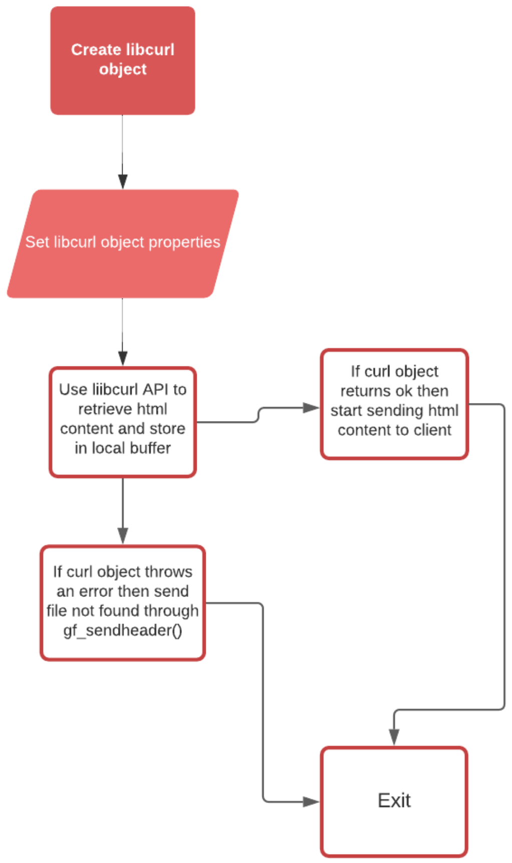
Vishu Gupta

November 09, 2021

Project 3 Readme

Part 1:

Design:

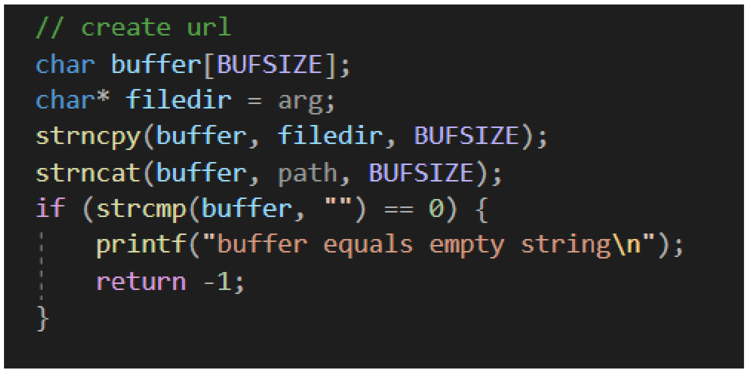

Create libcurl object
Set libcurl object properties
Use liibcurl API to retrieve html content and store in local buffer
If curl object returns ok then start sending html content to client
If curl object throws an error then send file not found through gf_sendheader()
Exit

Objective:

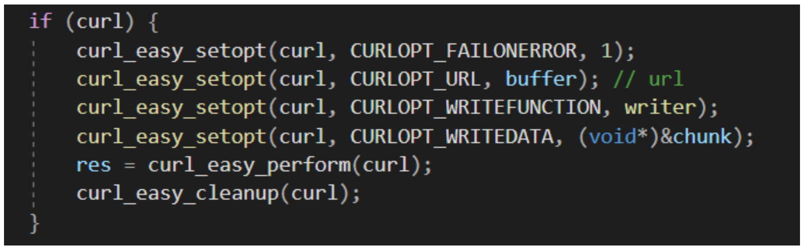
The objective of this part of the project was to retrieve a file from the remote server and use the libcurl library to send the data to the client based on the client's request.

Summary:

To meet this objective my handle_with_curl function stores the request path into a local buffer as shown in the following figure:


```
// create url
char buffer[BUFSIZE];
char* filedir = arg;
strncpy(buffer, filedir, BUFSIZE);
strncat(buffer, path, BUFSIZE);
if (strcmp(buffer, "") == 0) {
    printf("buffer equals empty string\n");
    return -1;
}
```

From there we initialize the curl object using curl_east_setopt(). Some of the options we are setting are shown in the picture below. We specify the url, the writer function and where to store the data from the specified url.


```
if (curl) {
    curl_easy_setopt(curl, CURLOPT_FAILONERROR, 1);
    curl_easy_setopt(curl, CURLOPT_URL, buffer); // url
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, writer);
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, (void*)&chunk);
    res = curl_easy_perform(curl);
    curl_easy_cleanup(curl);
}
```

