Josh Thieme
27365475

ENCE360 Assignment - The Producer-consumer Problem

N.B - this assignment has been submitted a week late as I was granted an extension.

Algorithm analysis

The downloader follows the following algorithm:

Firstly, a number of worker threads are created equal to a given command-line argument, and each is set to work doing the following:

- Take a task (i.e download to be completed) from the head of the queue
- Perform the download to a temporary buffer
- Enqueue the result to another queue, ‘done’, which holds the completed downloads
- Repeat as long as there remain tasks in the queue

When the workers are initially created there is not yet any work to be done, as the queue is empty, so each thread waits for the addition of a task to the queue. The program then iterates over the lines in the file (each containing to url to download from) and adds a corresponding task to the queue. A task consists of a URL and a buffer in which to store the contents. Once the queue is full of tasks (or the number of tasks reaches the number of worker threads), the main thread begins executing the following procedure:

- Take a result (completed download) from the head of the ‘done’ queue (if one exists)
- ‘Consume’ the result (i.e save its buffer contents to a file)
- Repeat as long as there remain downloaded results to consume.

program then begins iterating over the results as they are produced and consuming them (i.e dequeuing each of the completed downloads from the ‘done’ queue and saving their respective buffer contents to a file).

The algorithm described above, employed in downloader.c, is akin (but not identical) to a multi-producer, single-consumer generalization of the classic ‘producer-consumer’ problem (also known as the ‘bounded-buffer’ problem). In this case, the spawned worker threads act as the producers and the main thread acts as the sole consumer.

An improvement could involve multiple consumers - at present a single consumer (the main thread) is responsible for consuming all of the producers’ work (saving the contents of each filled buffer to a file). The advent of multiple consumers could ensure that the rate at which the filled buffers are saved as files approaches the production rate.

Josh Thieme
27365475

Performance analysis

For this analysis, both my own downloader implementation and the provided precompiled version were executed multiple times with different numbers of worker threads (4, 8, 16 and 32 threads), and the average execution times recorded for both small file sizes (small.txt) and large file sizes (large.txt). This was done using a bash script, and the processed output can be seen below.

My own implementation		
file	**threads**	**average time (seconds, 5 repetitions)**
small.txt	4	11.871
small.txt	8	11.9912
small.txt	16	11.813
small.txt	32	11.2565
large.txt	4	17.6618
large.txt	8	16.1466
large.txt	16	15.397
large.txt	32	15.7222

Provided, precompiled downloader		
file	**threads**	**average time (seconds, 5 repetitions)**
small.txt	4	11.1606
small.txt	8	10.7984
small.txt	16	11.4795
small.txt	32	11.9432
large.txt	4	17.9388
large.txt	8	16.7968
large.txt	16	15.5096
large.txt	32	16.307

From the above tables it can be seen that, though there exists a performance differential between my implementation and the precompiled one provided, it is negligible at best - indeed mine performs slightly faster in the case of small.txt with 32 threads (11.2565 seconds versus 11.9432 seconds), as well as in *every* case on large.txt, but only by small margins.

The situation becomes more complex, however, when we raise the question of optimal thread count. There is no clear-cut answer to this - when running my own implementation on small.txt the smallest average running time occurs using 32 threads, however in the reference implementation 8 threads is optimal. Shifting focus to large.txt we see that my implementation suggests 16 threads as the optimal number, whereas the average times for the reference

implementation would suggest 16 as optimal. If this analysis were to be repeated with a higher number of repetitions these strange results would likely even out.

In terms of potential improvements, increasing the buffer size is the obvious step - the number of system calls would naturally be reduced by implementing this change, resulting in a reduction of overhead. This, in addition to the increase in consumer threads discussed earlier, would likely result in a significant increase in efficiency.