README_project3

Part 1

1.1 Design and Flow

1.1.1 Overview

Part 1 of Project 3 mainly dealt with using `curl` functions. The general idea behind the proxy submitted is:

1. Obtain file path requested by the client.
2. Request headers from webserver *without* downloading the file.
3. Handle the webserver response (e.g., *FILE_NOT_FOUND*) or the expected file size if it exists.
4. Send the file to the client.

It **is** possible to first download the file locally, then send data to the client with `gfs_send` (some other classmates opted for this method). However, this was not ideal, since there will be overheads associated with writing a file that will serve no real purpose once the client receives the relevant data. This main concern informed the design choices behind the aforementioned steps (2) and (3).

Primarily, the appropriate header and the file (if it exists) must be sent in one "continuous" action.

In order to generate the response header for `gfs_sendheader`, as per (2), the webserver response headers were first requested. This was performed by setting `curl` to ignore the response body.

```
curl_easy_setopt(curl_handle, CURLOPT_HEADER, 1);
curl_easy_setopt(curl_handle, CURLOPT_NOBODY, 1);
curl_easy_perform(curl_handle); // execute request
```

The relevant information was extracted by calling `curl_easy_getinfo` with the relevant flags (e.g., `CURLINFO_CONTENT_LENGTH_DOWNLOAD_T`);

Next, if the file is present and everything is in order, a *second* `curl` request is made using the same `curl_handle`, this time for the content data itself. `curl` provides an option to specify the callback function associated with handling incoming data. For example, if saving to disk was the intention, this callback function should be able to save incoming data to disk (obviously).

Therefore, this callback function can be exploited to immediately invoke `gfs_send` -- this might be roughly akin to simply "passing the data on". As a final note, the `curl_handle` is explicitly reset to default values before this second request.

```
curl_easy_reset(curl_handle);
// other options to be set
```

```
curl_easy_setopt(curl_handle, CURLOPT_WRITEFUNCTION, custom_writer_func);
curl_easy_perform(curl_handle); // execute request
```

1.2 Testing

Testing for this project involved similar workflows to that of Project 1. Namely:

1. Image downloads should be at least verified with a cursory glance -- the image must be open-able and contain no visible artefacts.
2. Files were directly obtained from the test webserver; it's as straightforward as pasting the full URL into a browser search box. These files were manually downloaded and their checksums verified against the files downloaded with `gfclient_download` via `webproxy`.

1.3 Code References

The libcurl "Easy" API documentation [1] was heavily referenced in this portion of the project. The following functions were referenced:

- `curl_easy_init()`
- `curl_easy_setopt()`
- `curl_easy_getinfo()`
- `curl_easy_reset()`
- `curl_easy_cleanup()`
- `curl_global_cleanup()`

The solution to obtaining a file size prior to actually downloading was gleaned from multiple StackOverflow questions and answers [2, 3, 4]. Overall, the idea was to parse the response header for relevant information.

Part 2

2.1 Design and Flow

2.1.1 Overview

A few considerations, before starting:

1. There are two major components in this part of the project, the (i) webproxy process and the (ii) cache process.
2. However, communication between the two processes can be divided into **three sub-components**, which will be covered shortly below. Briefly, they are the (i) queue of shared memory segments owned by the webproxy, (ii) a command channel owned by the cache, and (iii) the access of shared memory by both the webproxy and cache. The project was constructed based on these three sub-components, and will be covered in this README as such.
3. Both processes are allowed to be multithreaded.

4. Multiple webproxy processes are allowed to communicate with a single cache.

The discussion regarding implementation design will more closely follow the three sub-components listed above, instead of examining each modified `.c` file in isolation (as was the case in Project 1).

