ENCE360 Assignment 2019

Noah King 96851177

1 downloader.c

Description of lines 228 - 264 of downloader.c i.e.

```
228: work = 0, bytes = 0, num_tasks = 0;
```

To

```
264: free(line);
```

1. Grab each of the URLs from the supplied file as a 'Line' and replace each newline char with aterminator sequentially
2. Determine the number of tasks for this Line (i.e. the number of downloads to specify re. range) (note: This performs an HTTP 1.0 HEAD query and extracts the content-size field from the response)
3. For each task, Create a new download with this URL and appropriate range; Push it to a queue.
4. Wait for all tasks to finish
5. Merge the individual files created by each task, remove extra files
6. Free all resources

This is most similar to the dispatcher/worker model shown in the lecture notes.

Jobs begin as soon as they are added to the 'todo' queue. We do however wait until all jobs have completed before merging the partial files. An alternate way to approach this may be to introduce the merge files procedure into each thread and provide each thread with a location within the resultant file to write to.

Alternatively we could act on each completed job whenever work is decremented. We can fork this process here and proceed per the first alternative. These both will have performance benefits. There are numerous ways we could do this, we may choose to fork the process and use pipes such that we are not concerning each thread with memory mapping and allow the parent to handle this. As long as we are no longer *wasting* the time around wait_task() there will be performance benefits.

The ./download script will download each partial file to a location on disk, read and remove these partial files and write a brand new complete file to disk. There is an obvious redundancy. Instead, we should cut out the middle man and download directly into the complete file.

This is different to the implementation within downloader.c; Here I have opted to use memory mapping and write each file sequentially into the map instead of copying parts of files sequentially and having them written and read to disk. We then allow the operating system to handle copying the entire set of changes back.

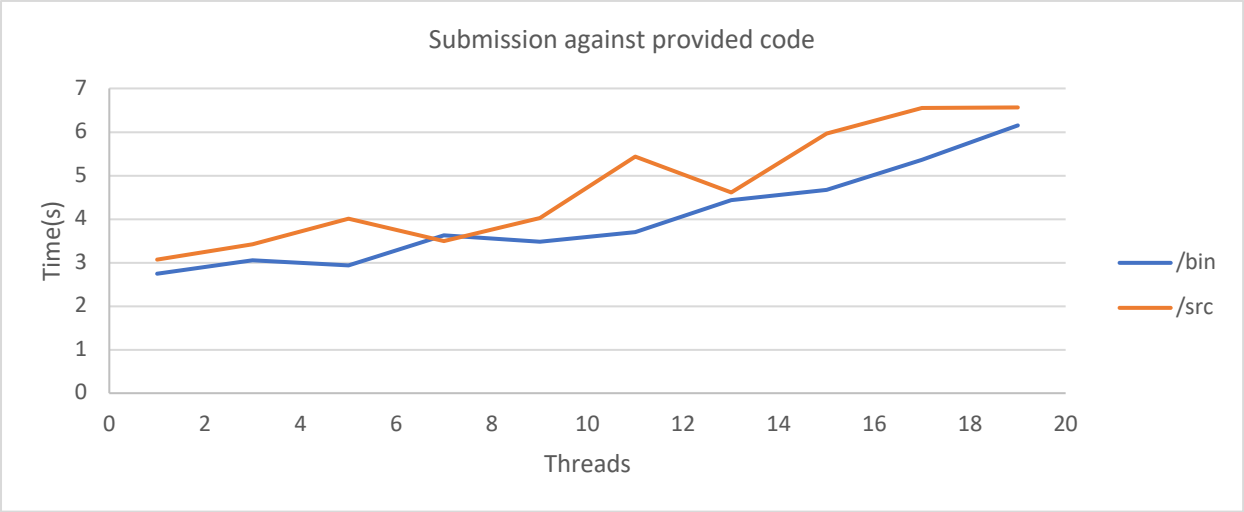

Figure 1 - Submission code vs Provided Code

We can observe that there is a difference between the submission code and the provided downloader. I believe that this may be related to the asynchronous vs. synchronous implementation of the merge procedures.

