ENCE360 ASSIGNMENT

YUHAO ZHANG

95354247

Introduction

This report aims to implement and analyze a minimal HTTP client, a concurrent queue using semaphore, a downloader which to download the file in parts, merge the parts to an intact file and delete these parts. The following sections will explain the requirements are following, algorithm and performance analysis for the **downloader.c,** and the improvements for algorithm and merging file.

Specific Requirements

There are no more than 3 levels of nesting loop and less than 40 lines of code, and most of the codes were commenting.

The **http.c** is to implement an HTTP client which performs an HTTP 1.0 query to a website, and a determination of maximum byte size to request in a partial download. Using **valgrind --leak-check=full** checked all heap blocks were freed and no leaks are possible, there was no error from any context.

The **downloader.c** was tested by **small.txt** and **larger.txt**. The number of parts for a file would download depending on the number of threads (tasks) will use, the merge process combined all parts into an intact file, and the removal process deleted all parts after combination. The downloaded file compared with the file downloaded from **wget.** to make sure the file is correct and using **valgrind --leak-check=full** checked all heap blocks were freed and no leaks are possible, there was no error from any context.

The **queue.c** was tested by **queue_test,** expected result and my results were the same, and the time cost of using CSSE LAB3 within 1 second. Using **valgrind --leak-check=full** checked all heap blocks were freed and no leaks are possible.

Algorithm Analysis

Part 1 - downloader.c

The step by step of algorithm analysis in downloader.c (lines 228 - 264) shows below:

```
// work is to record the number of available tasks
// bytes is the max chunk size
// num_tasks save the times for multiply download a file
int work = 0, bytes = 0, num_tasks = 0;
// looping for get each URL from fp (the file where URLs located),
// and &line is to save specific URL and using for later functions.
// The != -1 is to make sure there is a URL will get.
while ((len = getline(&line, &len, fp)) != -1) {
// Checking "\n" for a line of URL
    if (line[len - 1] == '\n') {
    // And using null byte to replace "\n"
    line[len - 1] = '\0'; }
    // Get number of tasks, which is the times of a file should download
    num_tasks = get_num_tasks(line, num_workers);
    // The maximum chunk size for each task
    bytes = get_max_chunk_size();
    // Loop multiply times until all
    for (int i = 0; i < num_tasks; i ++) {
        // Record all available tasks
        ++work;
        // Put the task to TODO QUEUE for downloading
        // Clarify range for multiply tasks: i * bytes - (i+1) * bytes - 1
        // 0 - 99
        // 100 - 199
        // 200 - 299 ...  to avoid overlap bytes.
        queue_put(context->todo, new_task(line, i * bytes, (i+1) * bytes -1));}
    // Get results back
    while (work > 0) {
        // Decrease the tasks, and be able to let new task to add in.
        --work;
        // Download the context form task which is currently in TODO QUEUE
        wait_task(download_dir, context);}
    // Merge the files -- simple synchronous method
    merge_files(download_dir, line, bytes, num_tasks);
    // Then remove the chunked download files
    remove_chunk_files(download_dir, bytes, num_tasks);
}
// Clean up
fclose(fp); // Close file descriptor
free(line); // Free allocated memory
```

```
free_workers(context);
return 0;
```

Part 2 – Similar algorithm in Lecture

The algorithm is like a Producer-Consumer problem. The main idea is to have a shared area (QUEUE), and if the queue is full, the producer (write threads) is blocked from continuing put data into a queue, if the queue is empty, the consumer (read threads) is blocked continuing to consume data.

Performance Analysis

The results were expected and showed below. The trends for both small.txt and larger.txt were the same. The time cost decreased until reached the lowest point and then increased. Two main processes cost occupied most of the time, one is the download process and the other is the merging process. The test is conducted separately below, which can view the impact of the merging process on overall execution time. Each thread was tested 7 times and averaged.