The overall flow of an incoming client request, and subsequent cache request, is depicted in the figure below:

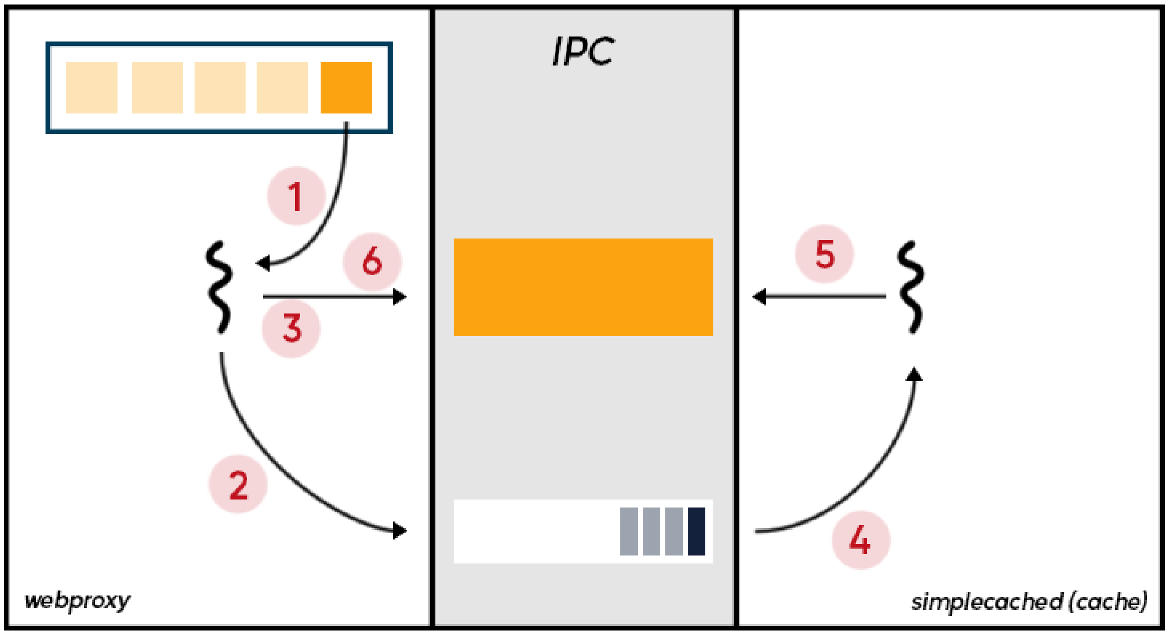

Figure 1: Overall flow of client request handling with webproxy and cache. Interprocess Communication (IPC) mechanisms used are shared memory and message queue.

1. A client request is received by the webproxy. A worker thread (webproxy) "claims" a shared memory segment for itself.
2. The worker thread (webproxy) communicates the request to the cache, alongside the shared memory segment identifier.
3. The worker thread (webproxy) waits at the shared memory segment for the cache to respond.
4. A worker thread (cache) notes the request for cache access and accesses the same shared memory via its identifier.
5. The worker thread (cache) writes data to the shared memory and informs the worker thread (webproxy) that data has been written. If not all data is yet transferred, this information is conveyed as well.
6. The worker thread (webproxy) accesses data in the shared memory and sends it to the client requester.

2.1.2 Shared Memory Queue; Webproxy

Introduction

The first component built was the queue of shared memory segments owned by the webproxy. Again, starting with the necessary considerations:

1. The shared memory segments must be "claimed" by a webproxy worker thread -- the same segment cannot be used by another intra-process thread other than the corresponding (inter-process) cache worker thread.
2. The shared memory segments must be available for use and reuse.
3. The shared memory segments must be initialised per webproxy process -- and uniquely identified in such as manner as well.

Consideration (1) dictates the use of some form of synchronisation construct, though having mutexes for *each individual* shared memory segment was deemed too complicated to handle. As such, the `steque` implementation provided lent itself very nicely to this use case:

- A mutex can be used to control access to the `steque` itself. The `steque` was used as a classic FIFO queue.
- Once a shared memory segment is popped from the queue, the thread effectively "owns" said shared memory.
- Naturally, once the shared memory segment is no longer needed, it is re-entered into the queue for use by other requests. This satisfies consideration (2).
- Synchronisation among threads within a process was already covered in great detail in the README for Project 1 Part 2, and will not be rehashed here for brevity.

