Project 3 README file

This is **Yufeng Xing's** Readme file.

Note - You will submit a PDF version of this readme file. This file will be submitted via Canvas as a PDF. You may call it whatever you want, and you may use any tool you desire, so long as it is a compliant PDF - and for us, compliant means "we can open it using Acrobat Reader".

Project Description

Your README file is your opportunity to demonstrate to us that you understand the project. Ideally, this
should be a guide that someone not familiar with the project could pick up and read and understand
what you did, and why you did it.

Project Design

1. Part 1 - LIBCURL Easy APIs

The first part is basically a simple example on showing how we can use the **Curl's Easy Interface** to create HTTP response and receivce the data. The only thing we should focus on is how to use the interfaces in the proper way.

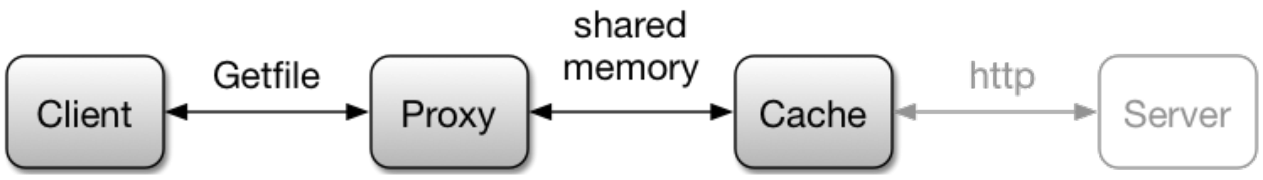
We can create an "easy handle" using the `curl_easy_init` call. The type of easy handle is a `CURL` type (which is defined by `typedef void CURL;` in the `curl.h` file). We then set your desired set of options in that handle with `curl_easy_setopt` function. In this case, we will only use the `CURLOPT_URL` option, which can be used to set the URL that we are going to work on. The `curl_easy_perform` call will start to perform the transfer. It will get all the content of the URL that we are requesting and then directly print all the contents directly in the output. It will be a good manner if we do the `curl_easy_cleanup` after we perform a translation. This function is called to eliminate all the data structures related to the easy handle that we have set.

However, sometimes instead of directly print the result to the terminal, we would like to save the contents as a file or we just want to print some of the content that we need instead of all the contents. In these cases, we can modify the write **callback function** of writing the data.

If the callback function is not referred by us, we will get a result of the arbitrary output. Or we can use the `curl_easy_setopt` to reset the callback function to the callback function we are going to use for other specific cases (e.g. partially print, save as a file, etc.). The option that can be used to reset the callback function is `CURLOPT_WRITEFUNCTION`.

2. Part 2 - Cache Server

In this part, we are told to implement a cache server, which can be used for improving the efficiency of the file transformation. The basic idea is that before we request the file from the HTTP server, we will first look into our cache server to see if we have our code locally in the cache. If the file is in the cache, we won't request to the HTTP server because we can directly send this file through the IPC.

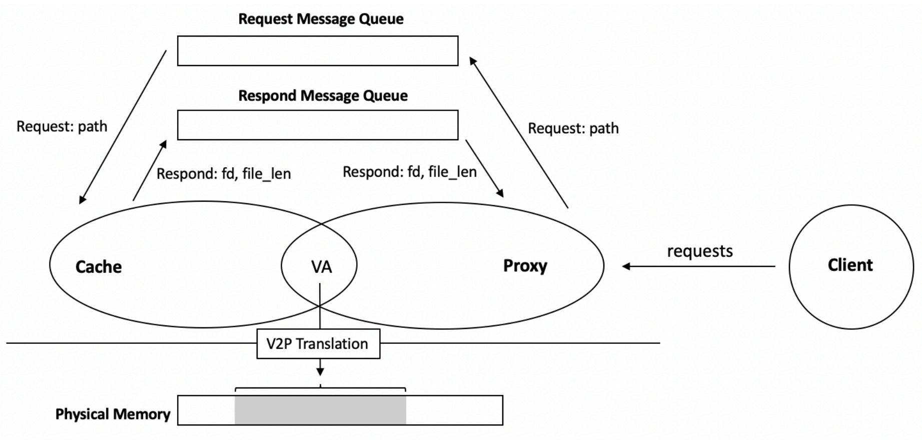


The basic idea is that, at the beginning, the client will send requests to the proxy. Because the proxy has no idea if it has the file (Only cache knows that because of it is initialized with the `simplecache[.ch]` files). Thus, the proxy will only kown the paths (or URLs) we would like to request and the cache only knows the files it has. They have to communicate to each other in order to get the whole information.

