Call that within the library implementation, and leave `gfs_sendheader` implemented but untouched.

- The active server will also timeout if nothing is being sent by the client in **four seconds**.

Testing

1. cparaz3's Python test server and test client [6] were used as a backbone, though edited to send valid requests and responses. For example, the server sends a response such as `GETFILE FILE_NOT_FOUND\r\n\r\n` to check if the client is even able to receive a server response in the first place.
2. Once simple messages were ascertained to be correctly received, `gfclient` was tested for its ability to handle header **and** content data, be it trailing or

otherwise. Dummy valid server responses such as `GETFILE OK 4\r\n\r\nABCD` were used for testing.

3. `gfclient` and `gfserver` were then tested together, using the provided `gfclient_download` and `gfserver_main` files. In this phase, images requested by the `gfclient` should be downloaded correctly into a separate directory.
    - This stage surfaced a perplexing bug, where **one** garbage byte would **always** be prepended to the content data. For example, the actual data received would amount to something like `GETFILE OK 4\r\n\r\n?ABCD`, where `?` is a random byte.
    - This issue did not arise when testing with the Python test server, which led to the belief that the implementation of `gfserver` was buggy.
    - Next, the header and content being sent by `gfserver` was checked **immediately** before calling `write`. No issues with the header nor content were observed. However, immediately checking the data received by `gfclient` always revealed the presence of the prepended garbage byte. The systemic issue was seemingly caused by `write` or `read`.
    - To investigate the garbage byte further, the VSCode extension Hex Editor [9] was used to manually inspect the first byte received in every server response. The value of the garbage byte itself was noted to be random, which led to the suspicion of memory mismanagement.
    - Eventually, it was discovered that the header buffer (server-side) was not initialised properly, which led to garbage bytes trailing the header terminator sequence `\r\n\r\n`.
    - This was not caught when parsing the response header (client-side), but the garbage byte was **still waiting to be received**.
    - Therefore, when `read` was called when trying to parse content data, the garbage byte is first received, followed by the actual content data. This explains why the garbage byte is only ever **prepended** and never scattered in between content data.
    - This was quite simply fixed by calling `memset` on the header buffer (server-side).
4. `gfclient` and `gfserver` continued to be tested after the off-by-one bug was fixed. Images received by the client were verified firstly by checking if they are open-able (quick check), followed by a checksum comparison.

5. As a sanity check, text files and other image files were also added to `workload.txt` and `content.txt` in order to ensure that `gfclient` and `gfserver` works with other files.

Code References

This section compiles code used and referenced in implementing `gfclient.c` and `gfserver.c`.

1. Regex pattern matching was referenced from multiple sources [10, 11]. Regex pattern matching was tested using regexr (PCRE) [12], though it was not a one-to-one match with regular expressions in C. Further testing in C itself was required.
2. Multiple string handling functions, primarily from the `string.h` library, were used. Their respective references are compiled here:
   - `strstr` [13]
   - `strcat` [14]
   - `strncpy` [15]
   - `sscanf` [16]
   - `strcmp` [17]
3. While technically a string-related task, splitting a string with a substring delimiter was much more involved than the simply library functions above. The relevant snippet is displayed here for ease of reference, and the following references were adapted for use[18, 19]:

```
/* gfclient.c :: read_and_parse_header */
char * storage_buffer;
char * header_only;
char * data_only;

/* init buffers accordingly */
/* read header and trailing content into storage buffer */

char * delim = strstr(storage_buffer, "\r\n\r\n");
data_only = delim + strlen("\r\n\r\n");
int position = delim - storage_buffer;
```

```
strncpy(header_only, storage_buffer, position);
size_t headerlen = data_only - storage_buffer;
```

4. The general workflow for socket-level code was copied wholesale from the Warm-ups. The strategy of looping while `read`-ing or `write`-ing was also copied from the Warm-ups (specifically the section on file transfer).
5. The implementation for timing out (both on the client and server side) was adapted from an answer to the StackOverflow question *"poll() waits indefinitely although timeout is specified"* by user Remy Lebeau. The relevant snippet is displayed here for ease of reference:

```
/**
 * gfclient.c :: read_and_parse_header
 * gfserver.c :: read_and_parse_header
*/

/* differs from actual code for brevity */
while (total_received < MAX_HEADER_SIZE) {
        int r= poll(&fds, 1, TIMEOUT);
        if (r == -1) { return -1; }
        else if (r == 0) { return -1; }
        else if (fds.revents & POLLIN) {
                /* read incoming data */
        } else if (fds.revents & (POLLERR | POLLNVAL)) { return -1; }
}
```