The handling of the shared memory queue was mostly abstracted away in a separate "library" file `shm_channel`. This involved implementing helper functions to enqueue and dequeue a shared memory segment, for example.

Shared Memory Size

The size of shared memory was to be dynamically decided (i.e., a user could enter the desired size, `segsisze` via command line arguments). However, this implementation uses a slightly larger size instead. The figure below illustrates this notion:

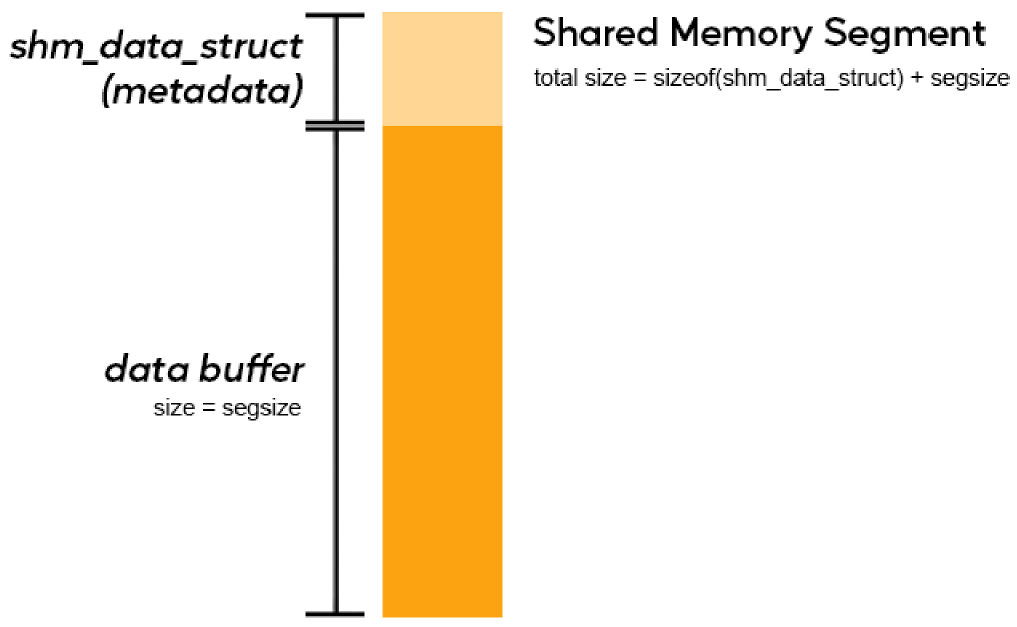

Figure 2: Arbitrary depiction of shared memory segments. The shared memory segment is divided into two main components: metadata and a large data buffer for file transfer.

Basically, the user-given size will be used as the size of the data buffer, while some additional space is used to contain certain information regarding the shared memory (e.g., synchronisation constructs, total file size, etc.). This was termed `shm_data_struct`, though one might simply consider this a `struct` used for containing metadata.

The virtual address of the data buffer was accessed with pointer arithmetic. In essence, the pointer returned by `mmap` (which should point to the start of the process' virtual address that maps to shared memory) was incremented by the size of the metadata `struct`. Using Figure 2 as a reference, this would simply be "moving" the pointer towards the start of the darker yellow (data buffer) section.

Shared Memory Identifier

Shared memory is to be identified with a string, prepended with a slash (`/`) as specified by the documentation for `shm_open()` [5]. Since a cache must allow requests from **multiple** webproxies (consideration (3)), there is a need to distinguish the different webproxy processes as well. In order to handle this, the shared memory identifier also encodes the process id, obtained by calling `getpid()`. Finally, within a process, each shared memory segment is arbitrarily identified with a 1-based numbering system.

A sample shared memory identifier would therefore be:

```
"/shmp1234i1" /* /shmp<pid>i<arbitrary_num> */
```

Shared Memory Creation/Cleanup

Shared memory created was intended to be uniquely identified. As such, the `O_EXCL` flag was used during creation. However, this resulted in issues if the program crashed prior to performing proper IPC cleanup. Subsequent attempts to create shared memory with the same identifier resulted in an error -- as expected.