So after the client sends some requests, the proxy will make a request with the information about the path. This request will be send to the proxy server and then the proxy will send a request to the cache with this path. The request will be sent to the request message queue and the cache will then read from this queue to get the request message.

After getting the request message with the path, the cache will try to find this file by the `simplecache_get` function provided by the `simplecache[.ch]` files. If the file is not found, we will get an file descriptor `fd` equals -1. However, if the file does exist in the cache, we will get a file descriptor > 0 and then we can read the file by `lseek` to get a lenth of this file `file_len` (aka `size`). Then these information will be added to the request for creating a response.

Then the cache will send the respond back to the proxy so that the proxy can know the information about the file length and whether the file is in the cache or not in the cache, so the proxy and the cache can then start transferring the data via some IPC methods. The proxy can directly send the information to the client when it reads data from the cache.



We will discuss more detailed information about this design later.

Trade-offs and Choices

1. System V Vs. Posix

In my solution, I chose the **Sys V APIs** because it is exactly the one that is taught by the instructor. However, I am regret for that because I finally find out that Sys V is not safefy for multithreading cases and unfortunately, I ran into many troubles because of that. I am stuck with the mutex and semaphores and it can be really hard for me to synchronize between different processes. In the end, I still have the efficacy problem which I am not sure why.

If I have another choice, I will suggest using posix from the beginning.

2. Attempts to Deal with Efficacy Problems

After reading the Slack and Piazza, I have figured out two potential reasons of the efficacy problems, but none of them works for me.

- The **first** approach is to simply remove all the `printf` s because the printing instructions reduce the overall performance. So I simply remove all the `printf` but I still have this error.
- The second approach metioned by **Muhammad Doukmak** (@1353) is that,

> I resolved it by realizing that I had a bottleneck in my cache server's boss thread. I was calling the simplecache_get in the boss, then passing the result of that to the workers. It turns out that simlpecache_get was an expensive operation, so I was serializing all the requests because only 1 boss thread could do that processing. You should design your cache so that as much processing can be done by the workers as possible. Keep the boss very lean and fast.

Basically, it is telling me to move the `simplecache_get` function from the boss thread and then put it into the worker thread. I indeed put the `simplecache_get` function in the boss initially, but I still have the efficacy after I move this function to the workers.

Therefore, I have to say that I can not handle this error. Someone mentioned that this problem can be unexpected for students who use the Sys V API, and they can not explain why we are having this error.

3. Choices of IPC Methods

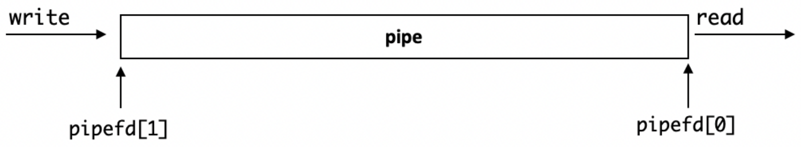
In fact, we have several IPC method that can be used by us.

