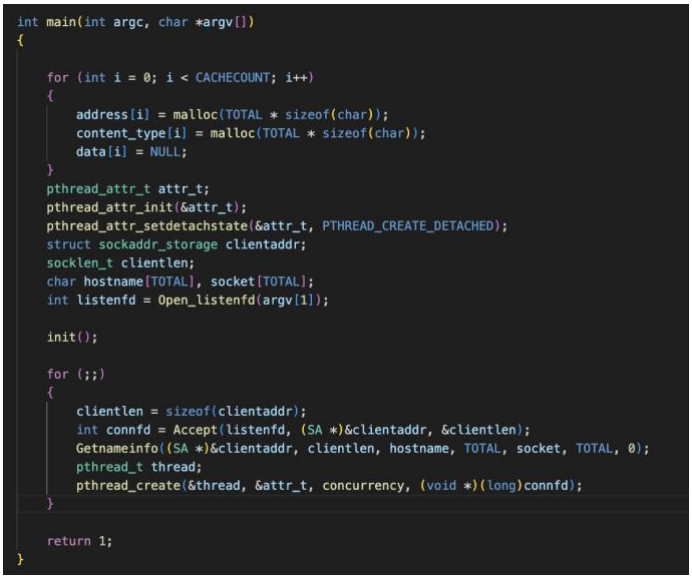
System Programming Report- Proxy Labe
Name: Taehyun Yang
2021-14284

1. Main purpose of this lab

The main purpose of this lab was to create a proxy, an intermediate connector between a client and the origin server. We had to implement connecting the client's request to the proxy, while receiving proxy responses to the client and vice versa for the server. Now in our proxy, the main purpose was to create a cachable proxy that takes web objects from any website that supports files such as images, styleships, scripts, and any HTML headers. We also had to implement the concurrency of this which is checked in my concurrency method using mutexes. Now we cached in by taking each object as a separate file in order to implement a sequential web proxy that does the basic operations of setting up the proxy to accept incoming connections, reading and parsing requests, forwarding requests to web servers, reading server's responses as well as forwarding those responses to corresponding clients.

The more advanced part of this section was to deal with multiple concurrent requests. We used multithreading and the uses of mutexes to cleverly implement the concurrency of these proxy requests. Also, the hardest part of all was to cache web objects. We had to add caching to our proxy using a simple main memory cache of recently accessed web content by caching individual objects and not the whole page. We used LRU eviction policy for this and allowed concurrent reads while maintaining the overall integrity of the website objects.

2. Implementation



```
int main(int argc, char *argv[])
{

    for (int i = 0; i < CACHECOUNT; i++)
    {
        address[i] = malloc(TOTAL * sizeof(char));
        content_type[i] = malloc(TOTAL * sizeof(char));
        data[i] = NULL;
    }
    pthread_attr_t attr_t;
    pthread_attr_init(&attr_t);
    pthread_attr_setdetachstate(&attr_t, PTHREAD_CREATE_DETACHED);
    struct sockaddr_storage clientaddr;
    socklen_t clientlen;
    char hostname[TOTAL], socket[TOTAL];
    int listenfd = Open_listenfd(argv[1]);

    init();

    for (;;)
    {
        clientlen = sizeof(clientaddr);
        int connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        Getnameinfo((SA *)&clientaddr, clientlen, hostname, TOTAL, socket, TOTAL, 0);
        pthread_t thread;
        pthread_create(&thread, &attr_t, concurrency, (void *)(long)connfd);
    }

    return 1;
}
```

Main

The main function of your program serves as the initialization and control center for a multithreaded proxy server. It begins by setting up the cache for the proxy server, where it allocates memory for different cache entries. These entries include the addresses (address[i]), content types (content_type[i]), and data (data[i]).