Part 2: Multithreading

Design and Flow

Overview

A boss/worker pattern was used, as specified in the project requirements. In general, both the multithreaded server and client follow the same pattern illustrated below:

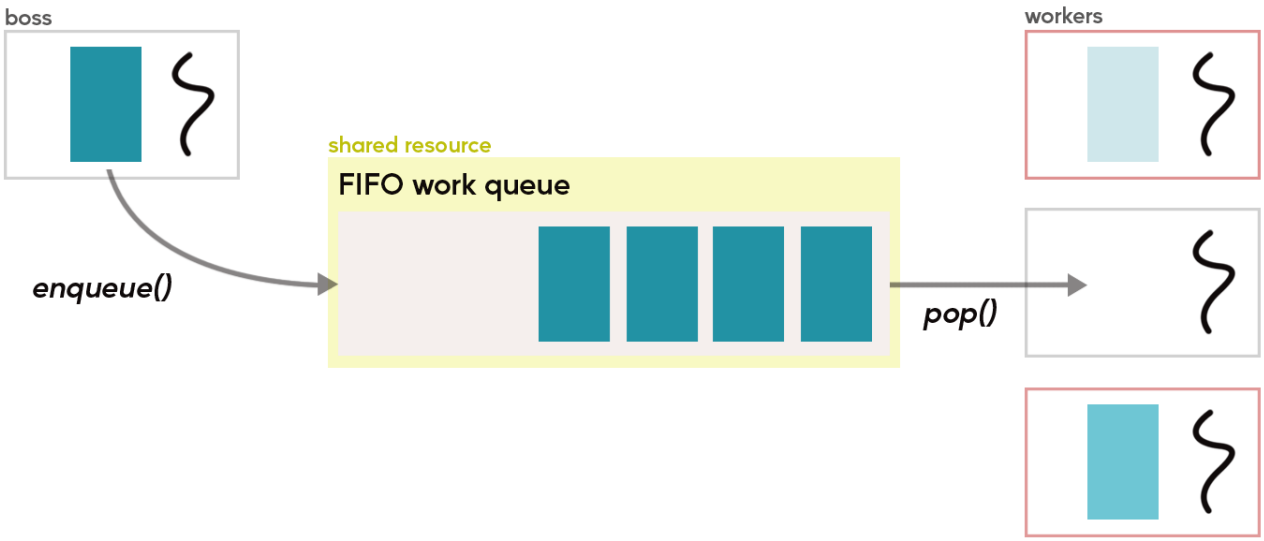

Figure 2: Overview of general boss/worker threading pattern for multithreaded server and client.

In short, a boss thread will push incoming tasks into the back of a shared work queue (First-In-First-Out). Worker threads will be responsible for removing tasks and handle them accordingly.

The only shared resource would be the work queue, which must be protected with mutexes.
Certain details differ between the server and client implementation, which will be covered shortly below.

gfclient_download

The general workflow of enqueuing and dequeuing tasks is applicable in `gfclient_download`. Importantly, there is a need to terminate worker threads only

when all tasks have been completed. This is achieved using the **poison pill** strategy [20, 21].

- For context, the worker threads loop infinitely, as they wait for any incoming tasks enqueued by the boss thread.
- When all tasks have already been scheduled, the boss thread enqueues one final signal that **all tasks have already been placed on the work queue**.
- Therefore, whenever a thread encounters this signal (termed: poison pill), it should stop looping infinitely and prepare for joining.

> The poison pill strategy was preferred over any global work counter as it does not require any more shared resources (and with it, the issues that relate to concurrency).
> Simply planting a poison pill is a straightforward manner of using already-existing tools to terminate worker threads.

In this implementation of the poison pill strategy, all threads first peek at the front of the work queue before deciding how to proceed. If a poison pill is encountered, simply terminate gracefully. Otherwise, pop the front of the queue and handle the task accordingly.