As such, the `EEXIST` error is caught. From the man page [5]:

> `EEXIST`: Both `O_CREAT` and `O_EXCL` were specified to `shm_open()` and the shared memory object specified by name already exists.

If `EEXIST` is encountered, the existing shared memory will be unlinked, and shared memory creation will be attempted until it succeeds (or a different, uncaught error is encountered).

Shared memory is cleaned up by simply calling `shm_unlink` on all shared memory segments initially created.

2.1.3 Message Queue; Cache

Introduction

There was a need for some message passing ability between the webproxy and the cache. Initially, sockets were considered due to familiarity courtesy of Project 1. However, sockets for message passing would require synchronisation, as well as the ability to handle messages that may be broken up in a stream of bytes.

There were discussions on Slack regarding message passing alternatives, and most seemed to settle on message queues as a viable option. In this implementation, a singular message queue was eventually used due to the following reasons:

1. It's a new form of inter-process communication; possibly well worth the experience learning.
2. Message queues do not require inter-process synchronisation.
3. Messages are passed as a singular, whole unit. In other words, there is no need to continually parse an incoming message until a whole "message" is received (as would be the case for sockets).

In fact, the usage of a one-way message queue effectively allowed webproxy worker threads to "fire-and-forget". A message is simply placed within the message queue, and the webproxy worker thread can proceed to wait at the appropriate shared memory segment for the cache to respond.

- Of course, this relies heavily on the assumption that the cache requests are always handled in some manner.
- For example, if the cache request is malformed (no shared memory segment identifier present), then there is no way for the cache to notify the webproxy of such an issue.
- This project does state upfront that the "[communication mechanism] does not have to be robust to misbehaving clients", which could be construed as "the cache requests are always valid".
- This assumption is therefore valid (at least, in this scenario).

Message Queue Creation/Cleanup

Message queue creation is rather straightforward. Again, the flags `O_CREAT` and `O_EXCL` were used, which mean that message queue creation would fail *if* a similarly named message queue already exists. As such, the message queue name used, `/cmsgq`, must be unlinked prior to creation.

On the webproxy end, there is a need to connect to the message queue. Of course, there is no guarantee regarding order in which the proxy or cache is started. Therefore, the webproxy will continually attempt to connect to `/cmsgq`, sleeping for 1 second in between each retry.

`mq_unlink` is called if the process terminates to clean up the message queue.

Message Sent

The webproxy worker threads will individually try to open a connection to the message queue and simply enqueue their message.

> I believe that the better design choice would be to have the boss thread (webproxy) open a singular connection instead. Worker threads should then use that connection to enqueue their messages. However, I do think that in terms of actual effect, there should be not too much of a difference, since a message queue is thread-safe anyway. In terms of code design, the current implementation is a rather poor choice.

Message Received

The cache implements a boss/worker threading pattern. The boss handles the creation of the message queue, and eventually waits for incoming messages to parse. An incoming message will be enqueued as a worker task within a FIFO queue (again, using the provided `steque` implementation).

The following information are required from the webproxy.

- The process id of the webproxy
- The shared memory segment id. Both the process id and the shared memory segment id will be used to identify the correct shared memory segment.
- The requested file path
- The size of the shared memory segment.

> As a quick aside, the shared memory segment identifier string should just be sent over in the message. In this implementation, the cache would have to construct the shared memory identifier (`/shmp<pid>i<i>`) using the provided process id and shared memory segment id. It should not have been its responsibility.

2.1.4 Shared Memory Access; Cache & Webproxy

Synchronisation (IPC)

Note that this section on synchronisation concerns two separate processes, and should not be confused with synchronisation between threads of a single process -- that is material for Project 1.

A mutex and a condition variable was used to control access to each individual shared memory segment (i.e., each segment contained their individual synchronisation constructs within the metadata section).

- I understand that semaphores are traditionally used, at least, where shared memory examples are concerned.
- However, this decision was made in accordance with the recommended lecture material (P3L3), where `pthread` mutexes and condition variables were noted to be available for use in IPC as well.
- Since semaphores were not yet covered, and with much needed background already established with Project 1, I opted for mutexes and condition variables instead.
- Given the overall design, a binary semaphore would have been used anyway, so mutex usage poses no effective difference (at least, to my knowledge).