The function then creates an attribute variable attr_t for thread creation, using the pthread_attr_init functions. The PTHREAD_CREATE_DETACHED state allows threads to be immediately cleaned up once they finish execution. After the initial setup, the function establishes a listening socket with the Open_listenfd function, using the port number passed in through argv[1]. The init function, likely used to initialize some global variables or data structures, is then called. Then, for each client connection, a new thread is created using pthread_create. The concurrency function is passed as the start routine for the newly created thread, and the file descriptor connfd from the Accept function is passed as an argument to the concurrency function.

```
void init()
{
    for (int i = 0; i < CACHECOUNT; ++i)
    {
        if (data[i])
        {
            free(data[i]);
            data[i] = NULL;
        }

        size[i] = 0;
        cache_lru[i] = CACHECOUNT - i;
    }
}
```

INIT

in the init function, we're setting the stage for our cache system. It iterates over a predefined range, which is up to CACHECOUNT. For each iteration, it checks if there's already some data present in the data[i] position. If it finds something, it frees up that memory, essentially clearing the cache.

```
void *concurrency(void *thread)
{
    int clientfd = (int)(long)thread;
    char buff[TOTAL], URL[TOTAL], cache_content_type[TOTAL];
    char header_names[HEADERCOUNT][TOTAL], header_values[HEADERCOUNT][TOTAL];
    target targetServer;
    rio_t clientRio;
    int serverfd;
    char *cacheData = NULL;
    int cacheSize;

    // Initialize clientRio and read the first line of the request
    Rio_readinitb(&clientRio, clientfd);
    if (!Rio_readlineb(&clientRio, buff, TOTAL))
        return NULL;
    sscanf(buff, "%s %s %s", targetServer.httpslisten, URL, targetServer.httpstype);

    // Try to find the requested content in cache
    pthread_mutex_lock(&mutex);
    int cacheIndex = cache_read(URL);

    if (cacheIndex != -1) // If cache hit, prepare to send data from cache
    {
        cacheSize = size[cacheIndex];
        cacheData = malloc(cacheSize);
        memcpy(cacheData, data[cacheIndex], cacheSize);
        strncpy(cache_content_type, content_type[cacheIndex], TOTAL);
    }
    pthread_mutex_unlock(&mutex);
```

```
    if (cacheData) // If data was found in cache
    {
        sendResponseFromCache(clientfd, cacheData, cacheSize, cache_content_type);
        free(cacheData);
    }
    else // Data not found in cache, retrieve from server
    {
        int header_num = header_read(&clientRio, header_names, header_values); // Read the remaining headers
        url_convert(URL, &targetServer);
        Header_insert(header_names, header_values, &header_num, "User-Agent", user_agent_hdr);
        Header_insert(header_names, header_values, &header_num, "Host", targetServer.address);
        Header_insert(header_names, header_values, &header_num, "Connection", "close");
        Header_insert(header_names, header_values, &header_num, "Proxy-Connection", "close");
        serverfd = Open_clientfd(targetServer.address, targetServer.socket);

        // Send request to remote server and forward the response to client
        connecttoServer(clientfd, serverfd, &targetServer, header_names, header_values, header_num);
        datacontext(clientfd, serverfd, URL);
    }

    // Clean up
    close(serverfd);
    close(clientfd);

    return;
}
```

CONCURRENCY

This concurrency function essentially manages the interactions between the client, the cache, and the server. It is designed to operate in a separate thread for each client connection. The variable clientfd holds the client's file descriptor. This descriptor is crucial as it enables communication with the client. On the other hand, serverfd is used to establish and maintain communication with the server. The function starts by reading from the clientfd to obtain the client's request. The data is stored in a buffer and the requested URL is extracted for further processing.

Once we have the requested URL, a mutex lock is initiated using pthread_mutex_lock. This action is essential to prevent other threads from accessing the shared cache while the current thread checks for the requested URL and potentially updates the cache. If the URL is found in the cache (cache hit), the cached data along with its size are copied to local variables. Once the operations related to cached data are completed, the mutex lock is released using pthread_mutex_unlock, which allows other threads to access the shared cache. If the data was found in the cache, the function proceeds to send the response back to the client using clientfd. After the data is sent, the locally allocated memory for cacheData is freed.