(1) Pipe

Before we call the function `pipe` to create a pipe for communication, we must maintain a 2-item array `pipefd` of the `int` type. The array `pipefd` is used toreturn two file descriptors referring to the two ends of the pipe. `pipefd[0]` refers to the read end of the pipe and `pipefd[1]` refers to the write end of the pipe. Then we can use the following code to create this pipe.

```
int pipefd[2];
if (pipe(pipefd) == -1) {
    printf("Unable to create pipe\n");
    return -1;
}
```

This pipe seems like the following diagram,



Then, we can use `fork` to create a new process. In the parent process, we are going to use `write` to send data to the pipe. For example,

```
write(pipefd[1], "Hi ", STREAM_SIZE);
```

Note because the parent process will only write to the pipe, it will never use the `pipefd[0]` endpoint. Thus, we can `close` this end for the safety concerns.

```
close(pipefd[0]);
```

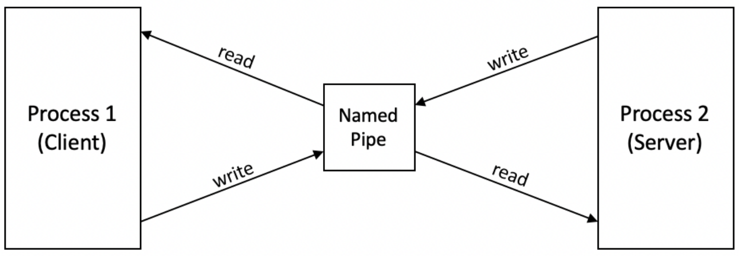
Similarly, in the child process, we will first close the unused write endpoint and then `read` from the read endpoint.

```
char readmessage[STREAM_SIZE];close(pipefd[1]);
read(pipefd[0], readmessage, sizeof(readmessage));
```

(2) Named Pipe

Now, we have implemented the pipe, but there is a problem. The pipe can only be used for related processes (e.g. parent process and child process) so that they can share the same pipe easily. However, there can be many situations that we want two unrelated processes (e.g. the client and the server) to communicate via the pipe. At this moment, the pipe will no longer be useful. Instead, we will use the concept of the **named pipe**. The named pipe is also called **FIFO** (means first in first out).

The named pipe is nothing like a real pipe, instead, it seems more like a special kind of **file**. The main idea of the named pipe is that we can attach this file to both of the two processes so that they can read or write from this file. The file follows the FIFO rule, which means that we will first read the first thing write to this file.



To create a named pipe, we must select the directory of the named pipe file. Commonly, we will put this file under the `/tmp` directory so that it can be automatically deleted after rebooting. In our case, we will store the file in the directory `/tmp/myfifo`,

```
#define FIFO_FILE "/tmp/myfifo"
```

Then the function `mkfifo` is called in each process to establish a FIFO file and attach this file to the current process. `DEFFILEMODE` is the permission we will use for this file, which simply means `S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH` (permission for all the users). You can also assign `0666` for this argument, which means basically the same.

```
mkfifo(FIFO_FILE, DEFFILEMODE);
```

After we attached this pipe two these processes, they can then communicate with each other via this named pipe. Let's see how it works. When one process wants to send a message to another, it will first `open` the file (as the write-only mode) and then `write` the message to the named pipe. After writing, the file should be closed by the `close` function.

```
int fd = open(FIFO_FILE, O_WRONLY);
write(fd, "Hello", strlen("Hello")+1);
close(fd);
```

Then, in another process, if we want to read from this process, we should first open the file (as the read-only mode) and then `read` the message from the named pipe. After reading, the file should be closed by the `close` function.

```
char str[80];fd = open(FIFO_FILE, O_RDONLY);
read(fd, str, 80);
close(fd);
```

(3) Shared Memory

The shared memory will be attached to both these two processes and then they are able to communicate via this shared memory. In this case, we are going to use the **Sys V APIs** for sharing the memory.

The basic idea of the first program is that, firstly, the shared memory is created by the `shmget` function. In this case, the unique key will not be generated, instead, we will assign `IPC_PRIVATE` (i.e. value 0) to this argument. This is because we have a private IPC and we don't really care about this unique key.

