Multi-threaded Online File Request System

Author: Yuan Yang (yyang998)

This project is developed under Ubuntu Linux 20.04 environment with Vagrant and VirtualBox.

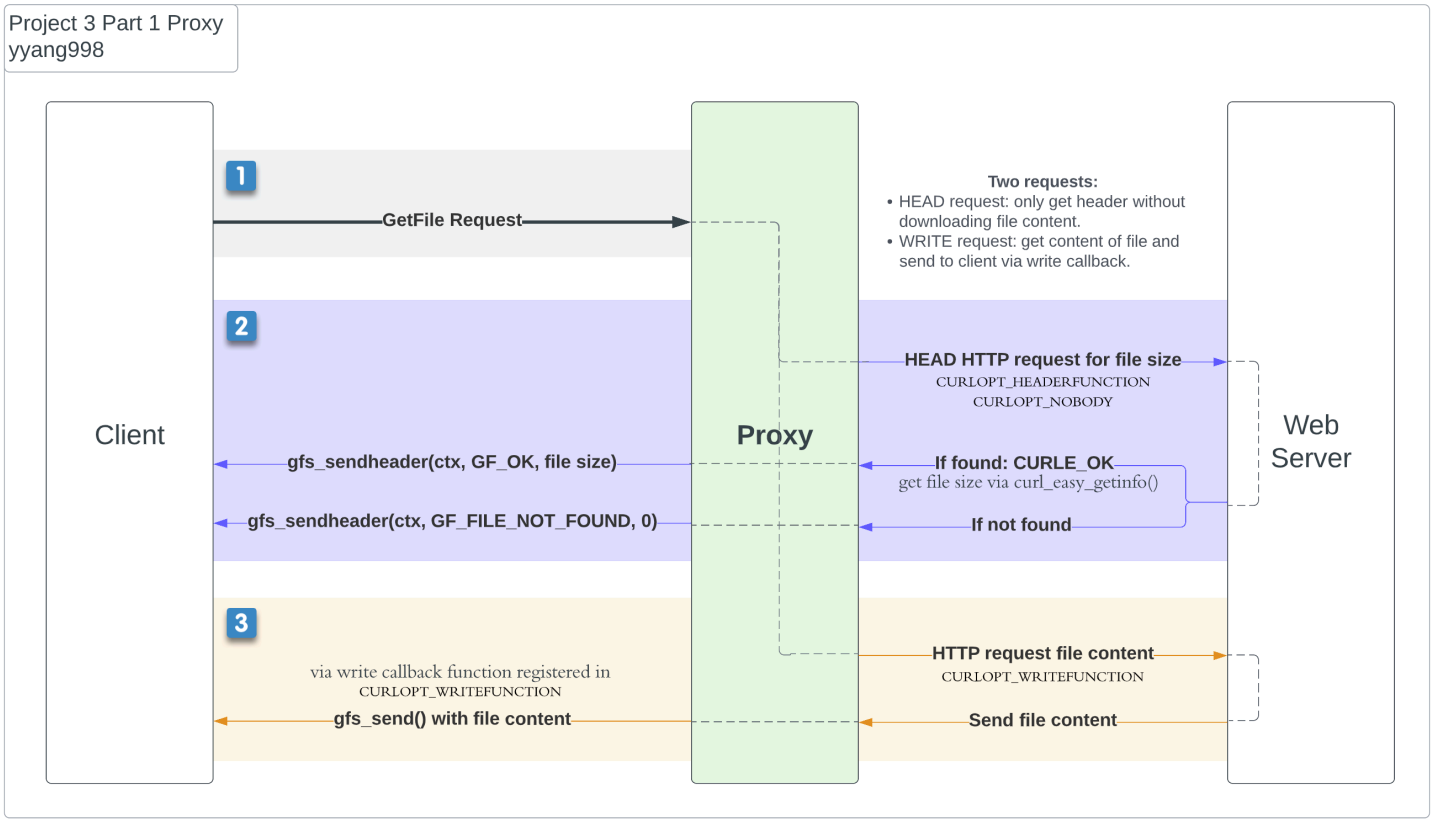
1. Part 1 - Proxy Server

Design

The goal of this part of the project is to implement a Proxy process to accept GetFile request from Client, then send HTTP request to Web Server to request the file. After getting file info and content back from Web Server, the Proxy sends the file back to Client. The main function of the multithreaded

Proxy is to communicate with Web Server via HTTP requests, and this part is implemented using libcurl's "easy" interface.

The figure attached below is the flow diagram of my design for this part of project.



As shown in the figure, I classified the whole flow into 3 major steps (blue labels):

1. Client sends GetFile request to Proxy;
2. Proxy requests file size from Web Server, and sends header back to Client;
3. Proxy requests file content from Web Server, and sends file content to Client via write callback function.