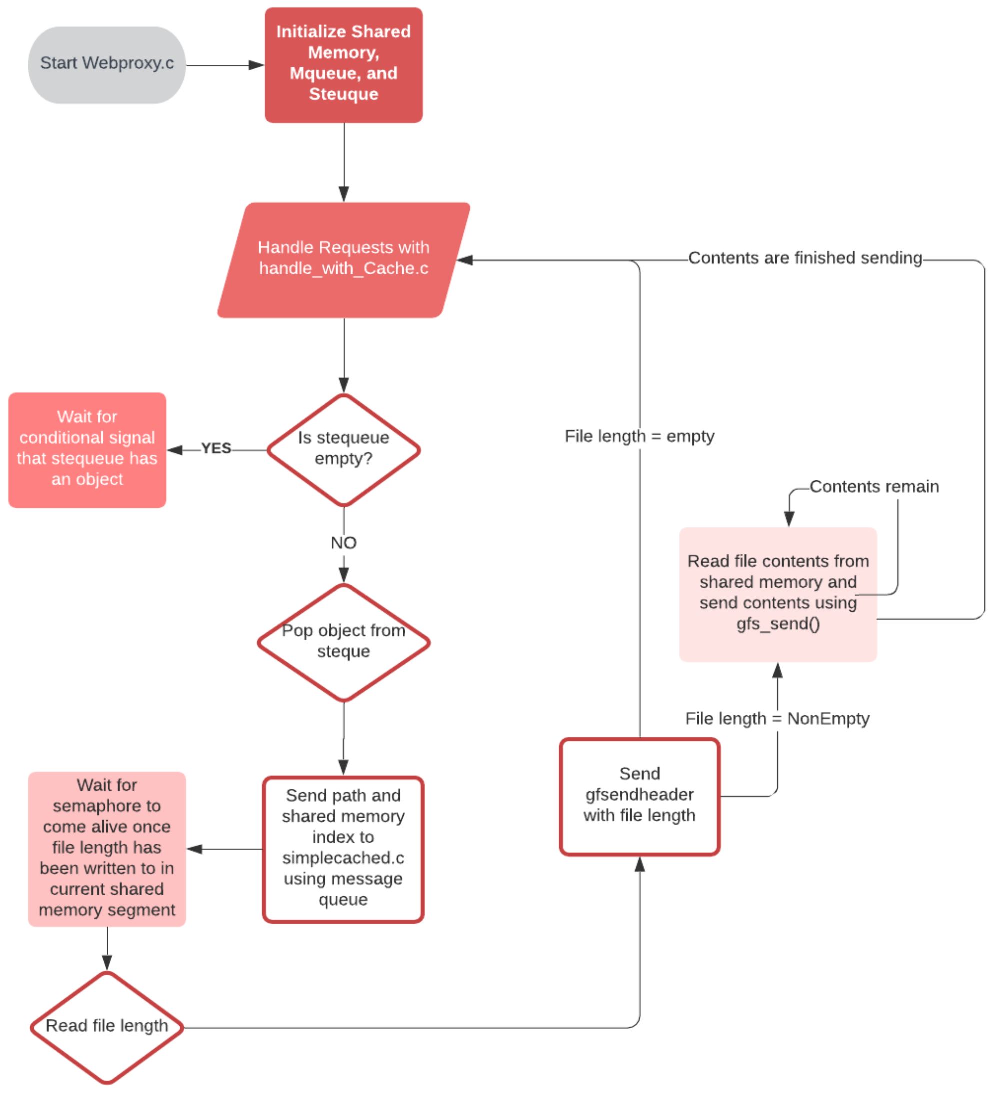
We can then check if we get an error code from writing the request to a locally malloced struct that carries the html content. If we get an error we can send a file not found through gf_sendheader(). If we are successful at writing the html content to the local struct then we can start sending the file content to the client using gf_send().