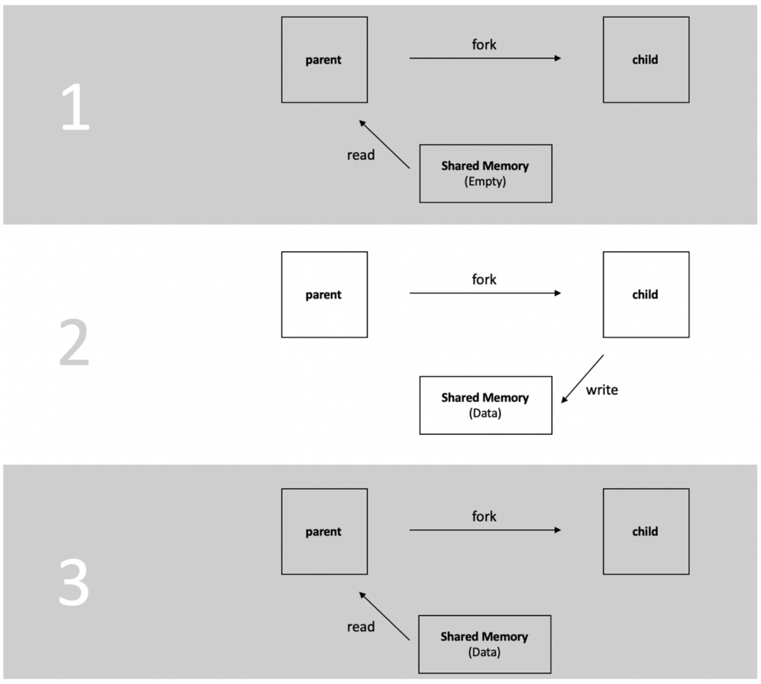
Then, the `shmat` is called to attach the current process to this shared memory we have created by `shmget`. The return value of `shmat` is `void *` so that we have to convert it to the data structure we will use to store the shared data.

Although we can use a simple string for communication, in this case, we are going to use a user-defined data structure called `datamemory`. Similarly, this data structure has two elements as we have discussed. One is the `message` variable that we are going to use for storing the shared data. The other is the `size` variable that represents the length of the shared data.

```
typedef struct {
    char message[SHARED_MEM_SIZE];
    int size;
} datamemory;
```

Then the `fork` is called to create the child process of the current process. Because we have already called `shmat` , now both of these processes are attached to the same shared memory. Because they can do write and read to the shared memory simultaneously, we have to do some **synchronizations** to make sure that the child process writes in the first place, and then the parent will read from the shared memory. We are going to discuss the synchronizations later and now we will simply use `sleep` to synchronize these two processes.

First, the parent will read from the memory and reads nothing because the memory is empty now. So the child should `sleep` at the beginning. After the parent reads from the memory, it should `sleep` and wait for the child to write. In end, after the child writes to the shared memory, the parent will read the shared memory again and get the shared data. In fact, we should have a workload of the following diagram.

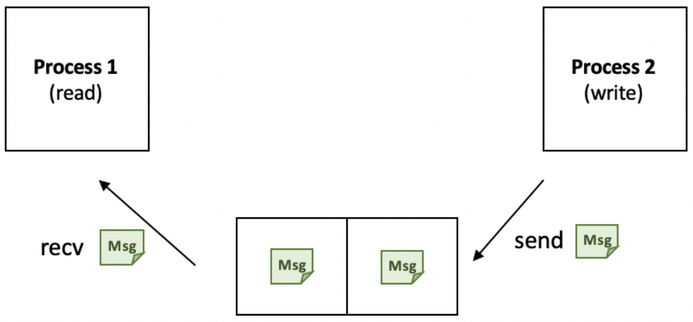


After the child writes to the memory, it will wait for another second before it returns (Why? You can delete the `sleep(1);` to see what will happen). It should also detach from the shared memory because it finishes its task. However, it must not destroy the shared memory because the parent may be reading from this memory.

When the parent process successfully read the data from the shared memory, it can be detached from it, and then we can call `shmctl` to destroy the shared memory because it will no longer be useful.

(4) Message Queues

The Sys V API for creating a message queue is quite similar to the shared memory. The `msgget` is called create a message queue and an identifier of this queue will be returned. To create a message queue, we should get a unique key for this message queue. The function `msgsnd` is used to send the data to the message queue and `msgrcv` is used to read the data from the queue. After the reader reads from the queue, `msgctl` will be called to destroy the message queue.



