Project 3 README

Spring 2023

Shide Qiu

Part 1 Understanding

In the first part of the assignment, it uses the same structure with the project 1 assignment. The main change is to replace the part of the code that retrieves the file from disc with code that retrieves it from the web. The libcurl's "easy" C interface is used for retrieving the file from the web. The procedures are described below:

1. The proxy keeps listening on a specific socket.
2. the client connects to the proxy's socket and sends the URL to proxy.
3. The proxy receives the URL and retrieve the file from the URL.
4. The proxy sends the file back to the client.
5. The client receives the file and save it to the local drive.

In part 1, all multithreaded programming is handled by the code already. The proxy will create multiple threads and calls handle_with_curl in each thread. The focus is the libcurl code implementation in handle_with_curl.c for file retrieving from the web server.

Part 1 Implementation and Testing

The workflow is described in the picture below. According to the libcurl API document, the file retrieving was implemented.

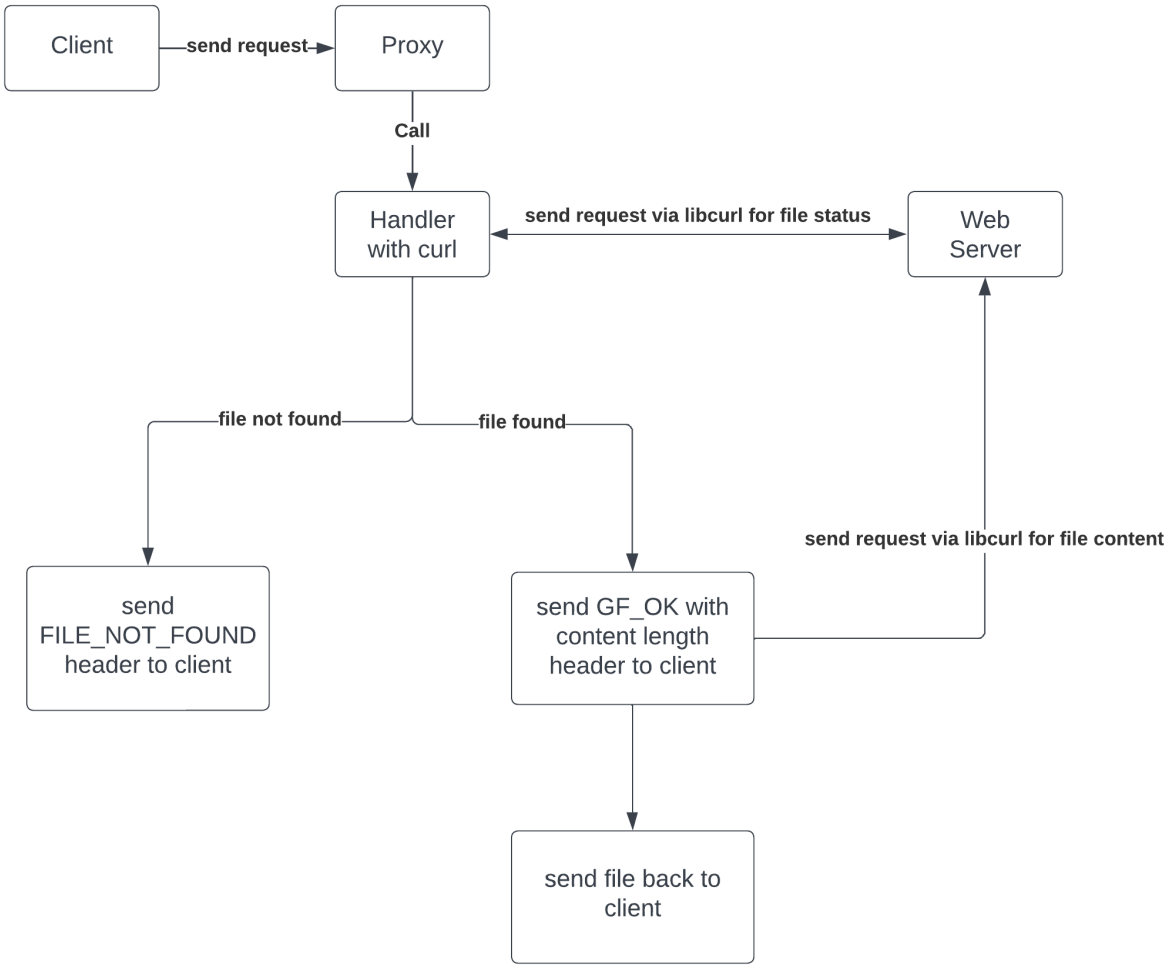
Specifically, the file status and file length were retrieved by curl handle curl_easy_getinfo () with parameters of CURLINFO_RESPONSE_CODE and CURLINFO_CONTENT_LENGTH_DOWNLOAD_T respectively. It's worth to notice that the CURLINFO_CONTENT_LENGTH_DOWNLOAD is deprecated, and we should use the new parameter instead to get the content length.

