Jhi56, 25670115
24/10/17
ENCE360 Assignment

An analysis of downloader.c:

Context *context = spawn_workers(num_workers);
Creates num_workers amount of threads to things in a queue, which is part of the context struct. The threads do as follows:

1) Wait for a task to be added to the queue
2) Take the task off the queue, and download the content to a buffer.
3) Put the buffer onto a separate queue and repeat the process as long as there is still at least one task.

int work = 0;
This integer keeps track of the amount of tasks that need to be done.

while ((len = getline(&line, &len, fp)) != -1)
This puts a line from getline into the address at line of size len from a file pointed to by a file pointer fp. It also simultaneously error check with getline()'s return value not being equal to -1.

if (line[len - 1] == '\n')
This checks if the last character of the line is a newline character

line[len-1] = '\0''
This sets the last character from the line to be a null byte.

++work
Tells the program that there is going to be another task.

queue_put(context->todo, new_task(line));
Puts the first task of the context's todo queue onto the queue for tasks. The task is new_task(line), which is a url to be downloaded from.

if (work >= num_workers)
If there are enough tasks in the queue then we can start getting results back from said tasks.

--work;
Decrement the amount of jobs to do by 1 as jobs are now being done.

wait_task(download_dir, context);
Completes a task from the aforementioned queue (not the context's queue!). It puts it into download_dir and the task itself is specified by context. Wait_task will open a file with snprintf() and put data into into the file with a file pointer (among other things).

while (work > 0)
We are now outside the loop and need to make sure that all the jobs are done. Work being greater than 0 tells us that there are indeed still things to be done.

--work; & wait_task(download_dir, context);
Same as before.

fclose(fp);
Closes the stream to the file made at the start of main().

free(line); & free_workers(context);
Frees the memory allocated to line, and the memory allocated to context. Free workers is a function that is used to free all of the data associated with the context objects as they have data in queues and have various attributes.

return 0;
Ensures main returns so the program can complete.

This procedure is similar to the bounded buffer problem, where producers and consumers are taking things in and out of a queue. The spawned worker threads, or the producers, are downloading content and putting it into a queue for the main thread, or the consumer to then put the content into the disk.

Data Analysis - My http.c & queue.c versus the provided downloader program.

NB: At times, the downloader for both my own and the provided compiled one would yield quite large times in comparison to the norm that were used in the averages. With larger pools to get an average or a different set of timings all together, the data might be a little bit different.

Large - 10 items, 32.6MB
Small - 36 items, 2.4MB

My downloader

File	Threads	Average REAL time in seconds. (5 runs)
small.txt	1	31.738
small.txt	4	10.917
small.txt	8	10.996
small.txt	16	11.029
large.txt	1	33.329
large.txt	4	22.104
large.txt	8	15.22
large.txt	16	15.973

Provided downloader

small.txt	1	28.565
small.txt	4	11.525
small.txt	8	12.645
small.txt	16	11.02
large.txt	1	32.563
large.txt	4	20.564
large.txt	8	15.228
large.txt	16	15.576

Speculation and summary

To address the elephant in the room, the larger files take longer to download than the smaller ones. Surprisingly, 1 thread for both the large and small files yielded an incredibly similar result, but the addition of extra threads resulted in a much faster execution time for small.txt. This could be because the threads can move on from file to file very fast due to the small file sizes while in large.txt the threads are stuck spending time downloading the bigger files. Another reason could be because there are more files to be downloaded in small.txt so the threads could go from one file to another with less hassle.

Reasons for why more threads do not continue to improve the results could be because of the program being bottlenecked by either the download speed or the size of the buffer in the http file. That, and more threads cause more overhead. With a much larger queue size this might be less of an issue as with smaller queues it could be the case that lots of threads are getting caught at the semaphores in queue_get and queue_put and are unable to do anything. This results in the program making more threads than it can even use. If the program was downloading a movie, for example, that was just one file that was very large, it would be expected that 1 thread would do just as well as 4, say, as each thread works on its own download.

Comparing the tables

The results from both the provided downloader and my own were mostly quite similar. Overall the provided downloader seemed to do marginally better than my one regardless of the amount of threads at work for the large.txt files. For small.txt my downloader performed better in comparison except for the case of 1 thread. The provided downloader surprisingly did worse with 8 threads than with 4 and 16, something that does not happen on a significant scale anywhere else in the data.

Optimal amount of threads

From the first table we can see the the optimal amount of threads is around 8.

The second table is a little more difficult to interpret. While 8 remains a pretty good amount of threads, the program seems to do comparatively better than my own for 4 threads in large.txt as opposed to different amounts of threads. That being said, as large.txt did much better with 8 threads we cannot consider 4 for an optimal number of threads. For small.txt, the provided downloader actually performed slightly better with 4 threads than with 8 or 16, but as the timings were much more negligible, especially in comparison to large.txt, it is probably better to stick with 8 threads if the downloads were known to be bigger.