4. Why Using Message Queues

- The main difference between the message queue and the pipe is that the latter one can only be used for IPC between relative processes.
- The main difference between the message queue and the named pipe is that the named pipe relies on the file, while the message does not rely on the file depend on the processes.
- The main difference between the message queue and the shared memory is that the former one doesn’t need synchronization.

Note: It is important to know that we have to do synchronizations between threads if we decide to use the message queue because the Sys V MQ is not thread safety.

We can use either the named pipe or the message queue for sending the requests and the responds. Because the instruction of the project 3 suggests using the message queue, I choose this in my implementation.

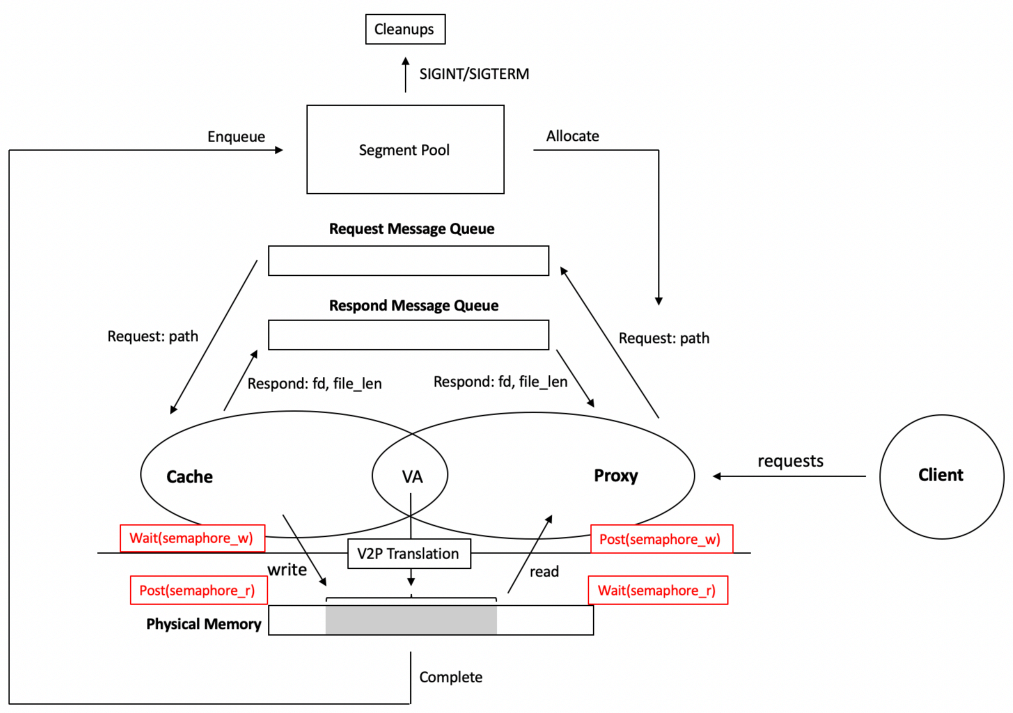
5. Why Using Two Message Queues

So the potential problem we implied is that if we can use a single message queue to achieve the same result of requesting and responding? Well, the answer is yes but we have to wait for longer time. Let's imagine that if we use a single message queue, the queue should be locked after a request is send to the cache. We can not send another request because then the messages in the queue we will pop is not the correct one. Thus, we have to wait for longer time and it actually reduces our efficiency. That's why we are using seperate message queues for requests and responds.

Flow of Control

The flow of control of part 1 can be really simple because we only have to change the `handle_with_curl` file. This file is quite similar to the `handle_with_file` so it is clear for us.

We have briefly introduced the design of the part 2. Now, let's see the real **flow of control** with all the details.



To understand this flow of control, we have to care about the following details,

- We have to build a segment pool for the shared memory segments. The segment is allocated to a request before it is sent to the request message queue.
- We have to do synchronzations between the cache and the proxy with the aspect of read/write from the shared memory. Two semaphores are used here to maintain a proper "writing-reading-writing" pattern.
- The segment should be put back to the pool after sending a file in the cache, so that it can be reallocate to another request in the future.
- We have to clean up all the segments we have created when we receive a SIGINT (which means ctrl+C) or a SIGTERM (which means the process properly ends).