<u>Part 2:</u>

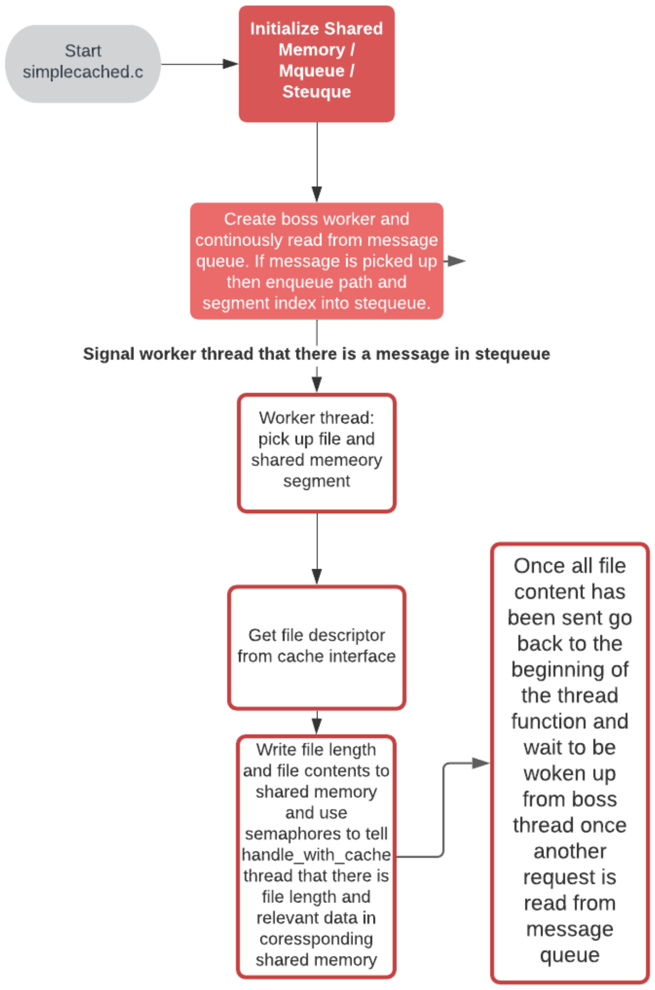
Objective:

The objective for this part of the project was to have two different processes communicate over a shared memory region. The proxy process receives requests from the clients and sends the requested path to the cache process through shared memory. The cache then picks up the path request and uses the cache API to get the file descriptor and send the contents of the file to the proxy via the shared memory region.

Design of webproxy.c/handle_with_cache.c:


Start Webproxy.c
Initialize Shared Memory, Mqueue, and Steuque
Handle Requests with handle_with_Cache.c
Is stequeue empty?
YES
Wait for conditional signal that stequeue has an object
NO
Pop object from steque
Send path and shared memory index to simplecached.c using message queue
Wait for semaphore to come alive once file length has been written to in current shared memory segment
Read file length
Send gfsendheader with file length
File length = empty
File length = NonEmpty
Read file contents from shared memory and send contents using gfs_send()
Contents remain
Contents are finished sending

Design of simplecached.c:


Start simplecached.c
Initialize Shared Memory / Mqueue / Steuque
Create boss worker and continously read from message queue. If message is picked up then enqueue path and segment index into stequeue.
Signal worker thread that there is a message in stequeue
Worker thread: pick up file and shared memeory segment
Get file descriptor from cache interface
Write file length and file contents to shared memory and use semaphores to tell handle_with_cache thread that there is file length and relevant data in coressponding shared memory
Once all file content has been sent go back to the beginning of the thread function and wait to be woken up from boss thread once another request is read from message queue

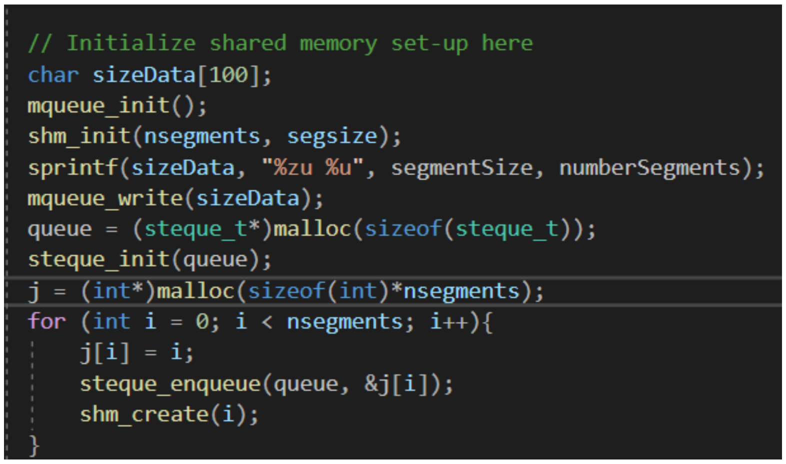
Summary:

To implement and meet this objective I had two different shared memories. I had n shared memory segments and a message queue. I initialized my shared memory segments and message queue in webproxy.c. The shared memory segments were used to write relevant file data including file length and file contents. My shared memory segment consisted of a struct that included a file length variable, a char array to store the file contents, and three semaphores for synchronization between proxy and cache. My message queue consisted of just a char array that sent the file path received from the client to the cache. My message queue was the command channel and my shared memory segments were the data channel.