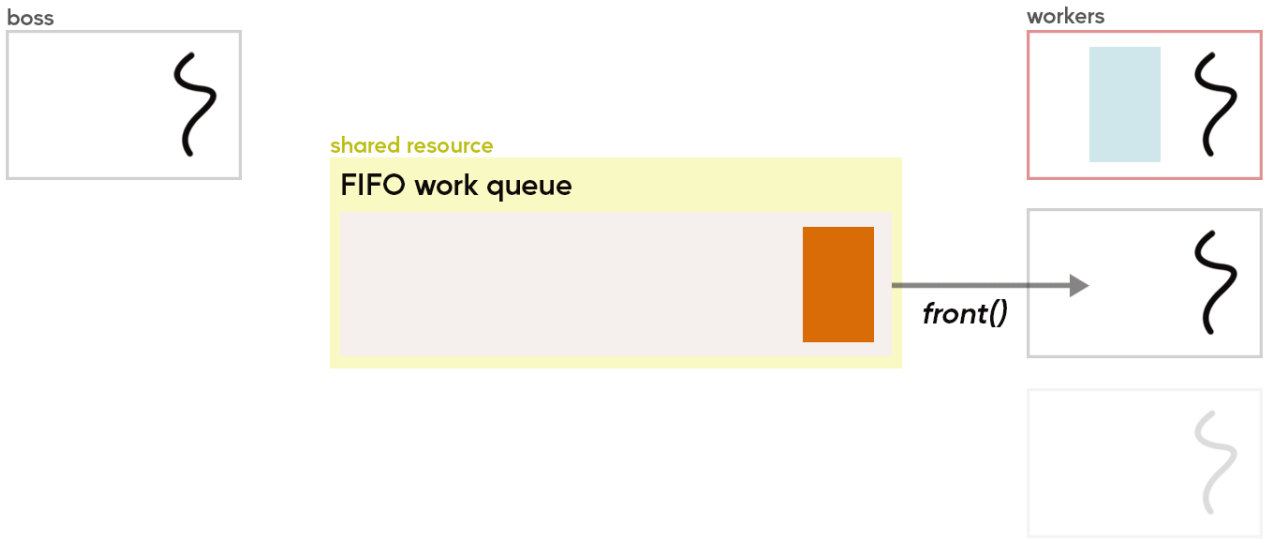

Figure 3: A worker thread encountering a poison pill (coloured in orange) before terminating. Note that the poison pill itself is not removed from the queue, to allow other workers to encounter this signal as well.

Pseudocode (gfclient_download boss)

The following pseudocode shows the flow involved in the boss thread implementation:

```
/* in boss thread */
spawn_worker_threads();

for (request in requests) {
    Lock(work_queue_mutex);
    /* begin critical section */
    work_queue.enqueue(request);
    Signal(cond_work_queue_not_empty);
    /* exit critical section */
    Unlock(work_queue_mutex);
}
/* all requests queued; push poison pill */
Lock(work_queue_mutex);
work_queue.enqueue(POISON_PILL);
Signal(cond_work_queue_not_empty);
Unlock(work_queue_mutex);

join_worker_threads();
```

A few comments:

- As the `work_queue` is the only shared resource amongst the various threads, mutex locking and unlocking must encapsulate any access to it.
- `Signal` is placed within the critical section, to ensure that the condition `cond_work_queue_not_empty` remains undoubtedly true upon the time of signalling.

  > This would mean that the worker threads waiting on this condition variable would wake up only to be placed on another queue associated with *acquiring the mutex*.
  > However, this is a trade-off considered, as the correctness of the signalling was a higher priority than the minimisation of thread waiting.

- The final `enqueue` call for the `POISON_PILL` must also be handled similarly (again, `Lock`-ing, `Unlock`-ing, and `Signal`-ling where necessary).

  > The enqueue process is copy-pasted in various segments of the boss thread code and should be abstracted into a helper function to avoid repetition. This would avoid the need to repeat the lock-enqueue-signal-unlock flow both in the `for` loop and when pushing in the poison pill.

- The following shows a quick sketch of a helper function that minimises the need to repeat the `Lock`, `enqueue`, `signal`, `unlock` pattern:

```
int safe_boss_enqueue(work_item *item) {
        pthread_mutex_lock(&mutex_workq);
        steque_enqueue(workq, item);
        pthread_cond_signal(&cond_workq_not_empty);
        pthread_mutex_unlock(&mutex_workq);
        return 0;
}
```

Pseudocode (gfclient_download worker)

The following pseudocode shows the flow involved in the worker thread implementation:

```
/* in worker thread */
while (1) {
    /* loop forever */
    Lock(work_queue_mutex);
    while (work_queue.is_empty) {
        Wait(cond_work_queue_not_empty, work_queue_mutex);
    }

    /* peek at front of queue */
    if (work_queue.peek == POISON_PILL) {
        Signal(cond_work_queue_not_empty);
        Unlock(work_queue_mutex);
        break;
    }
```

```
    /* otherwise pop and handle task */
    work_item = work_queue.pop();
    Unlock(work_queue_mutex);
    handle_work_item(work_item);
}
```

A few comments:

- The critical section in this worker thread implementation is less clear than that of the boss thread. If the worker sees a poison pill, then the mutex is released and the critical section ends there. Otherwise, the worker should proceed to pop the enqueued task **before** unlocking the mutex.
- The worker threads are subjected to `Wait`-ing on a condition variable, where the `work_queue` must not be empty before they are allowed to wake up. This `Wait` is of course, placed within a `while` loop that checks for the condition before allowing the thread to proceed.