Code Implementation

1. Part 1 - LIBCURL Easy APIs

In order to reset the callback function properly, we should also maintain a `memory` data structure that can be used both in the callback function and the main function. This structure will help us store the data that we receive as a response from the HTTP server.

There need to be 2 elements maintained in this function,

- `response` is the pointer to a string that can be used to store the data received from the HTTP server
- `size` is used to keep the length of the data we receive

So this function can be realized by,

```
typedef struct memory {
    char *response;
    size_t size;
} datamemory;
```

We would like to redefine the name of this structure as `datamemory`. To make this structure visible for both the main function and the callback function, we will create an instance of this structure in the main function and then pass its pointer to the callback function. So the callback function can also have access to this instance. We will call the **pointer** to this instance of `datamemory` we have created as a `chunk`.

Before we pass the `chunk` to the callback function, we have to remember to allocate a range of memory to our instance. We will use the `malloc` function in our case and we will also have to initialize the value in this by NULL and 0,

```
datamemory *chunk = (datamemory *) malloc(sizeof(datamemory));
chunk->response = NULL;
chunk->size = 0;
```

Remember we can also use a more advanced method to initialize the structure by `memset` and it can be used as,

```
datamemory *chunk = (datamemory *) malloc(sizeof(datamemory));
memset(chunk, 0, sizeof(datamemory));
```

Then, the pointer to this `datamemory` instance can be passed to the callback function by using the `CURLOPT_WRITEDATA` option of `curl_easy_setopt` by,

```
curl_easy_setopt(easy_handle, CURLOPT_WRITEDATA, chunk);
```

The callback function `writecb(void *buffer, size_t size, size_t nmemb, void *user_p)` contains the following arguments,

- `buffer` is the data that we have received as a response from the server
- `size` is the data size of the chunks (not the same as the `chunk` pointer that we have discussed) in the buffer. This value is used because for most transfers, this callback gets called many times and each invoke delivers another chunk of data in the buffer.
- `nmemb` is the number of chunks in the buffer
- `user_p` is the `chunk` pointer that we have passed to this callback function by the `curl_easy_setopt` function

Now, we are going to mimic the process of directly print the contents as the output. What's different is that we will get a copy of the content in the `datamemory` instance that can be used for us in the future in the main function.

First, the real size of the content should be calculated by multiplying the `size` and `nmemb`. The whole size of the output will be assigned to a `realsize` variable,

```
size_t realsize = size * nmemb;
```

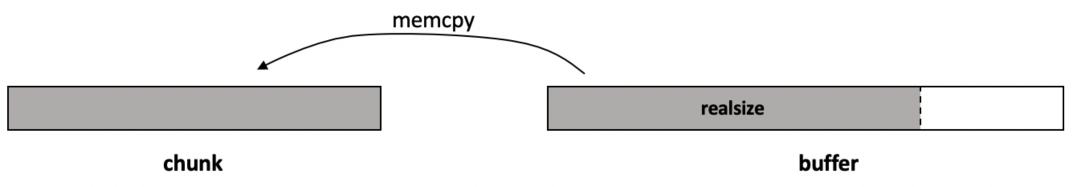
To use the `user_p` pointer as the `chunk` pointer, we have to convert its data type from `void *` to `datamemory *`. The memory allocated to the `response` variable should be reallocated if we want to store the data in the `buffer` to it. The newly assigned memory should have a size of `realsize + 1` so that it can be able to restore the entire data we need.

```
datamemory *chunk = (datamemory *) user_p;
chunk->response = realloc(chunk->response, chunk->size + realsize + 1);
```

Note that we have to check if there is enough memory for us to allocate when we use the `realloc` function,

```
if (chunk->response == NULL) {
    printf("Out of memory!");
    return 0;
}
```

After this procedure, we have a well-sized `datamemory` instance that can be used for us to store the data in the `buffer`. `memcpy` will then be called to copy the memory content in the buffer to the chunk.



```
memcpy(&(chunk->response[chunk->size]), buffer, realsize);
```