The major tasks of the project are the step 2 and 3, which are implemented in handle_with_curl.c. The handle_with_curl() function is the worker function that should be completed by each worker thread in Proxy process.

In order to save memory space (avoid using intermediate buffer) and save time, proxy should send two HTTP requests to Web Server: a HEAD request is sent in step 2 to check if the requested file can be found on Web Server and to get file size if file exists, another request is sent in step 3 to get file content. The third step is not mandatory for every GetFile request: if file is not found in Web Server, the third step should be omitted. *The detailed explanation of why I designed the Proxy to send two requests is elaborated in next section Choices and Code Implementation.*

The design details of the two requests sent by proxy in step 2 and step 3 are introduced below:

- **First request made by Proxy: HEAD HTTP request (labeled as step 2 in the diagram)**
  i. Based on GetFile request from Client in step 1, Proxy processes the requested file info, initializes the curl handle and does the setup of the libcurl's interface.
    - In order to make a HEAD HTTP request to Web Server without actually downloading/writing the file content, several libcurl options and callbacks need to be setup. The main setup is to register a header callback. The specific setups are listed in Choices and Code Implementation.

  ii. Proxy performs the HEAD HTTP request using `curl_easy_perform()` and gets file size if the file is found.
    - If the request succeeds (file found): gets file size via internal info CURLINFO_CONTENT_LENGTH_DOWNLOAD_T using `curl_easy_getinfo()` .
    - If the request fails (file not found), go to next step.

  iii. Proxy sends header to Client:
    - If file is found in last step: send header with GF_OK and the file size.
    - If file is not found in last step: send header with GF_FILE_NOT_FOUND.

  iv. Do curl cleanup.

- **Second request made by Proxy: WRITE HTTP request (labeled as step 3 in the diagram)**
  i. Proxy initializes another curl handle and does the setup: The main setup is to register a write callback to send the file content to Client as soon as Proxy receives content from Web Server. The specific setups are listed in Choices and Code Implementation.

  ii. Proxy performs the HTTP request using `curl_easy_perform()` and gets file content.

  iii. Once Proxy receives a chunk of file content, the callback function registered with curl handle can send the content directly to Client.

  iv. Do curl cleanup.

Choices and Code Implementations

- **How should Proxy request file size and content from Web Proxy.** There are two options I can come up with when I was designing, and option 2 is mechanism used in this project.
  In the actual design, I chose Option 2 because this method can save time and memory since saving and reading data to/from disk and memory can be time-consuming.
  - Option 1: per request from Client, first send only one HTTP request and save the whole file in an intermediate memory, second send header with file size via gfs_sendheader(), then read file content from intermediate buffer and send to Client via gfs_send() in chunks.

- **Option 2:** per request from Client, send two HTTP requests as explained in last section Design. First request checks the file existence and get file size, while second request get file content and send to Client directly.

- **How to perform the curl option setup for a HEAD request.** First request sent from Proxy to Web Server is a HEAD HTTP request with no file content downloaded: receives the file and get the size of the file without copying or saving any file content. Setups of curl_easy_setopt() options:
  - CURLOPT_URL: set to URL of the requested file;
  - CURLOPT_NOBODY: sets to 1 to not download body data (file content);
  - CURLOPT_HEADERFUNCTION: register a header callback for receiving header data;

```
curl_easy_setopt(cur_handle_1, CURLOPT_URL, req_url);
curl_easy_setopt(cur_handle_1, CURLOPT_NOBODY, 1L);
curl_easy_setopt(cur_handle_1, CURLOPT_HEADERFUNCTION, header_func_callback);
curl_easy_setopt(cur_handle_1, CURLOPT_HEADERDATA, cur_header);
```

- **How to get size of the requested file after performing the HEAD request (first request sent from Proxy to Web Server).** After performing a file request using curl_easy_perform(), the size of the requested file can be obtained by retrieving CURLINFO_CONTENT_LENGTH_DOWNLOAD_T info using curl_easy_getinfo().

```
CURLcode cur_res_1;
cur_res_1 = curl_easy_perform(cur_handle_1);
curl_off_t file_size = 0;
CURLcode get_info_res = curl_easy_getinfo(cur_handle_1, CURLINFO_CONTENT_LENGTH_DOWN
```

- **How to perform the curl option setup for a WRITE request.** Second request sent from Proxy to Web Server is a regular HTTP request to receive the actual file content: send file chunks directly to Client while receiving from Web Server. Setups of curl_easy_setopt() options:
  - CURLOPT_URL: set to URL of the same requested file;
  - CURLOPT_WRITEFUNCTION: register a write callback for downloading file content and sending file content to Client directly.

```
curl_easy_setopt(cur_handle_2, CURLOPT_URL, req_url);
curl_easy_setopt(cur_handle_2, CURLOPT_WRITEFUNCTION, write_func_callback);
curl_easy_setopt(cur_handle_2, CURLOPT_WRITEDATA, ctx);
```