However, if the data wasn't in the cache, the function initiates interaction with the server. It starts by reading the remaining headers from the client's request. Then, it prepares for the connection with the server by converting the URL, inserting necessary headers, and establishing a connection to the remote server. The server's response is then sent back to the client. Once the processing is done, the function closes the connections with both the client and the server by using close on clientfd and serverfd respectively. It's important to close these connections to free up system resources.

```c
void sendResponseFromCache(int clientfd, char *cacheData, int cacheSize, char *cache_content_type)
{
    char buff[TOTAL];

    // Format http header
    int len = snprintf(buff, TOTAL,
                       "HTTP/1.0 200 OK\r\n"
                       "Server: Tiny Proxy Server\r\n"
                       "Content-type: %s\r\n" empty,
                       cache_content_type);

    // Write HTTP header to client
    Rio_writen(clientfd, buff, len);

    // Send main content to the client
    Rio_writen(clientfd, cacheData, cacheSize);
}
```

sendResponseFromCache is a function that serves client requests directly from the cache when there's a cache hit. The function first establishes a buffer, buff, which is designed to store the HTTP response header. It uses snprintf to systematically format this HTTP header. The header comprises several elements including the HTTP version, the status code (200) some server

```c
int header_read(rio_t *rp, char header_names[HEADERCOUNT][TOTAL], char header_values[HEADERCOUNT][TOTAL])
{
    char buff[TOTAL];
    int headers_readcount = 0;

    // Read the first line of header
    Rio_readlineb(rp, buff, TOTAL);

    // Loop until a line with only empty is encountered
    while (strcmp(buff, empty))
    {
        // Parse header line into name and value
        sscanf(buff, "%[^:]:%s\r\n", header_names[headers_readcount], header_values[headers_readcount]);

        // Increment the headers_readcount
        headers_readcount++;

        // Read the next line
        Rio_readlineb(rp, buff, TOTAL);
    }

    // Return the count of headers read
    return headers_readcount;
}
```

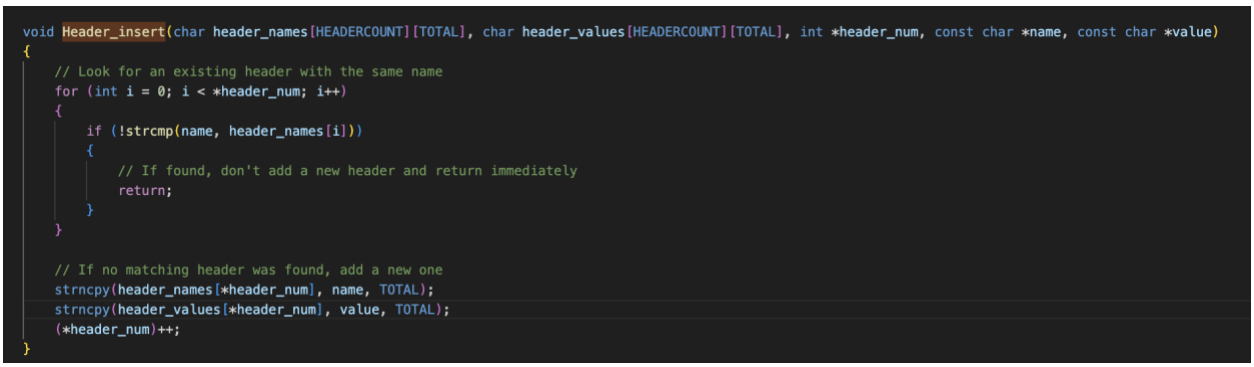
information, and the critical Content-Type field. The value for Content-Type is directly fetched from the cache_content_type variable we got from our cache.

Header_read

The header_read takes three parameters: a rio_t type, which represents a buffered input/output stream, and two two-dimensional arrays to hold the names and values of the

headers respectively, in order to parse the http headers from a request or a response. The function initiates a loop to read each line from the buffered I/O, where each line represents a single HTTP header. It stops when it encounters a blank line, indicating the end of the HTTP headers. Inside the loop, it uses the sscanf function to parse each line, dividing it into a name and a value, and storing them in the arrays header_names and header_values, respectively. Each time it successfully reads and parses a header, it increments the headers_readcount variable, which keeps track of the total number of headers read.