Part 1 - Testing without merging process

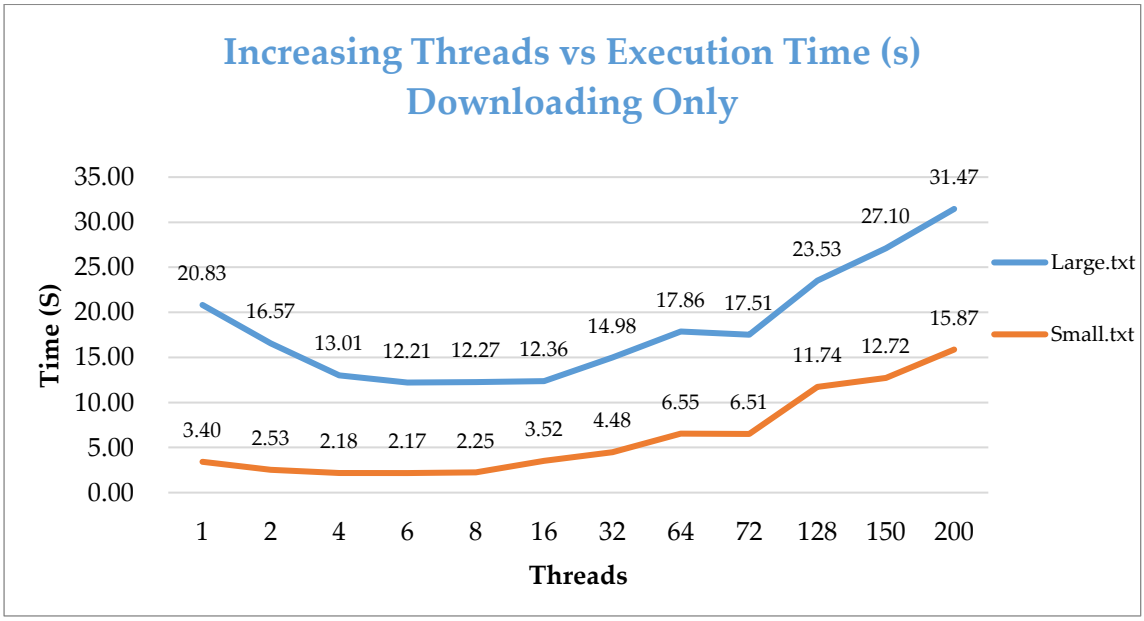

Figure 1: Relationship between increasing threads and time with Downloading process only

According to Figure 1, multithreading had a larger impact on downloading large files. For file downloading, the time cost usually depends on network speed. However, the firewall of downloading server and server data sending speed can limit the transfer of packet data. Multithreading gets multiple streams for downloading data from different servers, hence large

file lift more speed restrictions then small file. Overall, Figure 1 showed execution time was decreasing at the beginning for both large.txt and small.txt.

For the second part, the reasons for increment threads case execution time increasing are I/0, context switching, and mutex of each thread. For multithreading I/O would be re-addressed continuously, thus it loses efficiency for completing the task. In the software part, more threads will cause larger overhead for context switching and mutex to increase the time cost

Part 2 - Testing with Entire program (with merging process)