The webproxy and cache can be loosely categorised into "reader" and "writer" respectively.

- The state of shared memory -- being either ready for reading or writing -- is maintained by a single integer flag, `ready_code`. The integer flag is set to `1` if the shared memory is ready for writing (`WR_READY`) or `0` if the shared memory is ready for reading (`RD_READY`).
- There is only one mutex used. This owner of the mutex is allowed to access the shared memory region.
- There is only one condition variable used. Both the webproxy (reader) and cache (writer) will wait at the condition variable until they are signalled by the other.
- Of course, this assumes that only two parties are trying to access the same shared memory at any given point in time. This assumption is held true due to other synchronisation mechanisms employed within the webproxy and cache itself.

The following pseudocode shows how both parties wait to access shared memory for reading or writing.

```
/* Enter/Exit Critical Section Pseudocode */

// Webproxy, READER
enter_critical_section_reader() {
  Lock(mutex);
  while (ready_flag != RD_READY) {
    Wait(cond_var, mutex);
  }
}

exit_critical_section_reader() {
  ready_flag = WR_READY; // handover to writer
  Signal(cond_var);
  Unlock(mutex);
}

// Cache, WRITER
enter_critical_section_writer() {
  Lock(mutex);
  while (ready_flag != WR_READY) {
    Wait(cond_var, mutex);
  }
}

exit_critical_section_writer() {
  ready_flag = RD_READY; // handover to reader
  Signal(cond_var);
  Unlock(mutex);
}
```

Of course, due to the similarities in both reader and writer enter/exit critical section, general purpose functions were implemented to avoid repetition in code.

```
// taken from shm_channel.h
void shm_wait_and_acquire(shm_data_struct * shm_data, int wait_on);
void shm_release_and_signal(shm_data_struct * shm_data, int signal_to);
```

- Briefly, the reader or writer simply specifies an integer flag to `wait_on` or `signal_to`.
- For example, if a reader needs to access shared memory, it should `wait_on` the `RD_READY` flag. On the other hand, if a reader intends to release shared memory for the writer to access, it should `signal_to` the condition `WR_READY`.

Cache Request Metadata

The shared memory also maintains certain information to inform the webproxy (reader) about the status of a cache request:

- `status_code` is set to one of the available `gfserver` status codes. For example, this will be set to `GF_FILE_NOT_FOUND` if the cache does not contain the file request.
- `file_len` informs the webproxy of the total file size to expect. The webproxy should then continue to read data from shared memory until the expected `file_len` is received.
- `curr_len` informs the webproxy of the size of the current data in shared memory to read. This should usually correspond to the user-defined shared memory size, `segsize`.

Reading & Writing

The access of shared memory should *always* occur in the following order, where the respective reader (webproxy) and writer (cache) are concerned.

1. The webproxy waits at shared memory, since the `ready_code` is initialised to `WR_READY` (the writer gets first access).
2. The cache updates the shared memory to contain information regarding the cache request. If the file exists, the data buffer within the shared memory is loaded with as much data as `segsize` can contain.
3. If the whole file is not yet transmitted, the cache will still have to relinquish access to the webproxy to parse data currently shared.
4. The webproxy first notes the `status_code` of the cache request made and short-circuits if anything but a `GF_OK` is observed. If not all data has been received, it will hand access back to the cache after the currently shared data is conveyed to the client.
5. The back-and-forth between the webproxy and the cache continues until all data is transmitted, or if an error occurs midway (in which case the `status_code` or `curr_len` will be updated appropriately).

A minor note: `pread` was used instead of the `read` call provided within the starter example code. This is in line with the implementation used in Project 1 Part 2, where multithreading is concerned.

- `pread` removes any ambiguity regarding the position of the file being read, since each thread keeps track of its own offset.

The cache unmaps the shared memory segment from its own virtual address space after the cache request has been serviced.