<u>Webpoxy.c</u> – this file initialized my shared memory segments, my message queue, and a stequeue that would be used by my handle_with_cache functions to pop the shared memory segment that would be

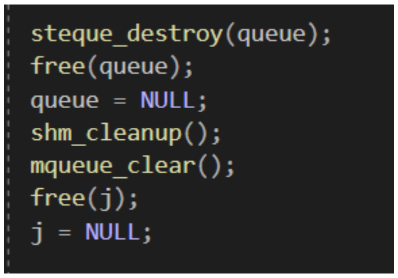
used. This file also used the message queue to send the number of segments and segment size to the cache. This file was also responsible for destroying any shared memory segments as well as the message queue.

This image displays the initialization done in webproxy.c. It is initializing the shared memory, the message queue, as well as a local stequeue to enqueue the shared memory indexes that will be used by handle_with_cache to open up a corresponding shared memory segment.



```c
// Initialize shared memory set-up here
char sizeData[100];
mqueue_init();
shm_init(nsegments, segsize);
sprintf(sizeData, "%zu %u", segmentSize, numberSegments);
mqueue_write(sizeData);
queue = (steque_t*)malloc(sizeof(steque_t));
steque_init(queue);
j = (int*)malloc(sizeof(int)*nsegments);
for (int i = 0; i < nsegments; i++){
    j[i] = i;
    steque_enqueue(queue, &j[i]);
    shm_create(i);
}
```

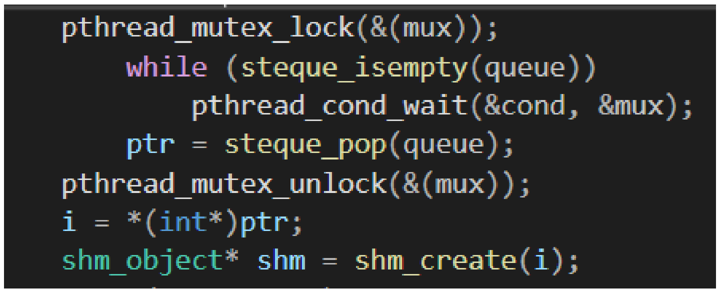
This image displays the destruction of the shared memory, the message queue, as well as the local stequeue used by webproxy.c. This is also used in my sigkill handler.



```c
steque_destroy(queue);
free(queue);
queue = NULL;
shm_cleanup();
mqueue_clear();
free(j);
j = NULL;
```

<u>Handle_with_cache.c</u> – this file popped the shared memory segment available in stequeue and used this shared memory segment to retrieve data from cache. This function was responsible for sending the message to cache, and then retrieving the file data. If the file length was zero then the corresponding handle_with_cache thread would send the header to client and enqueue the corresponding shared memory segment back into steuque. If the file length was not zero then the thread would send the header followed by the file contents in chunks. Once all the chunks were sent then the corresponding thread would enqueue the shared memory back into stequeue.

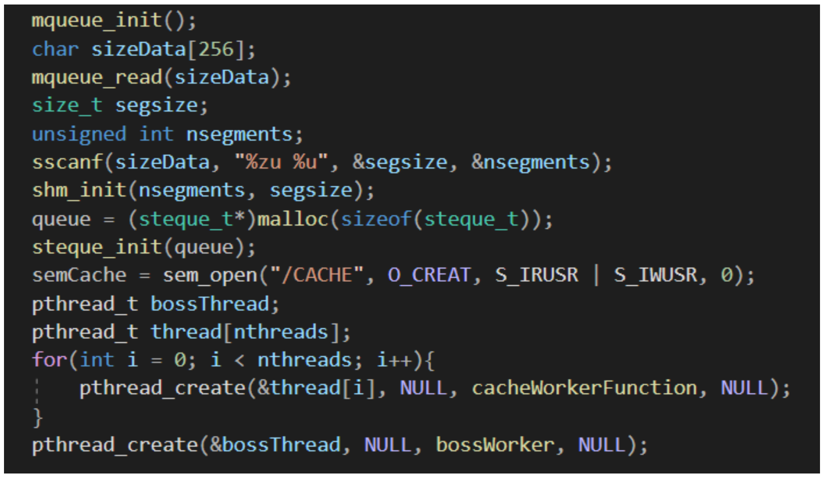
This image displays how handle_with_cache retrieves an available shared memory segment by popping it from stequeue. From there it will send this shared memory segment to simplecached.c by sending the appropriate integer index and path as well as handling receiving of the data from the shared memory region using semaphores.



```
pthread_mutex_lock(&(mux));
    while (steque_isempty(queue))
        pthread_cond_wait(&cond, &mux);
    ptr = steque_pop(queue);
pthread_mutex_unlock(&(mux));
i = *(int*)ptr;
shm_object* shm = shm_create(i);
```