Then we can update the `size` element in our `datamemory` instance with the `realsize`.

```
chunk->size += realsize;
```

To directly print the data now in the `datamemory` instance, we can use,

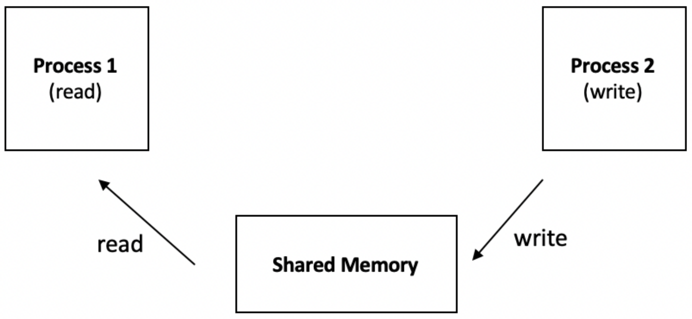
```
printf("%s", chunk->response);
```

After we conduct `curl_global_cleanup`, because we have saved the data in the `datamemory` instance, we can still print the data to the output. This can be checked by adding the following codes to the end,

```
printf("\nCheck the existence of the data =================== \n");
char data[256];
strncpy(data, chunk->response, 100);
data[100] = '\0';
printf("%s\n", data);
printf("end ================================================= \n");
```

2. Part 2 - IPC Synchronizations

The writer process will first write to the shared memory and then it will be detached from the memory. The reader process will then read from the shared memory. After it prints the data in the shared memory, it will be detached from the shared memory. In the end, the shared memory will then be destroyed through `shmctl` by the reader.



What is different in this case is that, because this is not a private IPC anymore, we have to assign a unique key to this shared memory so that the other process can know where to find this shared memory. Thus, the unique key is relatively important in this case. We will use `ftok` to generate the unique key. Note that the first argument of this function must be the path of an existing file, but the file content is not relevant for us. Also, the second argument is also not relevant.

But this implementation will end up with a race condition between the reading process and the writting process. Both the process 1 and process 2 will not wait for each other, and they just fread from the segment blindly. The reading process can read a same content because the writing process hasn't got enough time to write, and also the writing process can overwrite the content in the segment even through the reading process hasn't read from it. This is the reason why we need **semaphores**.

To get a write-read-write pattern, we have to first initialize the semaphores somewhere. The read semaphore should be set to 0 at the beginning and the write semaphore should be set to 1 because we have to write first. The they will looply wait&post between each other to perform a proper transformation.

```
if (sem_init(&(shm_address->semaphore_r), 1, 0) < 0) {
  perror("Initializing semaphore ERROR");
  exit(0);
};     // initialize semaphore
if (sem_init(&(shm_address->semaphore_w), 1, 1) < 0) {
  perror("Initializing semaphore ERROR");
  exit(0);
};     // initialize semaphore
```

Also, for proper cleanups, we also have to know something about the signals. Most Linux users use the key combination Ctrl+C to terminate processes in Linux. The reason behind this is that whenever Ctrl+C is pressed, a signal `SIGINT` is sent to the process. The **default action** of this signal is to terminate the process.

Now, let's add another print. Before the process terminates, we will have to print `Received SIGINT, quitting …` , this means we must modify the signal handler to some user-defined functions. In fact, the handler can be replaced by the `signal` function and the second argument will become a callback function of the present signal.

Because the cache maintains the message queues and the proxy maintains the segment pools, they should do different cleanups in each of the function seperately.

```
while (!steque_isempty(seg_queue)) {
  segment_t *segment = (segment_t *) steque_pop(seg_queue);
  shmctl(segment->shmid, IPC_RMID, NULL);
  free(segment);
}
```

Or,

```
msgctl(msgRqstId, IPC_RMID, NULL);
msgctl(msgRespId, IPC_RMID, NULL);
```

