

ENCE360 Assignment

Yongdun Yang
42637969

1. Introduction

According to the requirement of assignment, a minimal HTTP client which is socket based and partial downloading supported is implemented. Meanwhile in order to support multi threads concurrency, semaphore is used to implement a concurrent tasks Queue data structure. At last, combine all these tools, a multi threads concurrent supported downloader is implemented.

In the following parts, algorithm and performance analysis are conducted. After that some improvement suggestions and conclusion will be given.

2. Algorithm analysis

2.1 Algorithm of downloader.c (lines 228-264) step by step:

```
/** Initialization:
 * >> work: Initialize number of tasks in the queue need to deal with.
 * >> bytes: Initialize max chunk size.
 * >> num_tasks: Initialize the number of tasks needed satisfying max_chunk_size to download the resource.
 */
int work = 0, bytes = 0, num_tasks = 0;

// Read url from url_file line by line
while ((len = getline(&line, &len, fp)) != -1) {
    // Replace the char of the end of line into null byte
    if (line[len - 1] == '\n') {
        line[len - 1] = '\0';
    }

    /** Calculate max_chunk_size and Get the number of tasks needed satisfying max_chunk_size
     * to download the resource:

     * >> Send HEAD request for url "line".
     * From response, get the size of resource and check whether the server support partial download.
     * >> When server doesn't support partial download, max_chunk_size equal to total resource size
     * and number of tasks equal to 1.
     * Otherwise set number of tasks into the number of threads
     * and max_chunk_size equal to total resource size divide the number of tasks.
     */
    num_tasks = get_num_tasks(line, num_workers);
```

```

// Get calculated max_chunk_size
bytes = get_max_chunk_size();

/** Generate all needed tasks and put them into "TODO" queue to download the resource of current url.
 * >> Count the number of tasks currently in the "TODO" queue.
 * >> Put the generated download task into "TODO" queue. Wait for threads "workers" to process.
 * When there is existing free thread "worker" and processable "TODO" task,
 * task will be processed and put into "DONE" queue.
 */
for (int i = 0; i < num_tasks; i++) {
    ++work;
    queue_put(context->todo, new_task(line, i * bytes, (i+1) * bytes));
}

/** Get download results back from "DONE" queue after threads "workers" done the process.
 * >> Decrease the number of tasks currently in the "DONE" queue which wait for get the result back.
 * >> Get processed task from "DONE" queue
 * and save the download chunked data as files by the name of "min_range" into download dir.
 * After successfully saved the data, set the task free.
 */
while (work > 0) {
    --work;
    wait_task(download_dir, context);
}

/** Merge the files -- simple synchronous method
 * >> Combine download dir and the "min_range" of each task as file name, then read data from this file
 * >> Keep writing data into merge file which named by param "line".
 *
 * Remove the chunked download files
 * >> Combine download dir and the "min_range" of each task as file name, then remove this file.
 */
merge_files(download_dir, line, bytes, num_tasks);
remove_chunk_files(download_dir, bytes, num_tasks);
}

// Cleanup
fclose(fp);
free(line);

free_workers(context);

return 0;

```

2.2 Similar algorithm in notes:

The implemented algorithm is similar to the solution of “The Sleeping Barber Problem” and “The producer-consumer Problem” in the notes. Due to there is only one barber, the solution of “The Sleep Barber Problem” can not support multi threads.

But in “The producer-consumer Problem”, there are not number limitation of producers and consumers. Several producers process download tasks simultaneously. And several consumers consume the producer downloaded chunked data and save it as file.

So the implemented algorithm is mostly like the solution of “The producer-consumer Problem”.

2.3 Any Improvements:

Current solution: Only can download URL resources one by one from URL file. In the specific URL resource downloading, consumer can only run after all tasks are generated and added to “TODO” Queue. When the number of task is large, it will impact the performance significantly.

For these two situation following improvement can be made:

1. Read all target URLs into Queue, fork() to create new process to simultaneously download each URL.
2. In each URL downloading, when new task starting to be added, using multi threads to let consumer and producer simultaneously running.