To get the file content, the curl_easy_setopt() function was used with parameter CURLOPT_WRITEFUNCTION and a write_callback function. Also, a CURLOPT_WRITEDATA parameter was passed to the curl_easy_setop() with ctx pointer. The ctx pointer was passed to the write_callback function. The callback function gets called by libcurl as soon as there is data received that needs to be saved and the ctx pointer points to the delivered data. The function could be called multiple times and a chunk of data is delivered each time until all data is delivered.

According to the API, the write_callback function is called before getting the file status and file length. However, we need to get the file length for gfs_sendheader function first and then start to use gfs_send function for file sending. The main challenge is to solve this problem.

Firstly, I tried to use a large buffer and write the file content into the buffer in the callback function. Then I could safely retrieve the file status and file length for gfs_sendheader function and then call gfs_send function with the buffer. However, the extra writing step will reduce the

overall performance. Moreover, some files in the GradeScope test environment are probably larger than the buffer size which will cause memory leaking.



In the end, I called curl function twice to solve this problem. The first curl function was to get the file status and file length only. Once I got this information, the header was sent to the client and the curl was closed and cleaned up. After that, I called the curl function second time to get the file content only. In the write_callback function, I used gfs_send function sent the data retrieved from the web server to the client directly. Therefore, no buffer is needed and no memory leaking happened.

The test is done in the VM. A couple print functions were added in the handle_with_curl code to check if the code works as expected. Firstly, the request URL was printed to make sure the proxy could receive requests from the client. Secondly, we printed out the file status and file length in proxy to make sure we could retrieve the file information from the web server as expected. Lastly, we printed out the number of bytes received for each write_callback to make sure the function is called successfully and compared it to the file size to make sure we collect the right number of bytes.

With those print functions, we launched the client while the proxy is running to check if the terminal messages was valid and if the new file was created by the client. Also, we edited

the workload.txt to add some invalid paths. With the invalid path, we tested if the URL was printed and if the header status was FILE_NOT_FOUND.

Part 2 Understanding

The part 2 is based on part 1, we will add a cache process in between the proxy and server. Specifically, once the proxy receives the file request from the client, it calls the cache process to check if the file is stored in cache. If the file is in the cache, the process will send the file back to client. If the file is not in the cache, the process will send FILE_NOT_FOUND back to the client (it will not check the web server in this project).

In the project, the data channel and the command channel need to be separated. File content is transferred through the data channel while the file request is transferred through the command channel. The data channel needs to be implemented with shared memory and the command channel is implemented using the message queue.

Either System V or POSIX API can implement the related IPC mechanisms. The System V IPC facilities were designed independently of the traditional UNIX I/O model, and consequently suffer a few peculiarities that make their programming interfaces more complicated to use. The corresponding POSIX IPC facilities were designed to address these problems. Specifically, the main difference between them is listed below:

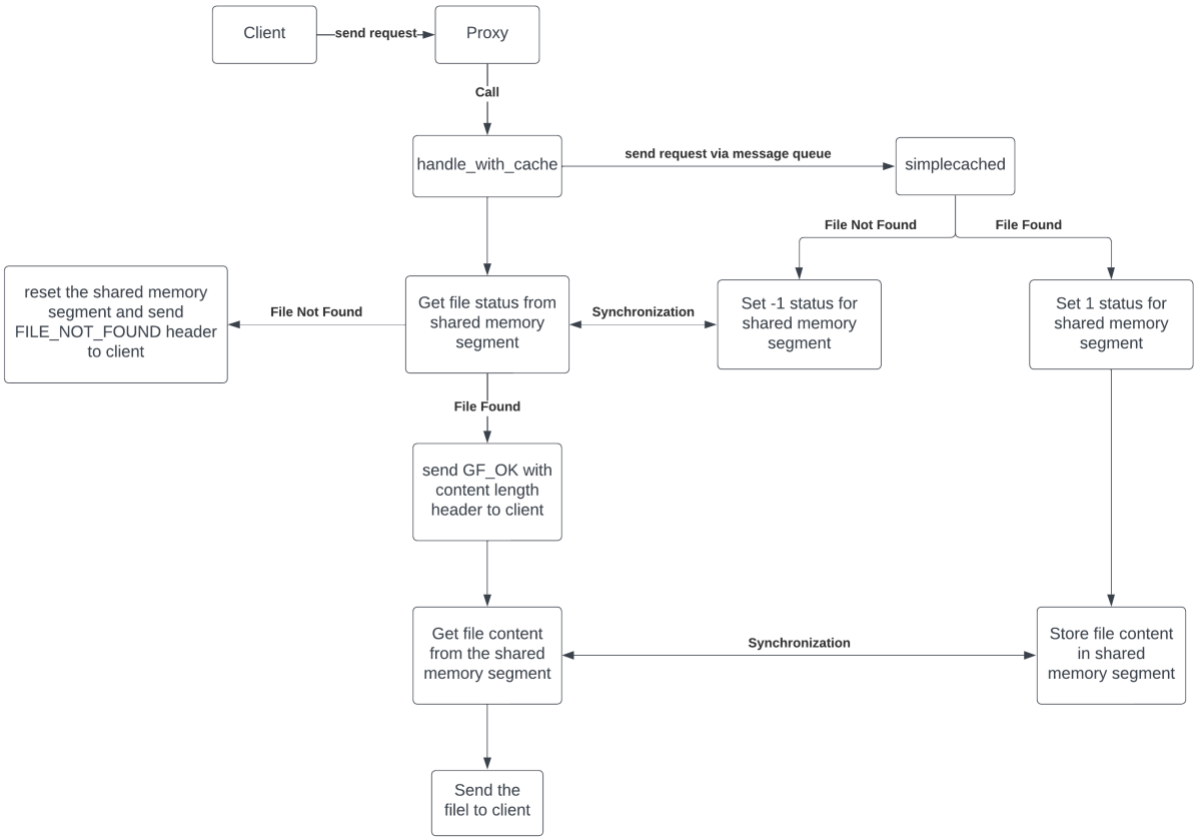
- The programming interfaces for the System V IPC facilities are inconsistent with the traditional UNIX I/O model. The POSIX IPC model is more consistent with the traditional UNIX file model.
- The POSIX IPC interface is simpler than the System V IPC interface.
- The POSIX IPC objects are reference counted and this simplifies object deletion.
- The POSIX IPC facilities are not quite as widely available as their System V IPC on older UNIX systems (older than 2.6.x). However, this is not an issue for our VM.

Overall, the POSIX IPC API is used in this project due to its advantages over the System V IPC. The focus of this part is to understand and use IPC mechanisms. Thus, the cache implementation is provided and can be accessed through the simplecache API. Most of the job will be on relaying the contents of the cache to the proxy by way of shared memory. The main challenge of this assignment is the synchronization of shared memory. The file info needs to be saved in the shared memory segment before being accessed by the handler. Also, the file content needs to be saved into shared memory segment before being retrieved by the handler.