Upon exiting the loop, the function returns headers_readcount, resulting in the total number of HTTP headers that were parsed. This count could be useful in subsequent operations that need to iterate through all the headers.



```
void Header_insert(char header_names[HEADERCOUNT][TOTAL], char header_values[HEADERCOUNT][TOTAL], int *header_num, const char *name, const char *value)
{
    // Look for an existing header with the same name
    for (int i = 0; i < *header_num; i++)
    {
        if (!strcmp(name, header_names[i]))
        {
            // If found, don't add a new header and return immediately
            return;
        }
    }

    // If no matching header was found, add a new one
    strncpy(header_names[*header_num], name, TOTAL);
    strncpy(header_values[*header_num], value, TOTAL);
    (*header_num)++;
}
```

Header_insert

The Header_insert function's main function is to add a new HTTP header to the existing collection of headers, which are stored in two two-dimensional arrays: header_names and header_values. This two by two dimensional array essential becomes the container of header files within our proxy code. The function takes 6 arguments, including these arrays, a pointer to the count of headers currently stored (header_num), and a name-value pair that represents the new HTTP header to be added. Header insert then begins by looping through the existing headers in search for a header that has the same name as the new one that is to be added. This is performed by comparing the name parameter with each entry in the header_names array using the strcmp function. If a match is found, the function terminates early, effectively ignoring duplicate headers. If no match is found during this search, the function proceeds to add the new header to the collection. It copies the name and value into the header_names and header_values arrays, respectively, at the position indicated by header_num, using the strncpy function. Subsequently, it increments the value of header_num to reflect the addition of the new header.

This function basically either ensures the uniqueness of headers by name, or adds new ones to the collection as needed.

```c
void connecttoServer(int clientfd, int serverfd,
                     target *targetServer, char header_names[HEADERCOUNT][TOTAL], char header_values[HEADERCOUNT][TOTAL], int header_num)
{

    // Buffer to hold our message
    char buff[3 * TOTAL + 5];

    // Write the first line
    int buff_len = snprintf(buff, TOTAL, "%s %s %s\r\n",
                            targetServer->httpslisten,
                            targetServer->content,
                            targetServer->httpstype);

    Rio_writen(serverfd, buff, buff_len);

    // Write http headers
    for (int i = 0; i < header_num; i++)
    {
        buff_len = snprintf(buff, sizeof(buff), "%s: %s\r\n",
                            header_names[i],
                            header_values[i]);

        Rio_writen(serverfd, buff, buff_len);
    }

    // End of headers
    Rio_writen(serverfd, empty, 2);
}
```

Connecttoserver

The function connecttoServer is used to handle the sending of HTTP requests to a target server. This function accepts six parameters: clientfd, serverfd, a pointer to a target server structure (targetServer), two arrays storing header names and values (header_names and header_values), and the number of headers (header_num). The process begins to connect to server by preparing a buffer to hold the HTTP request message. The first line of this message which is the request line is constructed using the snprintf function. This line includes the HTTP method like get and post and is used to the requested resource's URI, and the HTTP version, all of which are obtained from the targetServer structure. Once the request line is built, it is sent to the target server using the Rio_writen function. Then the function proceeds to handle the HTTP headers. It loops through the header names and values, again using snprintf to construct each header line in the format "header-name: header-value". These header lines are then sent to the target server one at a time using Rio_writen.

Finally when the headers have been sent, the function sends an additional line containing only a carriage return and line feed ("\r\n"), signifying the end of the HTTP headers according to the HTTP protocol.

```c
void datacontext(int clientfd, int serverfd, const char *end_Address)
{
    rio_t serverRio;
    Rio_readinitb(&serverRio, serverfd);

    char buff[2 * TOTAL + 5];
    int headerc_size = -1;
    char header_names[HEADERCOUNT][TOTAL], header_values[HEADERCOUNT][TOTAL];
    char *content_type = NULL;

    // Read and send the first line
    readAndSendFirstLine(&serverRio, clientfd, buff);

    // Read and send headers
    readAndSendHeaders(&serverRio, header_names, header_values, clientfd, &headerc_size, &content_type, buff);

    // Read and send body
    readAndSendBody(&serverRio, clientfd, end_Address, headerc_size, content_type);
}
```