- **How to handle 404: the returned code from HTTP request is >= 400.** For HTTP code >= 400, the curl_easy_perform() should fail and should not return CURLE_OK. The curl option

CURLOPT_FAILONERROR needs to be set to 1 for curl library to fail when encountering 400+ HTTP code.

```
curl_easy_setopt(cur_handle, CURLOPT_FAILONERROR, 1L);
```

Tests

Run Proxy: `./webproxy -p 6200`

- **Test 1 - use workload.txt provided in original git repo to download pictures from Web Server:**
  - Default Web Server: "https://raw.githubusercontent.com/gt-cs6200/image_data"
  - `./gfclient_download -w workload.txt`
  - Pictures listed in workload.txt can be downloaded successfully.
- **Test 2 - download several large cat picture on random online website:**
  - Web Server: "https://cdn.pixabay.com/photo"
  - `./gfclient_download -w workload_test2.txt -s https://cdn.pixabay.com/photo`
  - As shown in the figure below, pictures listed in workload_yy.txt can be downloaded successfully:

```
vagrant@vagrant:/vagrant/projects/pr3/server$ ./webproxy -p 6200
current requesting path: https://cdn.pixabay.com/photo/2017/02/20/18/03/cat-2083492_1280.jpg
current requesting path: https://cdn.pixabay.com/photo/2013/05/30/18/21/cat-114782_1280.jpg
sent_size: 243102
sent_size: 613580
current requesting path: https://cdn.pixabay.com/photo/2014/11/30/14/11/cat-551554_1280.jpg
current requesting path: https://cdn.pixabay.com/photo/2017/02/20/18/03/cat-2083492_1280.jpg
sent_size: 320729
current requesting path: https://cdn.pixabay.com/photo/2013/05/30/18/21/cat-114782_1280.jpg
sent_size: 243102
sent_size: 613580
current requesting path: https://cdn.pixabay.com/photo/2014/11/30/14/11/cat-551554_1280.jpg
current requesting path: https://cdn.pixabay.com/photo/2017/02/20/18/03/cat-2083492_1280.jpg
sent_size: 320729
```

- **Test 3 - download picture that not exists on Web Server:**
  - "https://raw.githubusercontent.com/gt-cs6200/image_data/notexists_yy"
  - Picture is not downloaded and Proxy sends FILE_NOT_EXIST to client.
- **Other tests - stress test with different combination of input arguments for Proxy and Client.**
  - Client side arguments: thread count, request count
  - Proxy side argument: thread count

2. Part 2 - Proxy Cache

Design

The main task for this part to implement the inter-process communication mechanism between two multithreaded processes Proxy and Cache. The two processes communicates using two strategies for different info: a POSIX message queue for Proxy to send file request it received from Client to Cache, and multiple POSIX shared memory segments are created and each of them is used for transferring file info and content between a Cache worker thread and a Proxy worker thread.

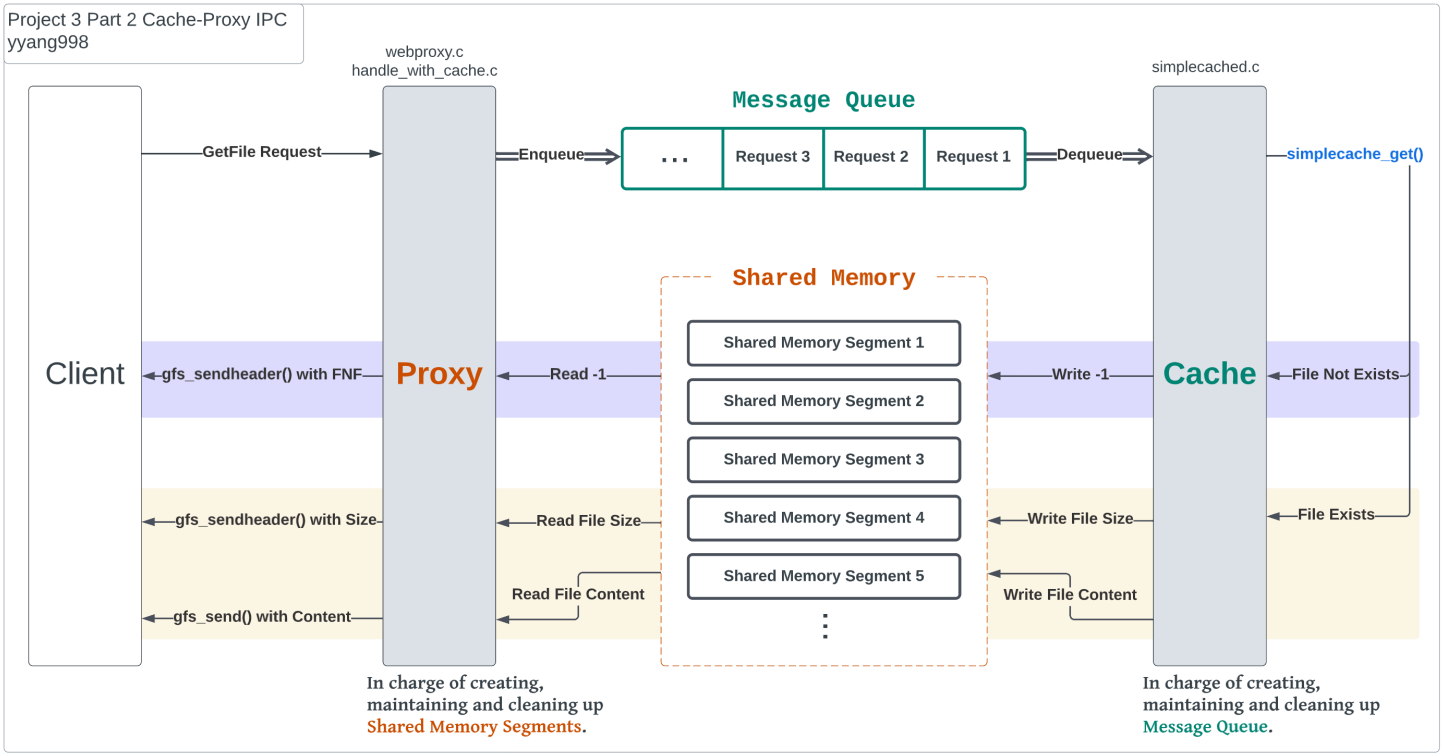
Since the two processes are multithreaded and there are shared resources between processes, synchronization is the most essential part. Resources are shared:

1. Between two different processes: Proxy and Cache;
2. Between threads in Proxy;
3. Between threads in Cache.

The resource sharing implementation between threads in one process is the same as the implementation in Project 1 Part 2. The design in this project focuses on the resource sharing between two processes Proxy and Cache.

General Flow Design:

The figure below is the general flow of the project.



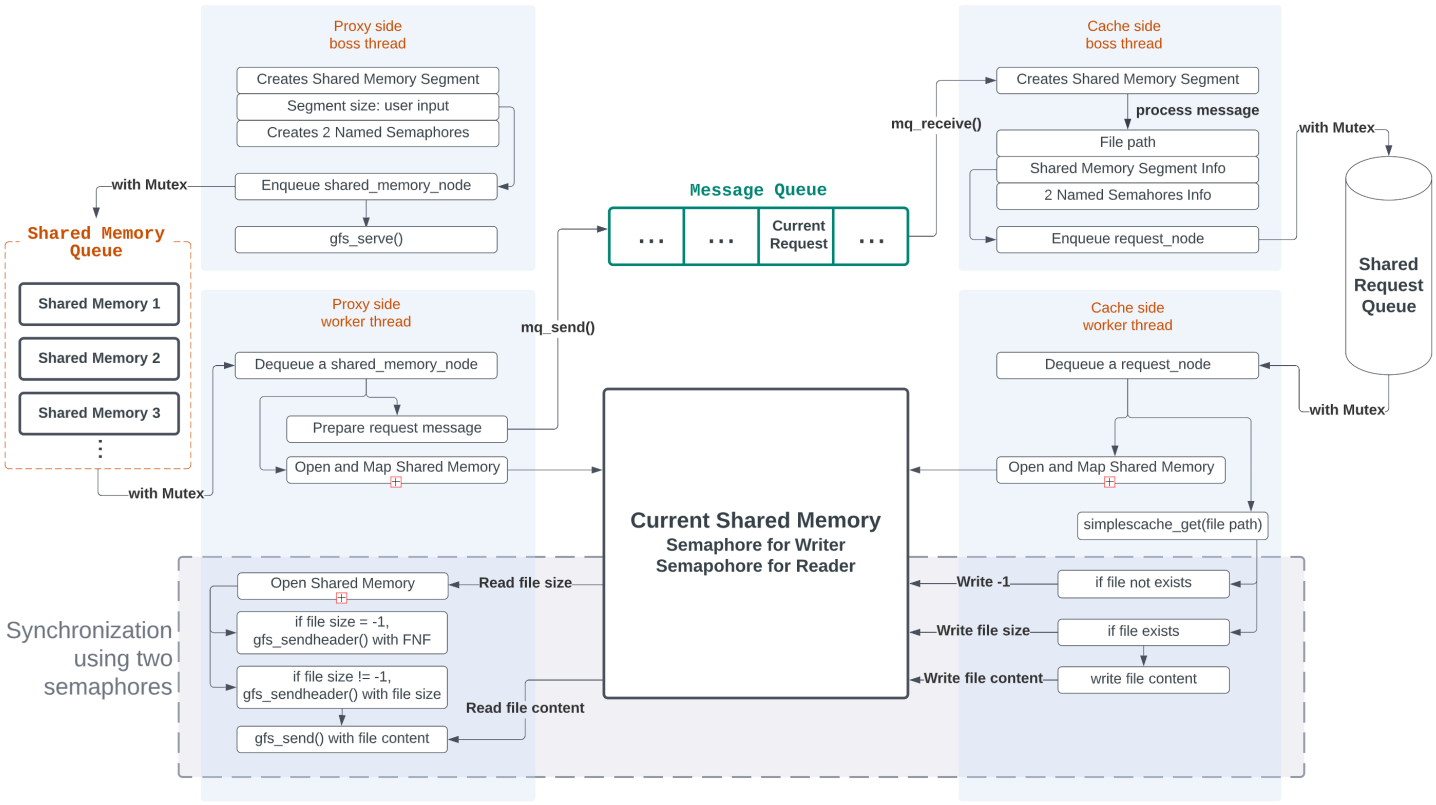
In the flow diagram above, there are three stages when handling a GetFile request sent from Client. (The handling of synchronization will be discussed later.)