Part 2 Implementation and Testing

The workflow is described in the figure below. Firstly, the shared memory segments and semaphores are created in the proxy. Then the client sends file request to the proxy. The proxy creates worker thread with handle_with_cache function and pass the shared memory queue to the thread. In the handle_with_cache function, it will create a message queue for command channel. And it will add the message including segment id and the file path to the message queue. At the same time, the simplecached will open the same message queue and retrieve the

command from the queue. Once it gets the command, it will open the shared memory segment with id specified in the command message and try to retrieve the file from the cache according to the file path provided in the command message. If the file is not found, it will assign the message to the shared memory segment and set the message type to -1 which represents the file is not found. On the other hand, if the file is found, it will also assign the message to the shared memory segment but set the message type to 1 which represents the file is found. Also, the file length and file content are written into the message in the shared memory segment. Once it's done, the handle_with_cache also opens the same shared memory segment and retrieves the message in the shared memory segment. If it gets message type of -1, it will send the GF_FILE_NOT_FOUND header back to the client and reset the shared memory segment. However, if it gets message type of 1, it will send the GF_OK header back to the client and start to retrieve the file content. It will then send the file content to the client and reset the shared memory segment in the end. There are some details need to notice including the command message structure, shared memory segments and semaphores initiation, and synchronizations for shared memory segments access. I will describe more about how and why I implement them below.



The shared memory segments and semaphores are initiated in the webproxy and opened in the handle_with_cache and simplecached later. Since there are multiple shared memory segments need to be used independently, a shared memory queue is created to hold all those segments. Each shared memory segment is used for one message (file request and file write) and it will be reset when the message is reset. In the beginning, the proxy will create

one semaphore for the shared memory queue. For each segment, the proxy creates one shared memory with that segment id. It also creates two semaphores with that segment id and we will talk about them in the later section. Each shared memory segment will then be added to the shared memory queue for future usage. Before closing the proxy, each shared memory segment in the memory queue will be freed. It is important to name shared memory segment and its semaphores correctly to make sure we could open it later as expected. Here, we used “shm_id” for shared memory segment name and “sem_1_id” & “sem_2_id” for semaphore names.

The same message structure is used in both command channel and data channel since they both need the segment id to open the same shared memory segment. The structure is designed as below:

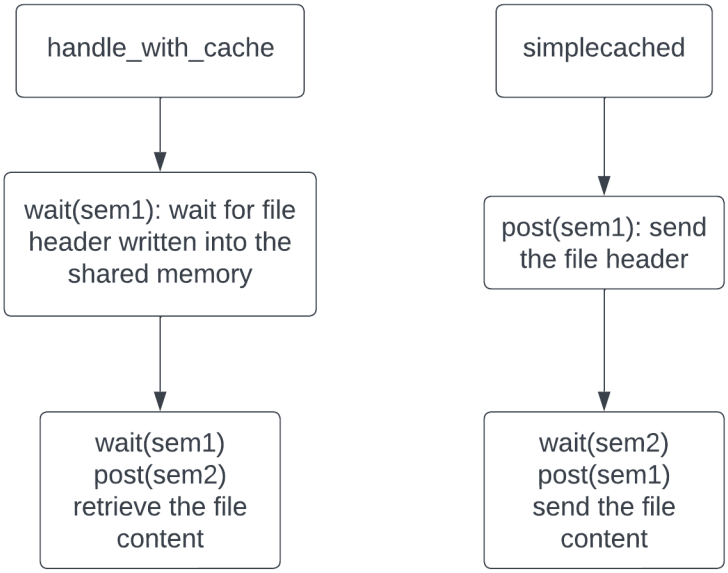
```
struct message {
    int id;
    int type;
    char path[256];
    char content[824];
    int file_len;
    int content_len;
};
```

The id represents the shared memory segment id, the type represents if the file is found or not found. The path represents the requested file path. The content represents the file content. The file_len and content_len represent the file length and content length of data chunk in each transfer. We need to get one shared memory segment from the shared memory queue at the beginning so that we could assign message in that shared memory segment. When we add the message in message queue, we only need to set the id (get from the shared memory segment) and path.

Once the simplecached retrieves the message from the message queue, it will try to retrieve the file from the cache according to the path. If the file is not found, it will set the message type as -1 and add the message to the shared memory segment with the id specified in the message id. At the same time, the handle_with_cache also opens the shared memory segment with the same id and tries to retrieve the message type to check if the file is found or not. Since the file is not found, it will send the header back to client and reset the message as well as the shared memory segment. Also, it will add the shared memory segment back to the shared memory queue so that it could be reused in the future.