  > In the duration between waking up and trying to acquire the mutex, it might be the case that another thread has popped the remaining item in the `work_queue`, resulting in an empty `work_queue` once more.
  >
  > Therefore, the `while` loop enforces this check -- if the condition is somehow not satisfied anymore, go back to waiting.

- If a worker encounters a task (represented above as a `work_item`), then it should promptly release the `work_queue_mutex` after popping the task from the queue.

  > After popping from the queue, access to the shared resource is no longer required for the worker to complete its task. The mutex can be freed without much concern at this point.

- The poison pill is never popped by the worker threads. They simply encounter the pill and terminate. As such, the **clean up** after all worker threads have terminated would be the **responsibility of the boss thread**.

gfserver_main & handler

gfserver_main

The `gfserver_main` implementation is essentially similar to that of Part 1, which shall not be repeated in this section. The bulk of actual multithreading code is implemented within `handler` instead.

Where multithreading is concerned, `gfserver_main` simply spawns/joins the worker threads and handles the declaration of the mutex and condition variable used.

Pseudocode (handler boss)

The following pseudocode shows the flow involved in the boss thread implementation:

```
/* in boss thread */
for (request in requests) {
    Lock(work_queue_mutex);
    /* begin critical section */
    work_queue.enqueue(request);
    Signal(cond_work_queue_not_empty);
    /* exit critical section */
    Unlock(work_queue_mutex);
}
```

Note then that this is a simpler version of the boss thread implementation within `gfclient_main`. Nothing of value can be added in description here.

Pseudocode (handler worker)

The worker task is slightly more involved, thus is split into two helper functions `gfs_send_wrapper` and `gfs_worker_task`. `gfs_worker_task` calls `gfs_send_wrapper`.

The following pseudocode shows the flow involved in the worker thread implementation:

```
/* in worker thread */
define gfs_worker_task() {
    while (1) {
        /* loop forever */
        Lock(work_queue_mutex);
        while (work_queue.is_empty) {
            Wait(cond_work_queue_not_empty, work_queue_mutex);
```

```
        }
        work_item = work_queue.pop();
        Unlock(work_queue_mutex);
        gfs_send_wrapper(work_item); /* call helper function */
    }
}

define gfs_send_wrapper(work_item) {
    file = work_item.file;
    file_len = file.length;
    bytes_sent = 0;
    while (bytes_sent < file_len) {
        buffer = file.read(file_len - bytes_sent);
        bytes_sent += gfs_send(buffer); /* call library gfs_send */
    }
}
```

A few comments:

- The worker should never terminate as the server should run indefinitely.
- The function `gfs_worker_task` in `handler` is very similar to that of the worker task in `gfclient_download`, albeit simpler. The actual work of sending files to a requesting client is handled by `gfs_send_wrapper` instead.

  > Simply put, `gfs_worker_task` only handles concurrency-related considerations.

- The function `gfs_send_wrapper` loops until it ensures that all bytes have been successfully sent via `gfs_send`. This is similar to the strategy used in the Warmup and Part 1 thus will not be rehashed.

Testing

1. cparaz3's Python test server and test client [6] were used in the first phase of testing. It does not matter that the test server and client are single-threaded, as the first goal was to ensure the correctness of requests and responses being sent or received.
2. cparaz3's Java test client was then used for testing the multithreaded server.