Testings

The following tests can be useful to testing our code locally,

```
// 1. Simple test
./webproxy
./simplecached
./gfclient_download -p 30605

// 2. Test for two threads
./webproxy -t 2
./simplecached -t 2
./gfclient_download -p 30605 -t 2

// 3. Test for fewer segments
./webproxy -t 2 -n 2
./simplecached -t 2
./gfclient_download -p 30605 -t 2

// 4. Test for many segments
./webproxy -t 2 -n 80
./simplecached -t 2
```

```
./gfclient_download -p 30605 -t 2

// 5. Test for a different segment size
./webproxy -t 2 -n 2 -z 4096
./simplecached -t 2
./gfclient_download -p 30605 -t 2

// 6. Test for a different segment size
./webproxy -t 2 -n 2 -z 65536
./simplecached -t 2
./gfclient_download -p 30605 -t 2

// 7. Test for more requests
./webproxy -t 2 -n 2 -z 65536
./simplecached -t 2
./gfclient_download -p 30605 -t 2 -r 16

// 8. Test for More requests
./webproxy -t 2 -n 2 -z 65536
./simplecached -t 2
./gfclient_download -p 30605 -t 2 -r 100

// 9. Test for MORE requests
./webproxy -t 2 -n 2 -z 65536
./simplecached -t 2
./gfclient_download -p 30605 -t 2 -r 1000

// 10. Test for More threads
./webproxy -t 8 -n 2 -z 65536
./simplecached -t 8
./gfclient_download -p 30605 -t 8 -r 16

// 11. Standard test used by the Grade Scope (Fewer threads)
./webproxy -s localhost:8200 -t 2 -z 65536 -n 2 -p 8803
./simplecached -t 2
./gfclient_download -s localhost -p 8803 -t 2 -r 16

// 12. Standard test used by the Grade Scope (More threads)
./webproxy -s localhost:8200 -t 8 -z 4096 -n 8 -p 8803
./simplecached -t 8
./gfclient_download -s localhost -p 8803 -t 8 -r 16
```

Debuggers

1. The futex facility returned an unexpected error code, cached_returncode -6

Reason: semaphores overwrote with unexpected values

Debugger: Found out where you have overwritten the semaphores. If you are using the shared memory to passing the semaphores, this is probably because the size of the segment.

2. Glitch Pictures

Reason: pontentially problem from the shared memory size

Debugger #1: Check for if you have this problem with -z 4096 or -z 65536. These are the values of n times the page size of the Linux (4KB).

Debugger #2: If you set the shared memory size to to size you used for file transfer, you can not include other things in the shared memory. I would suggest using a segment with the size of,

```
segsize + size of other data types in the shm + 1 page size
```

3. Client Hung

Reason: there can be several reasons for this

Debugger #1: check if there is a dead lock

Debugger #2: check if you have reinitialize semaphores in the hander. The semaphores should be initialized only once. If it is reinitialized, it can have unexpected values.

Debugger #3: probably too slow

Debugger #4: I think there can be more reasons for this ...

4. Exception: System V shared memory leak detected (start = [], end=['32769'])

Reason: some allocated segments are not freed properly

Debugger: find the segment that is not freed in the end

References

1. ftok [https://pubs.opengroup.org/onlinepubs/009696699/functions/ftok.html]
2. Key_t type [https://www.unix.com/unix-for-dummies-questions-and-answers/147863-key_t-type.html]
3. Sem_init [https://man7.org/linux/man-pages/man3/sem_init.3.html]
4. Sem_destory [https://man7.org/linux/man-pages/man3/sem_destroy.3.html]
5. shmctl [https://stackoverflow.com/questions/15601683/deleting-shared-memory-segment-with-shmctl]
6. FCM Messages [https://firebase.google.com/docs/cloud-messaging/concept-options]
7. Named pipe [https://www.geeksforgeeks.org/named-pipe-fifo-example-c-program/]
8. Futex [https://stackoverflow.com/questions/48714083/the-futex-facility-returned-an-unexpected-error-code-and-aborted]
9. IPC Overview [https://man7.org/conf/lca2013/IPC_Overview-LCA-2013-printable.pdf]
10. Shm synchronization [https://stackoverflow.com/questions/23926871/shared-memory-synchronisation-between-three-processes-in-linux]