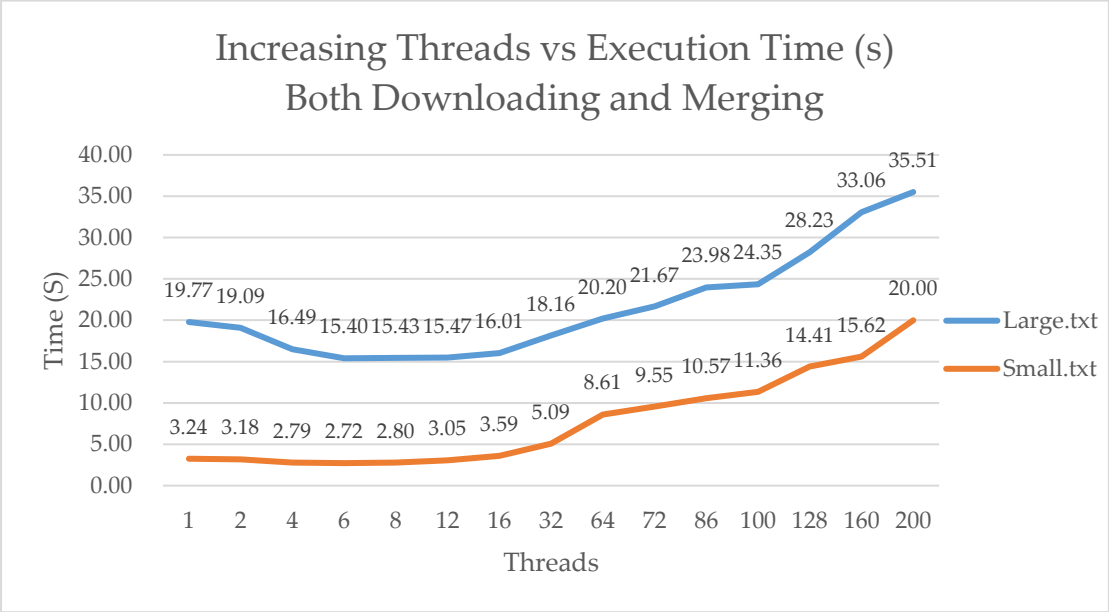

Figure 2: Relationship between increasing threads and time with both downloading and merging process

For entire program, the optimal number of worker threads is around 6, which tested in CSSE computer lab3 with CPU i7-8700K (6 cores 12 MB cache)

Compare with Figure 1, the execution time of Figure 2 is longer, as the number of threads increased. Also, the increasing threads caused the time of merging to become longer. It was expected that larger threads means a larger amount of chunk files and caused a longer time cost for merging them.

Part 3 - Compare with Pre-compiled downloader

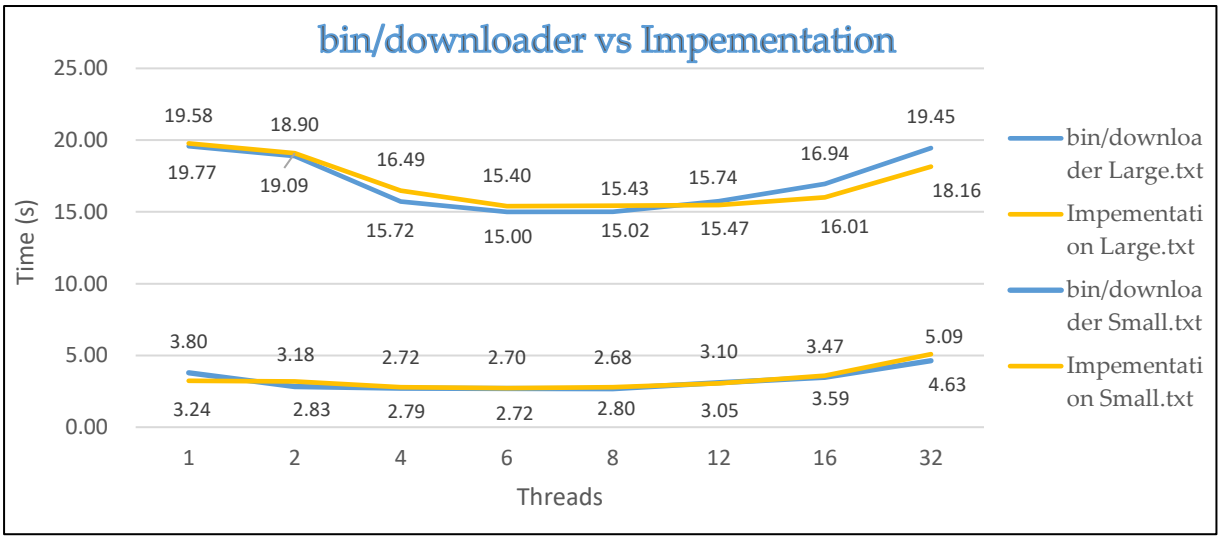

Figure 1: Difference between implemented downloader and pre-compiled downloader

According to Figure 3, the execution time tested by implemented and pre-compiled downloader, the difference between all threads are not more 1.5 seconds.

Improvements

The current way to generate files is inefficient, the cost of merge and delete every chunk is quite high. There is a way to improve this process is to use total content length which can get from the HTTP requests, and use it to pre-allocate space. After that, the range of file is constant and unique, so the part files can directly write into a file. There is no more merging and deleting of chunk files.

Conclusion

For multipart HTTP downloader, appropriate multithreading is an efficiency way to increase the performance. The way for merging and deleting processes need to improve for decreasing execution time. Overall, the entire project was successfully implemented.