    - Importantly, the earlier iterations of `gfserver_main` and `handler` started to break under duress (i.e. when stress tested as per cparaz3's suggestion).
    - This led to the realisation that I was enqueing double pointers into the shared work queue instead of the pointer to the actual work item `struct` itself. The double pointer would occassionally be freed and pointed to `NULL` (perhaps due to some memory management by `gfserver` itself), which led to SegFaults.
3. The multithreaded server was then uploaded to Gradescope, which returned a failed test case stating that `fstat` should be used instead of `lseek`.
    - According to StackOverflow user Nitin [22], `fstat` is generally preferred due to its speed. `lseek` may incur time penalities (disk reading) if the file itself is not present in cache.
4. The multithreaded client `gfclient_download` was then tested against the working multithreaded server.
    - Testing with the default options yielded no issues, though imposing stress resulted in a **client that would never terminate**.
    - Race conditions due to a lack of proper mutex lock/unlocking were then identified.
    - More specifically, the boss thread was not acquiring a mutex before modifying the shared work queue with a poison pill. This presumably led to inconsistencies within the work queue, resulting in the poison pill never being properly enqueued. Worker threads are therefore left to run forever.

Code References

This section compiles code used and referenced in implementing `gfclient_download.c`, `gfserver_main.c` and `handler.c`.

1. The work queue used the provided `steque` implementation. The following methods were used; the `steque` was only ever treated as a classic FIFO queue:
    - `steque_enqueue`, `steque_front`, `steque_isempty`, `steque_pop`, `steque_init`, and `steque_destroy`.
2. The general workflow for initialising and setting relevant parameters for `gfserver_t` and `gfcrequest_t` were pulled from Part 1's provided code.
    - This would include the usage of `gfc_create()`, `gfserver_create()`, and relevant setter functions.

3. The code used to extract file length information from a file descriptor was adapted from an answer to the StackOverflow question *"How can I get a file's size in C?"* by users Greg Hewgill and cubuspl42 [23]. The relevant snippet is displayed here for ease of reference:

```
/* handler.c :: gfs_send_wrapper */
struct stat st;
fstat(file_descriptor, &st);
file_len = st.st_size;
```

Feedback

1. The comment decorating `gfs_sendheader` within `gfserver.h`: "This function should only be called from within a callback registered gfserver_set_handler" [sic] is confusing, as mentioned above.
    - A quick note: there is also a grammatical error. It should be "... from within a callback registered **with** gfserver_set_handler" instead of "... from within a callback registered gfserver_set_handler" ("with" omitted).
    - The exact same phrasing is correct for `gfs_send`, where "with" is not dropped erroneously. The comment for `gfs_send` reads: "... from within a callback registered with gfserver_set_handler".
    - A list of Piazza and Slack posts referencing (and being confused by) this particular comment is compiled in Supplementary Note A.
    - Suggested rectification: if the intention is to disallow `gfs_sendheader` to be called by the library user, then perhaps the phrasing should be altered to "A user of the `gfserver` library should only call this function from within a callback registered with `gfserver_set_handler`".
    - If the intention is to disallow us students from calling `gfs_sendheader` anywhere in our code, then this is an error in my understanding and this feedback can be ignored. I am still of the opinion that this particular motive does not make much sense.
2. The hidden requirement of reusable sockets should be made known and the solution publicly available to students.
    - In brief, the following line of code is required: `setsockopt(sockfd, SOL_SOCKET, (SO_REUSEADDR | SO_REUSEPORT), &(int){1}, sizeof(int)` for test cases on Gradescope to pass.
    - Based on my limited knowledge, this was never communicated anywhere. This led to numerous questions about the Gradescope error `Address already in use`.
    - I believe that this project in particular tests not the ability to fix the `Address already in use` issue, rather tests our ability to handle concurrency and its related issues, moving slowly upwards from socket programming.
    - A simple fix would be to pin such a note in Piazza at the start of the semester: "Gradescope testing requires that you enable reusable sockets.

You may do so with the following line of code `setsockopt(...)` after you have obtained the socket file descriptor."

- Even better, the project README should be amended to include this note.

3. The list of files to upload per segment is overly complicated within the project README.
    - The Piazza post regarding the exact files to upload per section alleviated this issue.
    - However, since a formatted table is already available in Piazza, I doubt that it will be too difficult to include the same table within the README as well.
    - This is not a major issue and is more annoying than impeding.

References

[1] Gonzalo, Jonathan Feinberg, and unwind, “What is the difference between read() and RECV() , and between send() and write()?,” Stack Overflow, https://stackoverflow.com/questions/1790750/what-is-the-difference-between-read-and-recv-and-between-send-and-write (accessed Sep. 11, 2023).

[2] Write(2) - linux manual page, https://man7.org/linux/man-pages/man2/write.2.html (accessed Sep. 11, 2023).

[3] nycdavid, “Notes for GIOS Project 1,” *Gist*. https://gist.github.com/nycdavid/748da176af8d4e1330d6dae3c476a925 (accessed Sep. 11, 2023).