- **Stage 1** includes 4 general steps for transmitting the request from Client to Proxy, then to Cache:
  i. Client sends GetFile request to Proxy;
  ii. Proxy processes the request and sends to Cache via Message Queue;
  iii. Cache receives the request from Message Queue;
  iv. Cache checks if the requested file exists in cache using simplecache_get();
  v. Go to Stage 2.
- **Stage 2** handles two scenarios:
  - Scenario 1 - if the file not exits in cache:
    a. Cache writes -1 to current linked shared memory segment;
    b. Proxy reads data from current linked shared memory segment, which is that the file size is -1;
    c. Proxy sends header back to Client via gfs_sendheader() with GF_FILE_NOT_FOUND;
    d. Client receives the header;
    e. The handling of current request is done: file not found.
  - Scenario 2 - if the file exits in cache:
    a. Cache writes file size to current linked shared memory segment;
    b. Proxy reads file size from current linked shared memory segment;
    c. Proxy sends header back to Client via gfs_sendheader() with GF_OK and file size;
    d. Client receives the header and prepares for the transfer of file content;
    e. Go to Stage 3.

- **Stage 3** is a looping process for sending file content in chunks:
    i. Cache writes a chunk of file content to current linked shared memory segment;
    ii. Proxy reads file content from current linked shared memory segment;
    iii. Proxy sends this chunk of content back to Client via gfs_send();
    iv. Client receives the content;
    v. If the whole file has been sent, the handling of current request is done; If not, go back to step 1 in this stage to continue sending the rest content of the file.

Multithreading Design of Proxy and Cache:

For the multithreading part, Proxy and Cache both uses boss-worker thread pattern. The detailed design of multithreading and the tasks handled by boss and worker threads are illustrated in the figure below.



The tasks of the threads in both of the processes are explained below:

- Proxy:
    - Boss thread:
        - Creates a shared memory queue shared by worker threads with mutex and cond, several POSIX shared memory segments and semaphores coupled with these segments:

            - **shared memory queue with mutex and cond:** stores shared_memory_node.
            - **shared_memory_node struct:** store the info about each of the shared memory segments.
            - Creates shared memory segments and two named semaphores (reader and writer) for each of the segments, and then stores the info (IDs of the segment and semaphores) in shared_memory_node struct.
            - Enqueue all shared_memory_node created into shared memory queue.
            - Total number of segments is input argument specified by the user.
        - When terminating the Proxy process, does the cleanup to remove all existing IPC objects that are created by Proxy:
            - Unlink shared memory segments
            - Unlink semaphores
            - Destroy shared memory queue
            - Stop gfserver
    - Worker threads:
        - Dequeue a shared_memory_node from shared memory queue: this shared memory segment will be used by this thread for receiving file size and content.
        - Sends request message to POSIX message queue created by Cache
            - Info needed for preparing request message:
                - The path of the requested file
                - The shared memory segment and two named semaphores coupled with it: this shared memory segment will be shared by current worker thread of Proxy and another thread of Cache that receives this message.
        - Reads file size from the shared memory segment and sends to Client.
        - Reads file content from the shared memory segment and sends to Client.
- Cache:
    - Boss thread:
        - Creates POSIX Message Queue for receiving request message from Proxy.
        - Creates Request Queue with mutex and cond for sharing parsed requests (request_node contains file path, and the IDs of shared memory segments and semaphores) received from Proxy between worker threads in Cache.
        - Keep receiving request message from the Message Queue, stores the info in request_node, and enqueue the node to Request Queue to be shared by worker threads.
        - When terminating the Cache process, does the cleanup to remove all existing IPC objects that are created by Cache:
            - Close and unlink the POSIX Message Queue.
    - Worker threads:

- Dequeue a request node from request queue.
- simplecache_get() the requested file.
- Writes file length to the shared memory segment.
- Writes file content to the shared memory segment in chunks.

