ENCE360 Assignment

Dylan White
29861039

October 18, 2018

Contents

1 Introduction

This report describes the implementation and analysis of a minimal HTTP client and a concurrent queue using semaphores. These were used to download a selection of files from the internet. The concurrent queue uses a circular buffer to utilize the multiple cores available on the University of Canterbury CSSE lab computers. The following sections will cover an algorithm analysis for the downloader.c file, how program testing was carried out, a performance analysis of the downloader, comparisons between the implemented and given downloader files, profiling information of the implementation, and suggestions for potential improvements to the downloader.

2 Algorithm Analysis

2.1 downloader.c

The algorithm implemented in `downloader.c` is similar to the producer-consumer problem discussed in lectures. Producers and consumers are taking tasks in and out of a queue. Worker threads (producers) are downloading content passed by URL argument and placing it in a queue for the consumer to save onto disk. Issues can arise if the code is written in a way where the producer tries to place data in the queue if it is full and the consumer attempts to remove data from an empty queue. This is prevented in this implementation with the read and write semaphores used in the circular buffer. Care had to be taken to initialize the semaphores correctly such that reading was prevented before writing occurred. As there were multiple producers and consumers, the queue was also protected with a mutex which locked the queue when reading or writing and unlocked it at the end of the read or write function. This is necessary as with multiple producers and consumers sharing the same memory space a race-condition can occur where two or more processes attempt to read or write into the same memory slot at the same time.

Below is the `downloader.c` file with lines 200-230 commented for a step by step breakdown of what the algorithm is doing.

```
Context *context = spawn_workers(num_workers);
/* Wait for tasks to be added to the queue
 * Take the task off the queue and download into a buffer
 * Put the buffer onto another queue and repeat
 * the process for every task */

int work = 0; //the number of tasks that need to be completed
while ((len = getline(&line, &len, fp)) != -1) {

/* fp is the URL that we want to download from and is passed into the
* line char* array which gets passed to each new_task to make a task
* for the worker function. -1 checks that there is still URLs to pass
* to the worker task */

    //checking for newline char at end of line
    if (line[len - 1] == '\n') {
        //set last char to null byte before putting on the queue
        line[len - 1] = '\0';
    }

    //increment the number of tasks available
    ++work;
    //put the task on the context TODO queue which will be downloaded.
    queue_put(context->todo, new_task(line));

    //if we've filled the queue up enough, start getting results back
    //if all workers are busy - wait
    if (work >= num_workers) {
        /*take total tasks down to allow new tasks to be
        added as wait_task is about to start downloading*/
        --work;
        //downloads the URL on the context TODO queue
        wait_task(download_dir, context);
    }
}
/*out of main loop. while there is still
final tasks to complete keep going*/
while (work > 0) {
    //decrements number of tasks
    --work;
    //same as above
    wait_task(download_dir, context);
}
//cleanup
fclose(fp);         //close the file pointer
free(line);         //free allocated memory to avoid leaks.
free_workers(context);
return 0;
}
```

3 Testing

3.1 http.c

Initial testing of `http.c` file was conducted using the given `http_test` program. The output that was displayed in the terminal was compared to the data that the web page displayed when viewed in a web browser.

The `http_download` program was run to assert that the correct data was being downloaded from the web page. The `test_download.sh` file was executed to compare the downloaded data against a known working file downloader `wget`. `Valgrind` was used to check for memory leaks in the program. All resources were successfully freed with no possible leaks.

3.2 queue.c

The `queue.c` program was tested using the given `queue_test` test which computed a sum and returned the value. The sum was compared to a known answer and if a match occurred the program was assumed to be working. It was also given that the implementation should complete within two seconds on CSSE lab computers. The average execution time over 10 tests was found to be 0.9739 seconds.

`Valgrind` was used to check for memory leaks in the program. It was found that 1614 bytes in 4 blocks were still reachable at program completion but no blocks lost.

3.3 downloader.c

The main downloader was tested by running the `downloader` program. The MD5 checksum of the downloaded file was compared against the MD5 checksum generated when using `wget` to verify the file had downloaded correctly using the `md5sum` command in the terminal and manually comparing using a difference checker online. `Valgrind` was also used to check for possible memory leaks by running `valgrind --tool=memcheck ./downloader large.txt 32 download` and was found that all heap blocks were freed – no leaks are possible.