Threads	Small.txt	Unsplash.txt	Empty.txt	larger.txt
1.000	2.972	4.822	0.365	8.127
2.000	3.368	5.649	0.362	13.997
3.000	4.025	6.127	0.364	13.180
4.000	3.444	7.654	0.404	17.069
5.000	3.594	6.748	0.366	15.002
10.000	4.947	6.072	0.368	17.562
15.000	6.013	11.586	0.374	17.284
20.000	6.737	11.249	0.475	18.798
25.000	7.775	11.744	0.425	21.370
50.000	22.138	20.676	0.470	30.597

Figure 2 – Raw results

These are the averages for running each number of threads on each file in the root directory over 10 trials.

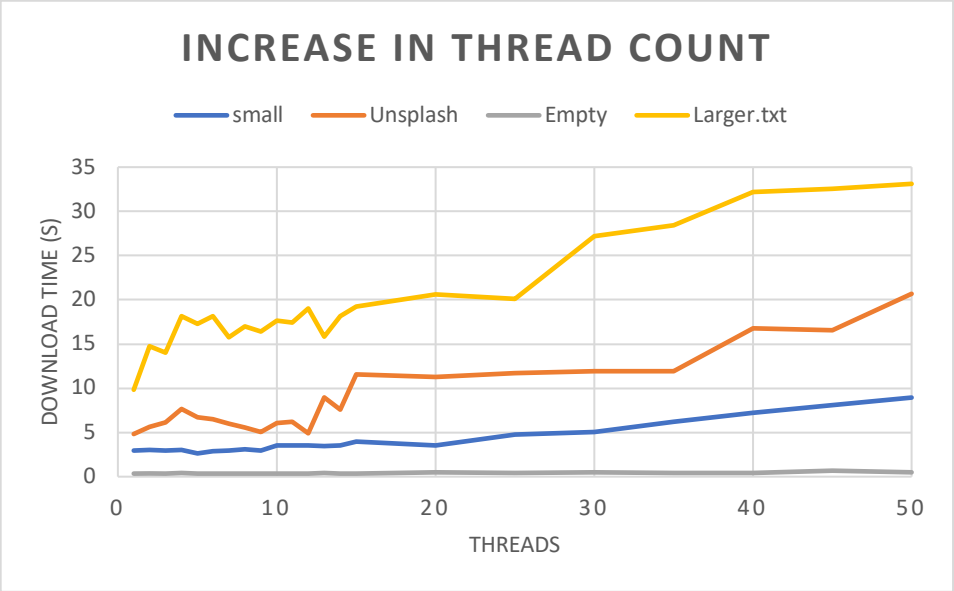

Figure 3 – Thread counts for each file

We may observe that there is a gradual increase in download time given the increase in threads. We may observe that there is an optimal number of threads somewhere between 3 and 15 threads, after this there is a gradual increase in download times. This 'sweet spot' will vary from machine to machine. Obviously these results are dependant on various factors such as computer speed, load, scheduler, connection speed and server load.

The overall increase in download time can be attributed to bottlenecking from our concurrent queue(s) and the associated memory allocation. This can slow down the overall time the procedure takes and slows the overall process.

From these results, we can clearly see that there is a direct relationship between download time and the file size. (Files in 'larger.txt' are from 3 – 19 mb). We should expect this as the procedure will require greater dynamic memory allocation than smaller files.

The file 'empty.txt' simply contains a URL for the deprecated google images site, which is a few kbs. We can observe a slight gradual but consistent increase in download time in figure 1. This supports the idea that we have a bottleneck caused by our concurrent queue,

2 Particulars about submission

- If the server does not support partial downloads, we will use a single thread and a single big chunk.
- If a single download fails, we simply proceed with the other downloads and abandon the procedure
- There seems to be issues with some of the URLs in 'large.txt'. In order to analyse the performance I have gathered several 'guaranteed to work' URLS in 'larger.txt'. This is likely a bug on the client end. (I think it is related to timeouts, but was unable to correctly diagnose the bug)
- There is an immediate segfault on macOS, likely pthread related. No issues on other platforms.