If the file is found, the simplecached will set the message type as 1 and set the message file_len. The message will be added to the same shared memory segment, and it will start to write file content into the message content. The file content may not be written into the message content all at once due to the large size. Every time a piece of data chunk is written into the message content, the content length is also recorded in the message content_len. Then the writing process will be paused and the handle_with_cache will start to retrieve the file content and send it back to the client. Once it is done, the simplecached will take over the control and write the remaining file content to the same message content (replace the old content) until the whole file is transferred. In the end, the handle_with_cache will reset the message and shared memory segment. Also, it will add the shared memory segment back the shared memory queue as described before.

Lastly, we will talk about the synchronization implemented in the project. There are two synchronizations: one is for shared memory queue and the other is for the message in the shared memory segment. As for the shared memory queue, a semaphore is used to make sure only one worker thread could select and remove the memory segment from the shared memory queue at the same time. We don't care when the worker thread adds the segment back to the queue since the order doesn't matter. The semaphore is created with value 1. The critical section is to remove and save one shared memory segment from the shared memory queue. The wait operation is called before we enter the critical section, and the post operation is called after we exit the critical section. Thus, with semaphore protection, we will not have multiple worker thread trying to get the same memory segment and write different message into one segment which cause inconsistence.



The other synchronization is more complicated and described in the figure above. Two semaphores are used here in each shared memory segment for writing and retrieving message. The semaphore 1 is initialized with value 0 and semaphore 2 is initialized with value 1. After adding the message into the message queue, the handle_with_cache will call wait(sem1) and it will pause there waiting for header information to be written into the shared memory. At the same time, the simplecached will retrieve the file information from the cache. If the file is not found, it will write the invalid information into the shared memory segment and then call post(sem1) to let handle_with_cache know that the header information is ready to retrieve. Then the handle_with_cache will retrieve the header and send it back to the client. The message is cleaned up and the shared memory segment is added back to the shared memory queue for reuse. It's worth to notice that in this case, the semaphore 2 is not used and the semaphore 1's value is 0 in the end.

On the other hand, if the file is found, the same post(sem1) and wait(sem1) are called and the header is also sent to the client. After this, it will start to transfer the file content. The file content is transferred chunk by chunk. After one chunk of data is written into the shared memory segment, it will wait until the data is retrieved and sent to the client for next chunk writing. As for

synchronization implementation, the handle_with_cache will call wait(sem1) and pause for data being written into the shared memory segment. At the same time, the simplecached will call wait(sem2) first to lower the value of sem2 to 0. Then it writes the data chunk into the shared memory segment and calls post(sem1) to wake up the handle_with_cache. The simplecached will repeat the procedure until all data chunks are written into the shared memory segment. However, the sem2 is 0 now and the wait operation will pause the writing process. Since the handle_with_cache is wakened up, it will retrieve the data chunk from the shared memory segment and send it to client. Then it will call post(sem2) to wake up simplecached for next data chunk writing. Until all data is transferred, the message will be cleaned up and the semaphores return to initial values. Also, the shared memory segment is added back to the shared memory queue for next use.

The test is done in the local VM. A couple print functions are used for debugging purpose. Firstly, semaphore names and shared memory name are printed out in the proxy to confirm they are created successfully. Secondly, the message is printed out with its id to confirm that different shared memory segment is opened with different message. Thirdly, the file status is printed out to check if retrieving from cache works as expected. Lastly, the semaphore's value is printed out before any operation to make sure it works as expected.

In the end, I adjusted the number of threads and requests in the gfclient_download to check if everything works as expected with more threads and requests. Also, I added some large files in to cache folder and update the workload document to make sure all files with different sizes are retrieved successfully.

Reference

1. The libcurl API https://curl.se/libcurl/c/
2. Kerrisk, Michael. *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, 2010.
3. POSIX vs. System V IPC https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/POSIXvSysV.html
4. Message Passing With Message Queues https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/MQueues.html
5. Shared Memory https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/ShMem.html#posix-shared-memory
6. Semaphores https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/IPCSems.html
7. Operating System – Three Easy Pieces