- This is performed as the cache should not have to be bogged down by multiple possible webproxies.
- If a webproxy wants to reuse its shared memory segment, it can simply be mapped back into the cache's virtual address space once more.
- The alternative would be to have multiple mappings -- that may or may not even be used more than once -- that persist within the cache until the cache is terminated.

Code Design Comments

As a final note, access to shared memory should have been completely abstracted away into the `shm_channel` library. The cache breaks abstraction and directly interacts with the details of shared memory multiple times, the most egregious offence would be the piecing together of shared memory identifiers.

If I were to rewrite some parts (not that I intend to), the following actions should be hidden under `shm_channel`:

- `mmap` shared memory file descriptor to process virtual address space (cache directly calls `mmap`)
- `shm_open` with the appropriate shared memory identifier.

2.2 Testing

Part 2 afforded more flexibility in testing, since files can be created locally (in contrast to Part 1, where files *must* be downloaded from some server.)

Again, testing followed the gradual implementation of the three sub-components, which will be listed here for the reader's benefit:

- The queue of shared memory segments owned by the webproxy
- The command channel owned by the cache
- Access of shared memory by both the webproxy and cache.

2.2.1 Shared Memory Queue; Webproxy Testing

This phase was mainly concerned with safely interacting with the queue of shared memory segments. In other words, the correctness of shared memory was not a priority. The implementation of shared memory itself was stubbed with some arbitrary construct.

The webproxy was first tested in its ability to create the **correct amount** of shared memory segments.

Next, each client request must see the webproxy workers "claiming" a queue item for themselves.

- A queue item (i.e. shared memory segment, eventually) should not be interacted with by two webproxy worker threads.
- The worker threads were set to block indefinitely if they acquired a queue item. This allowed me to check that no errant worker threads were accessing queue items if the queue is already exhausted.

Finally, once the webproxy workers have completed their task, they should re-queue the shared memory segment for further use.

- This was performed quite simply by enforcing a single queue item (single shared memory segment).
- If all client requests can be "serviced", then dequeuing and subsequent enqueuing must have worked.

2.2.2 Message Queue; Cache Testing

This component required that messages from the webproxy are successfully communicated over the the cache.

- Simple strings were sent over as an initial proof of concept.
- Finally, a properly formatted cache request was sent over, and the cache was checked for successful receipt. The cache request included necessary details such as shared memory segment identifier, file path request, etc.

The webproxy was also tested for its ability to sleep and retry connection to the cache's message queue.

The cache was also tested for its ability to recreate a message queue in the event that it crashed without proper IPC clean-up. This was simply performed by calling `assert(1==0)` at any arbitrary point in the cache's code. As mentioned above, this is necessary due to the `O_CREAT` and `O_EXCL` flags used for message queue creation.

2.2.3 Shared Memory Access; Cache & Webproxy Testing

Reading & Writing (Part 1)

The ability to read and write from shared memory was first tested. Admittedly, proper synchronisation was not set up at this point.

- However, the likelihood of race conditions were slightly reduced by allowing the reader to sleep for a significant amount of time before attempting a read.
- Only integers are communicated via shared memory as well, which would hopefully be much faster than strings.
- In essence, the hope is that the writer completes its simple task before the reader attempts to view shared memory.

Synchronisation

Both the webproxy and cache worker threads were set to block indefinitely (taking turns, of course), to check that only one of the two can access shared memory at any given point in time. This was rather straightforward.

Next, the order in which the webproxy and cache worker threads access shared memory (as mentioned above) was tested.

- This phase allowed both the webproxy and cache worker threads to read and write to shared memory.
- The two take turns changing data within shared memory (again, starting with simple integers), and the change should follow the pattern delineated above.

Reading & Writing (Part 2)

A simple string was then transmitted over shared memory, via a `char *` array whose size was statically determined at compile time.

Moving forward, the ability to dynamically determine the size of shared memory was implemented and tested. This was easily the most challenging part of the project to figure out.