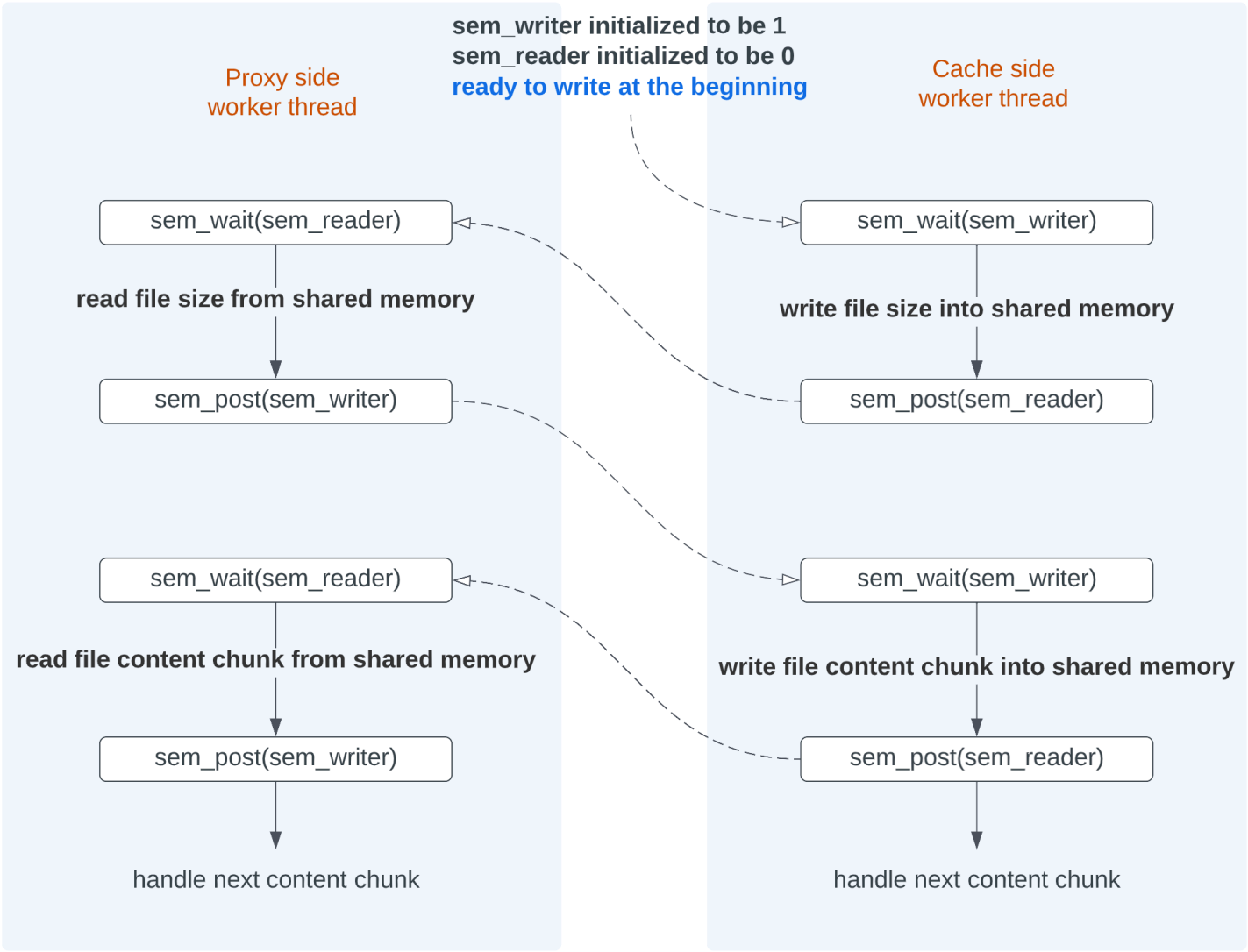
The synchronization mechanisms of shared resources between worker threads in Proxy/Cache are handled by pthread mutex as illustrated in the figure above, and the mechanisms are the same as the design in Project 1.
On the other hand, for a scenario when a Proxy thread and a Cache thread write into and read from the same shared memory segment (in the box with the dashed line in the figure above), the synchronization mechanism is implemented using two semaphores. The design is introduced in the next section.

Synchronization Design of Shared Memory between a Proxy thread and a Cache thread:

Two semaphores are used for each shared memory segment: one is a semaphore for Cache to write, another one is a semaphore for Proxy to read.

- Semaphore for cache `sem_writer` : Cache is the writer, whose worker threads write data (file size and file content) into shared memory.
- Semaphore for proxy `sem_reader` : Proxy is the reader, whose worker threads read data (file size and file content) from shared memory.