[4] “Beej’s Guide to Network Programming,” *beej.us*. https://beej.us/guide/bgnet/html/split/index.html (accessed Sep. 11, 2023).

[5] “Lorem Ipsum - All the facts - Lipsum generator,” *Lipsum.com*, 2019. https://www.lipsum.com/

[6] cparaz3, "6200-tools" *GitHub*. https://github.gatech.edu/cparaz3/6200-tools (accessed Sep. 11, 2023).

[7] “Splitting a string using strtok() in C,” *Educative: Interactive Courses for Software Developers*. https://www.educative.io/answers/splitting-a-string-using-strtok-in-c (accessed Sep. 11, 2023).

[8] “Coding Games and Programming Challenges to Code Better,” *CodinGame*. https://www.codingame.com/playgrounds/14213/how-to-play-with-strings-in-c/string-split (accessed Sep. 11, 2023).

[9] “Hex Editor - Visual Studio Marketplace,” *marketplace.visualstudio.com*. https://marketplace.visualstudio.com/items?itemName=ms-vscode.hexeditor

[10] “Regular expressions in C,” *GeeksforGeeks*, May 09, 2020. https://www.geeksforgeeks.org/regular-expressions-in-c/ (accessed Sep. 11, 2023).

[11] Dervin Thunk and Mohammed H, “Regular expressions in C: examples?,” *Stack Overflow*. https://stackoverflow.com/questions/1085083/regular-expressions-in-c-

examples (accessed Sep. 11, 2023).

[12] “RegExr: Learn, Build, & Test RegEx,” *RegExr*, 2017. https://regexr.com/

[13] “C library function - strstr() - Tutorialspoint,” *www.tutorialspoint.com*. https://www.tutorialspoint.com/c_standard_library/c_function_strstr.htm

[14] “C strcat() - C Standard Library,” *www.programiz.com*. https://www.programiz.com/c-programming/library-function/string.h/strcat

[15] “Different ways to copy a string in C/C++,” *GeeksforGeeks*, Dec. 16, 2020. https://www.geeksforgeeks.org/different-ways-to-copy-a-string-in-c-c/

[16] “C library function - sscanf() - Tutorialspoint,” *www.tutorialspoint.com*. https://www.tutorialspoint.com/c_standard_library/c_function_sscanf.htm

[17] “C strcmp() - C Standard Library,” *www.programiz.com*. https://www.programiz.com/c-programming/library-function/string.h/strcmp

[18] it-person and Marco Bonelli, “strtok c multiple chars as one delimiter,” *Stack Overflow*. https://stackoverflow.com/questions/59770865/strtok-c-multiple-chars-as-one-delimiter (accessed Sep. 11, 2023).

[19] Spikatrix, “Implementing `strtok` whose delimiter has more than one character,” *Stack Overflow*. https://stackoverflow.com/questions/29847915/implementing-strtok-whose-delimiter-has-more-than-one-character (accessed Sep. 11, 2023).

[20] Ioan, “Few gotchas for MT Project 1 client (part 2),” *www.piazza.com*. https://piazza.com/class/llfb9mwtjk75hv/post/42 (accessed Sep. 11, 2023).

[21] “Reading 20, Part 2: Message Passing with Threads & Queues,” *web.mit.edu*. https://web.mit.edu/6.005/www/fa14/classes/20-queues-locks/message-passing/ (accessed Sep. 11, 2023).

[22] Nitin, “Best practices for determining binary regular file size in POSIX compliant C program,” *Stack Overflow*. https://stackoverflow.com/questions/48308963/best-practices-for-determining-binary-regular-file-size-in-posix-compliant-c-pro/48310046#48310046 (accessed Sep. 11, 2023).

[23] Nino and Matthias Braun, “How can I get a file’s size in C?,” *Stack Overflow*. https://stackoverflow.com/questions/238603/how-can-i-get-a-files-size-in-c

Supplementary Notes

Supplementary A: List of student posts regarding `gfs_sendheader` confusion as of Sep 12.

Any posts after Sep 12 are not included. This section of the README is finalised as of Sep 12.

slack user ayong30, myself; Aug 31
slack user sfoo6; Sep 2
slack user Twan; Sep 9
slack user tkhan66; Sep 10
slack user ang75; Sep 11
slack user aditya; Sep 12
piazza user Zhongyuan Chen; Sep 8
piazza user Bharath Prabhu; Sep 8
piazza user Anibal Castineyra Calcano; Sep 9
piazza user Moiz Hussain; Sep 11