- Usually, the words "dynamic", "allocation", and "memory" would suggest the use of `malloc` or its counterparts.
- However, `malloc` did not seem to work for shared memory at all. While one process (e.g., cache) was able to write and read using the pointer returned by `malloc`, the same pointer was essentially useless when conveyed to the other process.

- I soon learned that `malloc` returns a pointer that points to some virtual address within a process' address space -- this basically would mean that the pointer is "private", and the same virtual address (e.g., `0x123`) would not correspond to the same item in another process [6, 7, 8, 9]
- This meant that the data buffer within shared memory (anything that is not metadata, basically) was to be accessed with pointer arithmetic.
- Once this concept was solidified, a simple `HELLO WORLD\n` string was successfully transmitted.

2.3.4 Everything Everywhere All at Once

Once the individual moving parts were in place, there was a need to test how everything integrated with each other. Testing here was performed by using client requests, and the entire chain of events: webproxy -> cache -> webproxy -> client was observed.

Firstly, simple client requests were made with small text files. Shared memory size was set as an arbitrarily large number.

- This ensures that all file content can be transferred in one shared memory segment.
- This therefore tests for successful interaction between all three parties, client, webproxy, and cache.
- All files were verified first by a quick glance at their contents and then with a simple checksum. This is true for all subsequent tests as well.

Next, shared memory size was set as the lowest possible value. In this case, `segsize = 824`. Recall that `segsize` refers to the data buffer size, and does not include space required for storing shared memory metadata.

- Larger files were used for transfer here.
- This ensures that the cache and webproxy can continually take turns writing and reading to/from shared memory, especially when the file-to-transfer is larger than the available shared memory space.

A few more other cases were tested -- the workflow was essentially the same. Make client requests and then check that files are valid. The cases are listed here:

Webproxy

- Single-threaded webproxy with multiple shared memory segments
- Multi-threaded webproxy with one shared memory segment
- Multiple webproxy processes, each are multi-threaded and use multiple shared memory segments.

Cache & Webproxy

- Single-threaded cache with single-threaded webproxy
- Single-threaded cache with multi-threaded webproxy
- Cache started before webproxy

- Webproxy started before cache

2.3 Code References

The code used for interacting with the provided `steque` implementation is essentially similar to my submission for Project 1 Part 2. The `steque` is used both in handling shared memory segments on the webproxy-end, as well as for enqueuing cache requests on the cache-end.

The spawning of worker threads within `simplecached.c` is also copied exactly from Project 1 Part 2.

Some of the syscalls used for dealing with IPC were directly pulled from a slide deck linked in the Project 3 Piazza post [10]. For message queues specifically, a tutorial website by James Madison University was followed closely [11]. A list of such syscalls are copied here for ease of reference:

```
// Message Queue
mq_open("/mqname", O_CREAT | O_EXCL, 0600, NULL);

mq_unlink("/mqname");

struct mq_attr attr;
mq_getattr(msgqueue, &attr);
char msg_buffer[attr.mq_msgsize];
mq_receive(msgqueue, msg_buffer, attr.mq_msgsize, NULL);

mq_send(msgqueue, message, MAX_CACHE_REQUEST_LEN, 0);

// Shared Memory
shm_open(shm_name, O_CREAT | O_EXCL | O_RDWR, 0600);

mmap(NULL, (get_struct_metadata_size() + segsize),
    PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0));
```

- Most of the flags/permissions used were directly copied.
- Of course, the man pages for all syscalls used were consulted heavily as well.

Feedback

This section is purely personal opinions, and should naturally be taken with a grain of salt. A student only knows so much regarding how a course is conducted.

I am of the opinion that passing a variable server name to be concatenated with the requested file path could have easily been implemented for us.

- Specifically, I am referring to Part 1, where passing a server name to the `handle_with_curl` callback was necessary by editing the line with `gfserver_setopt`.
- The change is extremely minor -- replace the given dummy argument `"arg"` with the variable `server`, which will contain the server name given via command line arguments.

- I believe the intention of Part 1 was to get students used to the notion of having a middleman (webproxy) communicate with the server. The act of changing `"arg"` to `server` does not serve this learning objective in any meaningful way.
- In fact, two sentences were required to explain how the server name should not be hardcoded within the project README.