The key point here is to wait and post semaphores at the appropriate points in Proxy and Cache to avoid Proxy and Cache accessing data in shared memory at the same time, and also prevent deadlocks. The figure below illustrated the synchronization mechanism for this part of resource sharing.



Before sharing data through the shared memory segment, the `sem_writer` is initialized to be 1 and the `sem_reader` is initialized to be 0. So at the beginning of the transmission, writing to shared memory by Cache is permitted, but reading from shared memory by Proxy is restricted.

The process of the synchronization design are explained here:

1. Cache starts on waiting sem_writer - sem_wait(sem_writer), while Proxy starts on waiting sem_reader - sem_wait(sem_reader);
2. Cache writes file size into shared memory. After writing, Cache side signals Proxy side indicating that it is ok to read file size - sem_post(sem_reader); then wait to write file content - sem_wait(sem_writer).
3. Proxy then reads file size from shared memory. After reading, Proxy side signals Cache side indicating that it is ok to write file content - sem_post(sem_writer); then wait to read file content - sem_wait(sem_reader).
4. Cache writes file content chunk into shared memory. After writing, Cache side signals Proxy side indicating that it is ok to read this chunk of file content - sem_post(sem_reader); then wait to

write next chunk of file content.

5. Proxy then reads this chunk of file content from shared memory. After reading, Proxy side signals Cache side indicating that it is ok to write next chunk of file content - sem_post(sem_writer); then wait to read next chunk of file content.

Choices and Code Implementations

- Why use message queue for transferring request command between Proxy and Cache:
  - message queue is asynchronous, the Proxy worker thread could send the message into the message queue and continue to proceed to other work without waiting.
- Use POSIX instead of SysV for IPCs:
  - POSIX APIs are similar to file APIs.
  - Multi-thread safe: especially for message queue, APIs for receiving and sending messages are thread safe and much easier to be used.
- Use named semaphores instead of unnamed semaphores for shared memory:
  - Semaphores need to be shared between two processes Proxy and Cache.
  - Unnamed semaphores need to be stored in shared memory, using named semaphore can help saving space since they are stored in system folder.
- The shared memory segments will be reused to handle multiple requests. When a Proxy worker thread is done with a file request (sending and receiving data to/from a Cache worker thread), the shared memory segment should be reset and put back into shared memory queue to be used to handle another file request that will be processed by another pair of Proxy thread and Cache thread.
- Clean-ups:
  - IPC clean-up work when Proxy process and Cache process receives terminating and interrupting signals:
    - Proxy process handles shared memory and semaphore clean-ups since it is in charge of managing the resources related to shared memory.
      - Clean up shared memory segment:

        ```
        shm_unlink(cur_shm_node->shared_memory_seg_ID);
        ```

      - Clean up two semaphores with this segment:

        ```
        sem_unlink(cur_shm_node->sem_reader_ID);
        sem_unlink(cur_shm_node->sem_writer_ID);
        ```

        - Destroy shared memory queue after shared_memory_node are cleaned up and freed:

```
steque_destroy(shared_memory_queue);
pthread_mutex_destroy(&mutex_shm_queue);
pthread_cond_destroy(&worker_cons_shm_queue);
```

    - Cache process handles message queue clean-ups since it is in charge of managing the resources related to message queue.

```
mq_close(message_queue);
mq_unlink("/message_queue");
```

  - Clean-up work when Proxy/Cache worker thread finished handling of current request.
    - Proxy worker thread:
      - Un-map the shared memory segment
      - Reset semaphores: every Proxy worker thread will open and initialize the semaphores by themselves. So at the end of work for each worker thread, just close and unlink semaphores.

```
sem_close(sem_reader);
sem_close(sem_writer);
sem_unlink(cur_shm_node->sem_reader_ID);
sem_unlink(cur_shm_node->sem_writer_ID);
```

      - Put the shared memory used back to shared memory queue

```
pthread_mutex_lock(&(mutex_shm_queue));
steque_enqueue(shared_memory_queue, cur_shm_node);
pthread_mutex_unlock(&(mutex_shm_queue));
pthread_cond_broadcast(&(worker_cons_shm_queue));
```

    - Cache worker thread:
      - Un-map the shared memory segment
      - Free the current request_node
- Why unlinking semaphores at the end of Proxy worker thread instead of Cache?
  - When handling a request, Proxy worker thread will be the one to finish up the work. After Proxy worker reads the last chunk of the file, the processing of this request is done.
  - If Cache worker unlinked the semaphores after it writes the last chunk of content to shared memory, the Proxy worker could lose the access to the semaphores before reading the rest of the file content.

- The maximum length (message_queue_attr.mq_maxmsg) of the POSIX message queue created by Cache should be 10 to prevent an invalid argument error.