4 Performance Analysis

4.1 Hardware

All tests were run on UC CSSE lab computers. Hardware specifications were found by running `cat /proc/cpuinfo` and `cat /proc/meminfo` for CPU and memory details respectively. The CPU was an Intel Core i7-7700 @ 3.60 GHz with four cores and eight threads available per core for a total of 32 threads. Total memory was 16 GB with approximately 13 GB available. Cache info was also found by running `cachegrind`.

Cache	Description
L1	32 KB, 64 B, 8-way associative
L2	32 KB, 64 B, 8-way associative
L3	8 MB, 64 B, 16-way associative

Table 1: Cache levels of CSSE lab computers

4.2 Software

Performance analysis was conducted using the given *small.txt* and *large.txt* files with the downloader. A python program was written to automate testing which can be found in the `/test` directory. Timing data was written into a `.csv` file and then processed with MATLAB to produce plots. To find a preliminary trend with increasing threads, the downloader was run 10 times per thread and then averaged. This was completed for 20 thread values between 1 and 100 threads for each text file. This produced the results seen in Figure 1.

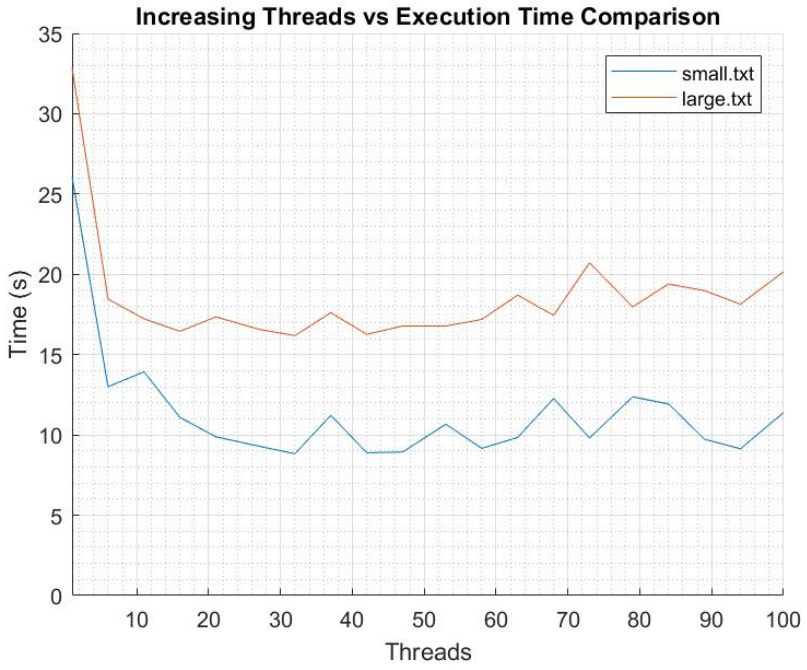

Figure 1: Relationship between increasing threads and decreasing time.

It becomes apparent that the *large.txt* files take longer to download which is expected behaviour due to requiring more data to be transferred. It was found that for both the *small.txt* and *large.txt* files that 32 threads gave the smallest execution time. This is consistent with the hardware data discussed in 3.1. The timing data became volatile for thread counts greater than 32, and had a slight performance decrease if extrapolated. This is due to context switching which is required when more threads than can be run are trying to run. If the threads all have the same priority, they will be given equal time to execute until

completion. Saving the state of each thread before initializing the next has an overhead. This overhead is what decreases the performance for thread values greater than 32.

All testing was completed with a sample size of 10 and averaged. This was in an effort to reduce the deviation caused by external factors such as UC network traffic and other IO bottlenecks that could potentially occur during testing.

4.3 Profiling

Profiling tools `cachegrind` and `Oprofile` were used to further examine the efficiency of the implementation. `cachegrind` was used to determine how effectively the cache was being utilized when executing the downloader.

Cache	Read Miss (%)	Write Miss (%)
L1	0.09	0.00
L2	6.5	9.2
L3	0.7	2.2

Table 2: Cache miss rate of downloader.c with *large.txt*

The high miss rate in the L2 cache suggests that the algorithm has poor locality and is one of the performance bottlenecks.