  > **Note: the grading server will not use the same URL.** Do not encode this URL into your code or it should fail in the auto-grader.

- By implementing this for the student, the above two sentences are not necessary.
- All the student has to focus on is then the understanding of a webproxy, and how best to structure `curl` calls to complete the task at hand.
- In fact, the act of creating a full server file path should just be implemented for the student.
- A suggestion on how to modify the default files provided:

```c
// within server/webproxy.c

/**
* Pass the server name to the GFS_WORKER_FUNC callback.
* This is to be appended with the requested file path.
*/
gfserver_setopt(&gfs, GFS_WORKER_ARG, i, server);

// within server/handle_with_curl.c

/**
* Append requested path to server (provided as arg).
* For example, if the path requested is foobar.txt, the full path
* should be
* "https://raw.githubusercontent.com/gt-cs6200/image_data/foobar.txt"
* These lines already handle that for you.
*/
char full_path[BUFSIZE];
strncpy(full_path, char(*)arg, BUFSIZE);
strncat(full_path, path, BUFSIZE);
```

This project was significantly less messy compared to Project 1. All the necessary information was contained within the project README, in stark contrast to Project 1, where information was scattered throughout `.h` files or `.c` files.

The same comment for Project 1 applies here as well: it would be nice to have a table of files to submit. The table of files to submit already exists on Piazza. It wouldn't hurt to include one within the README for Project 3 as well.

References

[1] “The LIBCURL API,” libcurl - API, https://curl.se/libcurl/c/ (accessed Oct. 16, 2023).

[2] rturrado and Remy Lebeau, “How to get the length of a file without downloading the file in a curl binary get request,” Stack Overflow, https://stackoverflow.com/questions/7677285/how-to-get-the-length-of-a-file-without-downloading-the-file-in-a-curl-binary-ge (accessed Oct. 16, 2023).

[3] Snazzer and Mukul Gupta, “Using libcurl to check if a file exists on a SFTP site,” Stack Overflow, https://stackoverflow.com/questions/2477936/using-libcurl-to-check-if-a-file-exists-on-a-sftp-site (accessed Oct. 16, 2023).

[4] Anthony O and Daniel Stenberg, “Is there a way to get the content-length from an HTTP header using libcurl >=7.3.x?,” Stack Overflow, https://stackoverflow.com/questions/58388748/is-there-a-way-to-get-the-content-length-from-an-http-header-using-libcurl-7-3 (accessed Oct. 16, 2023).

[5] shm_open, https://pubs.opengroup.org/onlinepubs/007904875/functions/shm_open.html (accessed Oct. 16, 2023).

[6] ChiP, Rerito, and askmish, “C - shared memory - dynamic array inside shared struct,” Stack Overflow, https://stackoverflow.com/questions/14558443/c-shared-memory-dynamic-array-inside-shared-struct (accessed Oct. 16, 2023).

[7] golden_boy615 and JohnGraham, “How to allocate dynamic memory to shared memory,” LinuxQuestions.org, https://www.linuxquestions.org/questions/programming-9/how-to-allocate-dynamic-memory-to-shared-memory-918953/ (accessed Oct. 16, 2023).

[8] Roundstic and Remy Lebeau, “How to initiate a dynamic array in struct with shared memory C - language,” Stack Overflow, https://stackoverflow.com/questions/65100580/how-to-initiate-a-dynamic-array-in-struct-with-shared-memory-c-language (accessed Oct. 16, 2023).

[9] Shared memory and malloc! - cboard.cprogramming.com, https://cboard.cprogramming.com/c-programming/144829-shared-memory-malloc-printable-thread.html (accessed Oct. 16, 2023).

[10] An introduction to linux IPC - man7.org, https://www.man7.org/conf/lca2013/IPC_Overview-LCA-2013-printable.pdf (accessed Oct. 16, 2023).

[11] “3.6. message passing with message queues¶,” 3.6. Message Passing With Message Queues - Computer Systems Fundamentals, https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/MQueues.html (accessed Oct. 16, 2023).