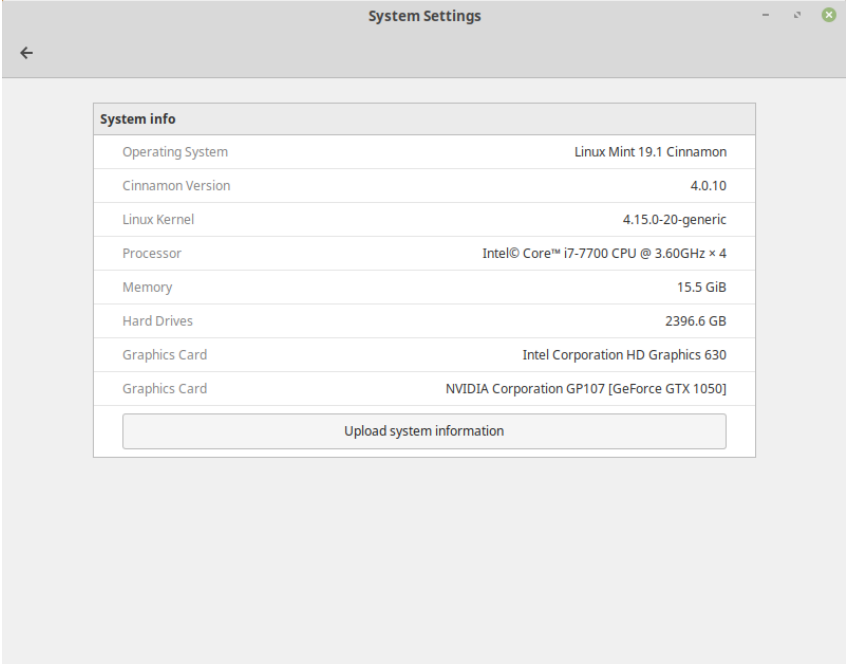
By doing these two improvements, it can be achieved that multi process for downloading several URLs at the same time, and have an improvement on the running timing of consumer.

3. Performance analysis

3.1 Performance test environment:

PC: Using Lab PC (detail as following screen shot.)

Timing: Conducting test off internet peak hours (from 6:00PM – 11:30PM).



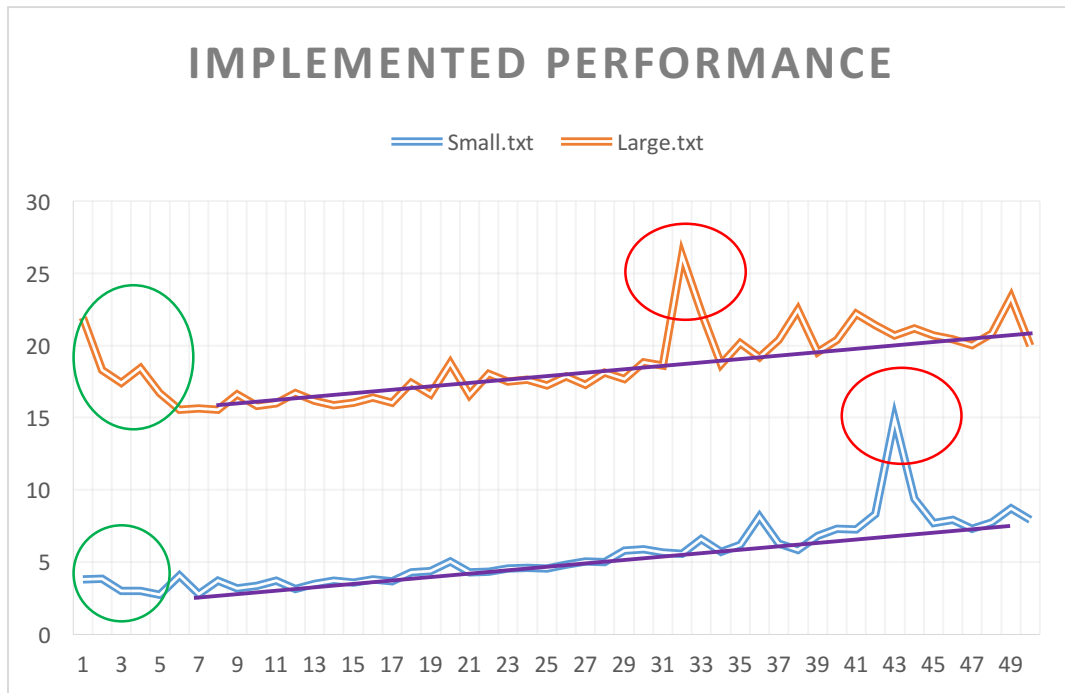
The screenshot shows the 'System Settings' window in Linux Mint. The 'System info' section is expanded, displaying the following details:

System info	
Operating System	Linux Mint 19.1 Cinnamon
Cinnamon Version	4.0.10
Linux Kernel	4.15.0-20-generic
Processor	Intel® Core™ i7-7700 CPU @ 3.60GHz × 4
Memory	15.5 GiB
Hard Drives	2396.6 GB
Graphics Card	Intel Corporation HD Graphics 630
Graphics Card	NVIDIA Corporation GP107 [GeForce GTX 1050]

At the bottom of the 'System info' section, there is a button labeled 'Upload system information'.

3.2 Performance Test Description:

Performance test is separated by two different kinds of URL resources: small file and large file. By running implemented downloader several times to calculate the mean running time in different number of threads. Considering the stability of network, following red highlight huge lumpy data should be ignored.



3.3 From the line chart, performance changes as the number of worker threads increases are as below:

1. Firstly, download time decreased.
No matter downloading small file or large file, in a certain number range of thread (as green highlight shown), compare to single thread, multi thread indeed help on the downloading performance. But multi threads have more effect on downloading large file. When number of thread increases, download time of large file is significantly decreased. But for small file, download time is decreased but not that huge.
2. Secondly, downloading time hit the smallest point and then start to increase.
As the purple line shows, when the number of threads keep increasing, it is found that downloading time does not decrease anymore. In contrast, it increases in trend. The reason is that when thread increasing, the downloaded partial file also increased. This will increase the I/O time when merging all the partial files.

3.4 What is the optimal number of worker threads?

As the time of whole download process can be separated roughly into two parts: "HTTP response time" + "Disk I/O time". Multi thread will decrease "HTTP response time" but at the same time it also increases the "Disk I/O time". There should be a balance between the effect on "HTTP response time" and "Disk I/O time". When the balance is met, the current thread number is the optimal number of worker threads. Due to the test environment is different, so the optimal number of worker threads maybe different also. Base on my test environment, the optimal number is around 5 to 7.

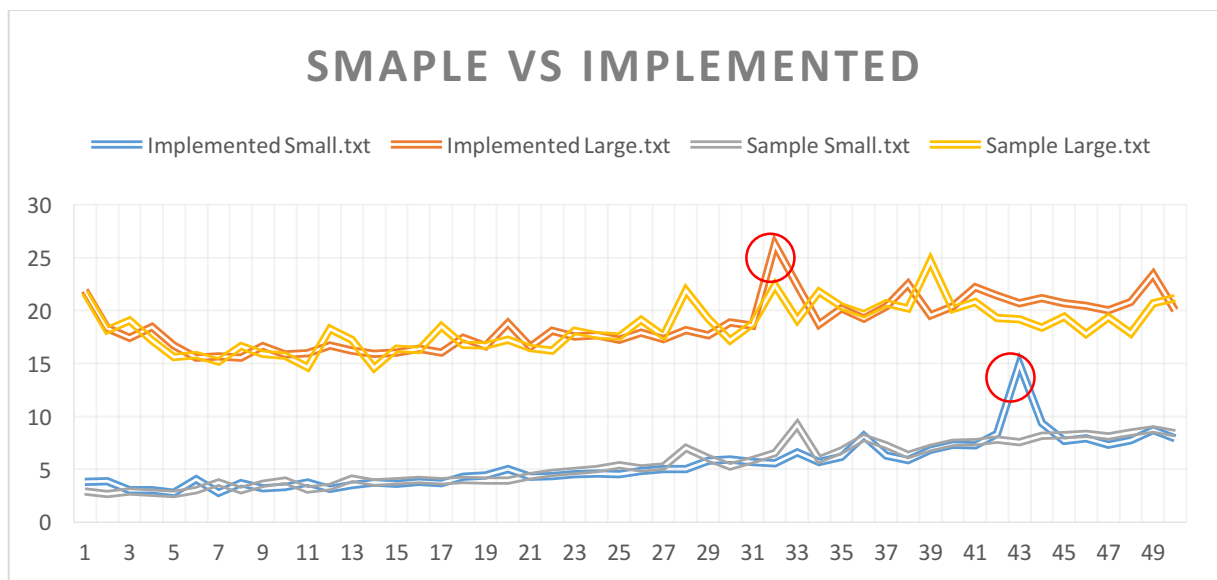
3.5 What are some of the issues when the number of threads is increased?