Tests

Tests used are from multiple sources (developed by myself and test posted in Piazza by classmates):

- My initial test (all arguments of proxy and cache are default):
  - With 50 threads, 500 requests and workload list provided:
    ```
    ./gfclient_download -p 25496 -t 50 -r 500 -w workload.txt
    ```
  - With 50 threads, 500 requests and workload list contains same file:
    ```
    ./gfclient_download -p 25496 -t 50 -r 500 -w workload_AllSameFile.txt
    ```
  - With 50 threads, 500 requests and workload list contains files cannot be found in cache:
    ```
    ./gfclient_download -p 25496 -t 50 -r 500 -w workload_AllNotFound.txt
    ```
  - With 50 threads, 500 requests and workload list contains files of different type and size:
    ```
    ./gfclient_download -p 25496 -t 50 -r 500 -w workload_Mix.txt
    ```
- My own stress test: varying the input arguments of Client, Proxy and Cache.
  - Client request file with size smaller/larger than shared memory segment size (proxy input).
  - Different type of file and file cannot be found.
  - Vary number of threads for Client, Proxy and Cache.
  - Vary the number and the size of shared memory segment for Proxy.
  - Vary the number of requests for Client.
- IPC Stress Test from 6200-tools by Miguel Paraz: https://github.gatech.edu/cparaz3/6200-tools
  - change minimum segment size of shared memory to 824 in ipcstress.py.
  - `python3 ./ipcstress.py ./ parameter`
  - `python3 ./ipcstress.py ./ base`
  - `python3 ./ipcstress.py ./ soak`
  - `python3 ./ipcstress.py ./ stress`
- Helder stress test by Helder Melendez: https://piazza.com/class/lco3fd6h1yo48k/post/1083

```
./simplecached -t 100 -c ./locals.stress.txt
./webproxy -t 100
./gfclient_download -t 100 -w ./workload.stress.txt -r 1000
```

References

Part 1 references:

- Sample source code for Libcurl programming: https://www.hackthissite.org/articles/read/1078
- The libcurl API easy interface: https://curl.se/libcurl/c/
- getinmemory example: https://curl.se/libcurl/c/getinmemory.html
- https://curl.se/libcurl/c/CURLOPT_NOBODY.html
- https://curl.se/libcurl/c/ftpuploadresume.html
- https://curl.se/libcurl/c/curl_easy_getinfo.html
- https://itecnote.com/tecnote/how-to-use-curlopt_headerfunction-to-read-a-single-response-header-field/
- https://curl.se/libcurl/c/libcurl-tutorial.html
- https://manpages.ubuntu.com/manpages/impish/man3/libcurl-tutorial.3.html

Part 2 references:

- Lectures: P3L3 and P3L4.
- Piazza posts.
- IPC in general:
  - https://man7.org/conf/lca2013/IPC_Overview-LCA-2013-printable.pdf
  - Beej's Guide to Interprocess Communication: https://beej.us/guide/bgipc/pdf/bgipc_a4_c_1.pdf
  - https://piazza.com/class/lco3fd6h1yo48k/post/977
  - https://www.omscs-notes.com/operating-systems/inter-process-communication/
  - https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_system_v_posix.htm
- Message Queue:
  - https://www.softprayog.in/programming/interprocess-communication-using-posix-message-queues-in-linux
  - https://users.pja.edu.pl/~jms/qnx/help/watcom/clibref/mq_overview.html
- Shared Memory:
  - POSIX Shared Memory with C Programming: https://linuxhint.com/posix-shared-memory-c-programming/
  - https://man7.org/linux/man-pages/man7/shm_overview.7.html
  - POSIX Shared Memory In Linux: https://www.softprayog.in/programming/interprocess-communication-using-posix-shared-memory-in-linux
  - POSIX shared-memory API: https://www.geeksforgeeks.org/posix-shared-memory-api/

    - sem_open()--Open Named Semaphore: https://www.ibm.com/docs/en/i/7.1?topic=ssw_ibm_i_71/apis/ipcsemo.html
    - https://piazza.com/class/lco3fd6h1yo48k/post/1022
- Semaphore:
    - https://man7.org/conf/lca2013/IPC_Overview-LCA-2013-printable.pdf
    - https://www.youtube.com/watch?v=ukM_zzrIeXs
    - POSIX Semaphores In Linux: https://www.softprayog.in/programming/posix-semaphores
    - https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_semaphores.htm
    - Semaphore Basics: https://www.softprayog.in/programming/semaphore-basics#semop
    - semaphore (posix) for synchronization: https://linux.die.net/man/7/sem_overview
- Linux signals:
    - https://www.thegeekstuff.com/2012/03/catch-signals-sample-c-code/
- cache server:
    - https://www.techtarget.com/whatis/definition/cache-server
- tests:
    - https://github.gatech.edu/cparaz3/6200-tools
    - https://piazza.com/class/lco3fd6h1yo48k/post/1083