`Oprofile` data for `./downloader small.txt 32 download` showed that the program spent the majority of the time in `malloc.c` at 17%. This is expected as when running `valgrind` on the same executable it returned 100,000 allocations and 100,000 frees. `http.c` was the next most accessed file at 4.9% with the majority of the time being spent reading data in the `recv()` function and in `realloc()` function for extending the file size.

4.4 Comparison

A pre-compiled version of the downloader was given to be used to compare the performance of the written implementation with a known working, efficient implementation. To compare the two downloaders, `./downloader large.txt 32 download` was executed for both the implemented and `.bin` programs. This test was executed 10 times, and averaged, for each program. This produced the data seen in Figure 2.

It was found that on average the implemented downloader ran 0.6778 seconds slower than the pre-compiled downloader. This is approximately 4% slower than the pre-compiled version. This difference can be assumed to be due to varying test environments over the course of the day. The standard deviation was also calculated to be larger when testing the implemented version than when testing the .bin program which could suggest differing network patterns on the UC network while being tested.

Version	Mean Execution Time (s)	Standard Deviation (s)
.bin	16.0464	1.1525
implemented	16.7242	1.2790

Table 3: Mean time and standard deviation for .bin and implemented downloader files averaged over 10 tests on *large.txt*

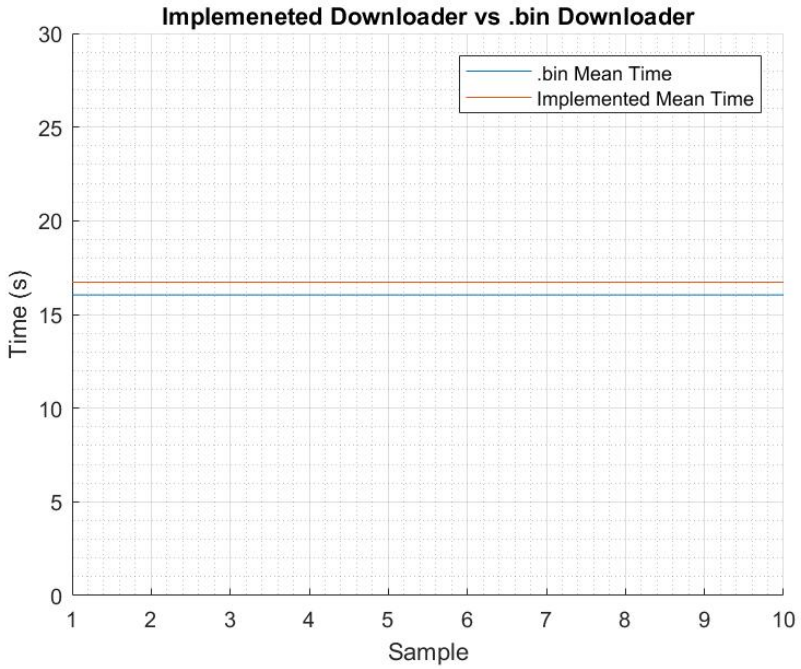

Figure 2: Timing comparison between implemented code and pre-compiled code.

5 Potential Improvements

- Performance improvements could be gained by analyzing how the data is stored in memory and organizing the data in a more sequential way to reduce the number of L2 cache misses.
- A larger buffer size would reduce the number of calls to `realloc()` which in turn would reduce the overhead.
- The program could be written as a distributed model to execute over multiple computers to make use of more threads for downloading.
- Using the GCC optimization flag `-Ofast` or `-O3` may also increase performance but at the cost of size. Preliminary testing with `-Ofast` showed an reduction in execution time averaged over 20 tests. However, this is likely to have been due network traffic.

- Printing the file information upon successful download was found to take about 4% of program time. Removing the print statements may improve efficiency.

6 Conclusion

The `http.c` and `queue.c` files were successfully implemented in an efficient way. Testing showed that both files executed in an acceptable time-frame compared to pre-compiled versions and previously known timing data. The optimal number of threads for the program to run on a CSSE computer was found to be 32. Several suggestions were made on how to potentially improve the efficiency of the code.