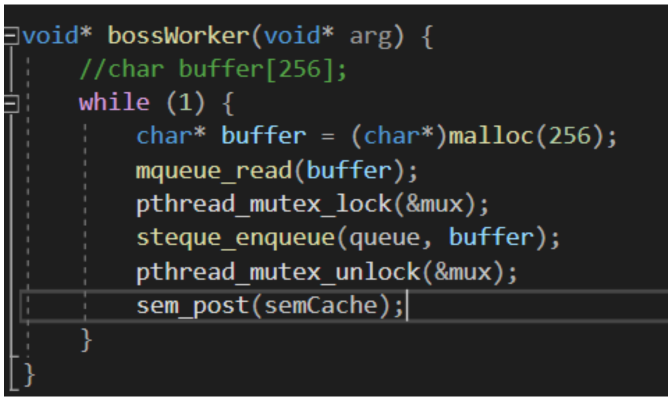
Simplecached.c – this file was a boss worker thread pattern that handled the requests sent from the proxy side. This file opened the message queue to receive the messages. The boss would receive the message then enqueue the path and shared memory segment into its own stequeue. The boss thread would then signal one of the worker threads that there is something in stequeue through a semaphore. One of the worker threads would wake up and handle this message request appropriately by writing to the same shared memory that the handle_with_cache function that sent the request points to. This worker thread would write the file length, and file contents to the shared memory and use the shared memory semaphores to synchronize the transmission of data.

This is an image that shows the initialization done on the simplecached.c side. The first message the simplecached.c receives is the number of shared memory segments as well as the segment size. It then uses these values locally.



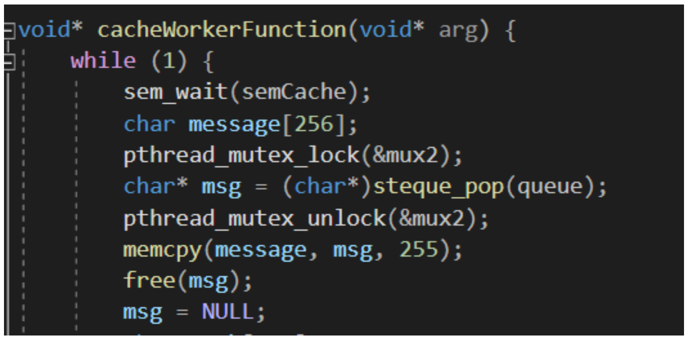
```
mqueue_init();
char sizeData[256];
mqueue_read(sizeData);
size_t segsize;
unsigned int nsegments;
sscanf(sizeData, "%zu %u", &segsize, &nsegments);
shm_init(nsegments, segsize);
queue = (steque_t*)malloc(sizeof(steque_t));
steque_init(queue);
semCache = sem_open("/CACHE", O_CREAT, S_IRUSR | S_IWUSR, 0);
pthread_t bossThread;
pthread_t thread[nthreads];
for(int i = 0; i < nthreads; i++){
    pthread_create(&thread[i], NULL, cacheWorkerFunction, NULL);
}
pthread_create(&bossThread, NULL, bossWorker, NULL);
```

This is an image that shows the boss thread in simplecached.c. It reads from the message queue and once a message is received it enqueues the message to a local stequeue and signals the worker threads that there is some work to be done.



```c
void* bossWorker(void* arg) {
    //char buffer[256];
    while (1) {
        char* buffer = (char*)malloc(256);
        mqueue_read(buffer);
        pthread_mutex_lock(&mux);
        steque_enqueue(queue, buffer);
        pthread_mutex_unlock(&mux);
        sem_post(semCache);
    }
}
```