1. When the number of threads is increased, equally the target URL resource will be divided into more partial files. When merge all these files into one, I/O time will significantly increase.
2. Also when threads increased, it means the number of HTTP request will increase too. When the number of HTTP request come over the threshold, Network performance will decrease. The result of HTTP response time will increase.

3.6 How does the file size impact on performance? Are there any parts of your http downloader which can be modified to download files faster?

1. It is easy to understand that downloading large file will cost more time than small file.
2. Meanwhile, multi threads have more efficient on large file. The reason is that HTTP response time can be divided into "HTTP request send time" + "Extract data from server (DB I/O time)" + "HTTP response send back time". That means no matter request a small size data or large size data, the "HTTP request send time" and "HTTP response send back time" are the needed time cost. Only when requesting large size data, the "Extract data from server (DB I/O time)" is significantly large than "HTTP request send time" and "HTTP response send back time", there will have performance improvement.
3. For the improvement, as introduced above multi process should be used to download several URL resources simultaneously. Also for file downloading, a paired threshold KPI (file size and approximately optimal threads) should be add to avoid the performance negative impact of large threads number.

3.7 Sample downloader vs Implemented downloader



Considering the stability of network, following red highlight huge lumpy data should be ignored. Basically the performance between Sample downloader and Implemented downloader are likely the same. For large file download when number of threads is over 40, sample downloader is fast than implement downloader. Suspecting that sample downloader is implemented a higher efficient merge function.

3.8 Analysis the approach used for file merging and removal.

In the implemented downloader, the approach of file merging and removal is as follow:

1. Base on the source directory where holds the files combine combine with “min_range” which is calculate by max chunk size into a target file path. Then read from the file and write into the merging file. When come to removal, remove target file path need to be assembled by the source directory and the “min_range” again, and then delete it.

3.9 There are two improvements for above approach:

1. Combine merging and removal together to reduce assembling the target file path again and again. When data is read from the partial file, it should be removed directly.
2. When memory is large, instead of saving partial files into disk, assign a place in memory to store the download data. This will avoid low performance of disk I/O and there is not need to remove the partial files.

3.10 Is the current method of partially downloading every file optimal? If not, suggest a way to improve performance.

For single file downloading, although currently is using multi threads to improve the productivity. But it is not the optimal solution. Such as in this solution, only after HTTP response data is saved into disk, then reading and merging start to process.

A data read-write simultaneously cooperate solution could be implemented to avoid disk I/O and other useless data operation.

Also as previously mentioned, for file downloading, a paired threshold KPI (file size and approximately optimal threads) should be add to avoid the performance negative impact of large threads number.

4. Conclusion

Through this performance analysis, it shows that implemented downloader is successfully implemented.

Although it has similar performance to sample downloader, but there are still some improvements can be made. Also it is known that multi threads has its advantage and disadvantage. To achieve an optimal result, people should focus on finding the balance of between advantage and disadvantage. (Such as the balance of multi threads decreasing time and I/O increasing time)