Data Context

The function `datacontext` is used to handle the reading and sending of HTTP response messages from the target server to the client. It takes in 3 parameters which are clientfd, serverfd, and end_Address. Here clientfd is the file descriptor for the client-side connection, serverfd is the file descriptor for the server-side connection, and end_Address is the address of the requested content. First, the function initializes a rio_t structure (serverRio) for the server-side connection with `Rio_readinitb`. This makes it easier to handle the reading operations from the server file descriptor.Then, it sets up a buffer and arrays for header names and values, along with variables for the content size and type. The content size is initialized to -1, indicating that it's unknown at the start. The content type is initialized to NULL.

Datacontext then calls `readAndSendFirstLine`, passing the address of serverRio, the client file descriptor, and the buffer. This reads the status line of the HTTP response from the server and sends it to the client.

Then `readAndSendHeaders` is called, which reads the headers of the HTTP response and sends them to the client. It also checks for Content-length and Content-type headers, updating the corresponding variables if they are found. And then `readAndSendBody` is called to read and send the body of the HTTP response. This involves passing the serverRio, the client file descriptor, the end address of the request, the content size, and the content type. This function basically reads an HTTP response from the server and sends it to the client, while also providing some processing of the response headers.

With some helper function, I was able to finalize the implementations and create a working proxy that successfully connects the server and the client.

3. Challenges

The proxy lab was personally one of the hardest lab I had to implement and I had to spend a lot of time thinking about its technicalities. The first challenge I encountered is to understand how networking works and debugging with the curls of nonproxies. Understanding how HTTP and other protocols function and communicate over the internet, as well as structuring HTTP requests and responses correctly was pretty hard due to the complexities when dealing with concepts like HTTP headers and the various roles they play in the communication process. Also, working with multithreaded programming in C was another big obstacle. Implementing concepts like mutexes to avoid race conditions in a concurrent environment was difficult as my brain normally works in a sequential matter but thinking asynchronously to block off shared variables was difficult. Ensuring the program behaves as expected when multiple threads are involved demands meticulous planning and debugging was pretty hard.

Fixing my pointers and memory allocation and deallocation, was also very difficult as I was not dynamically allocating my memories and causing memory leaks or segmentation faults can lead to program crashes or unexpected behavior so I had to spend a lot of time debugging.
Cache management was also pretty hard as implementing cache with efficient search, insertion and deletion, while making sure to maintain the correct cache size according to the given constraints took a lot of time to think about.

Finally, the hardest one of the major hurdles would have been debugging the code. tracking these down within a networked, multithreaded environment can be quite tricky. I had to utilize programs such as gdb for debugging, or valgrind for memory leak detection.

4. Things I have learned

I first had to learn a lot about the technicalities involved in understanding the communication over the internet, especially structuring HTTP requests and responses. They were intricate and required a many searches and references inorder to understand. I was surprising to see how these headers control so many aspects of the communication process, from caching to compression and beyond.

I also learned a lot about multithreaded programming in C, as it required a shift in thinking from sequential to parallel and concurrent processing. I had to wrap my head around shared variables and the potential for race conditions, which can occur in concurrent environments. The introduction and proper application of mutexes as a way to manage these potential issues might have been a surprising yet a learning experience

Another learning moment was when I realized the implications of incorrect memory management in C. It was surprising and challenging to encounter the issues caused by memory leaks or segmentation faults, and how these can result in unexpected behavior or even crashes. This gave me a new level of appreciation for the importance of proper dynamic memory allocation and deallocation.

The management of cache was also something new I have learned. Developing a cache with efficient search, insertion, and deletion operations, all the while ensuring the correct cache size, required a strong understanding of the class materials and a good understanding of the assignment.

Finally, the difficulty of debugging code, especially in a networked, multithreaded environment was when I learned a lot. Tools like gdb and valgrind are not easy to use and they required a lot of reading to understand and required a significant time investment to learn effectively. This challenge however, gave me understanding as I can use this in my programming future

5. Conclusion

Overall, this lab provided me a rigorous yet very interesting lab learning experience. Despite the challenges, each challenge I overcame I felt like I was learning something new and it led me to try different strategies.