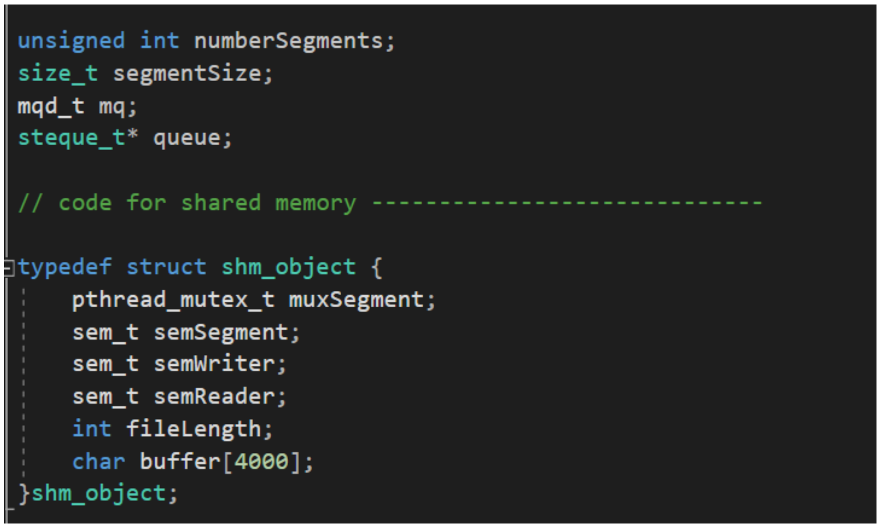
This is an image that shows my worker thread waiting to be woken up from the boss thread using a local semaphore.



```c
void* cacheWorkerFunction(void* arg) {
    while (1) {
        sem_wait(semCache);
        char message[256];
        pthread_mutex_lock(&mux2);
        char* msg = (char*)steque_pop(queue);
        pthread_mutex_unlock(&mux2);
        memcpy(message, msg, 255);
        free(msg);
        msg = NULL;
```

Shm_channel.h – this file was used to instantiate the shared memory struct, helper functions, and whatever else that was needed to be utilized by both the cache and proxy.

This image shows the shared memory struct defined in shm_channel.h



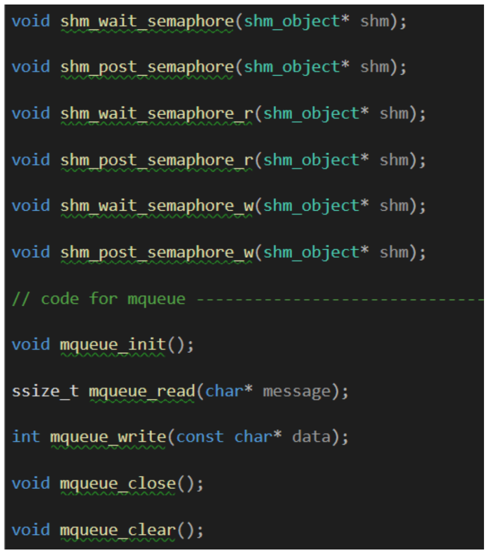
```c
unsigned int numberSegments;
size_t segmentSize;
mqd_t mq;
steque_t* queue;

// code for shared memory ----------------------------

typedef struct shm_object {
    pthread_mutex_t muxSegment;
    sem_t semSegment;
    sem_t semWriter;
    sem_t semReader;
    int fileLength;
    char buffer[4000];
}shm_object;
```

Shm_channel.c – this file was used to create helper functions to allow me to write and read to shared memory as well as providing initialization and destroying functions. This file was shared with both the proxy and cache side.

This image shows some of the functions I have created in shm_channel.h/c to be used in both webproxy.c, handle_with_cache.c, and simplecached.c



```
void shm_wait_semaphore(shm_object* shm);

void shm_post_semaphore(shm_object* shm);

void shm_wait_semaphore_r(shm_object* shm);

void shm_post_semaphore_r(shm_object* shm);

void shm_wait_semaphore_w(shm_object* shm);

void shm_post_semaphore_w(shm_object* shm);

// code for mqueue ----------------------------

void mqueue_init();

ssize_t mqueue_read(char* message);

int mqueue_write(const char* data);

void mqueue_close();

void mqueue_clear();
```

References:

https://man7.org/linux/man-pages/man3/sem_wait.3.html
https://man7.org/linux/man-pages/man2/mmap.2.html
https://linux.die.net/man/3/pthread_mutex_destroy#:~:text=Destroying%20Mutexes&text=Implementations%20are%20required%20to%20allow,is%20unlocked%20(line%20C).
https://man7.org/linux/man-pages/man3/sem_unlink.3.html
https://man7.org/linux/man-pages/man3/sem_open.3.html
https://man7.org/linux/man-pages/man2/pread.2.html
https://man7.org/linux/man-pages/man3/sem_init.3.html
https://man7.org/linux/man-pages/man3/sleep.3.html
https://curl.se/libcurl/c/
Class lectures