ENCE360 Assignment: Multi-threaded HTTP downloader

Author: Jordan Pyott
Student ID: 87433186

Algorithm analysis

The algorithm found in the main routine of the `downloader.c` file (lines 175 and onwards) initially takes in a URL, the number of worker threads and an output directory as parameters to the program. The program then creates a new directory with the name provided as an argument; the program then opens a file, spawns working threads.

From here, the program starts its main event loop; this will run as long as we keep receiving a new URL from the URL file, then it will return the size of the line and assign the line to a variable called `line`, note this size does not include a null terminator character. Upon receiving an end-of-file signal, the `getline()` function will return `-1` and break out of the loop.

If the end-of-file is not received, the algorithm will have received the size of the URL and the URL itself. The main routine will then create new tasks and add them to the `context->todo` queue; these tasks are used to download a section of the file. After these tasks are added to the `context->todo` queue, the worker threads will then get an item from the `context->todo` queue (removing it from `context->todo`) and process the task, the received bytes will be then placed into the task's buffer `task->result`, and the task will be placed into `context->done` to establish that the task has been completed.

The dispatcher thread (initial thread that the user starts) gets tasks from the `context->done` queue that have been processed and writes the downloaded bytes from `task->result` into a file on disk corresponding to the URL provided, this will happen one by one until the file is fully written.

The final stages of the main routine are to free excess memory and workers and close the open file, finally the process terminates, and the program has concluded.

Mapping to a design pattern and performance improvements

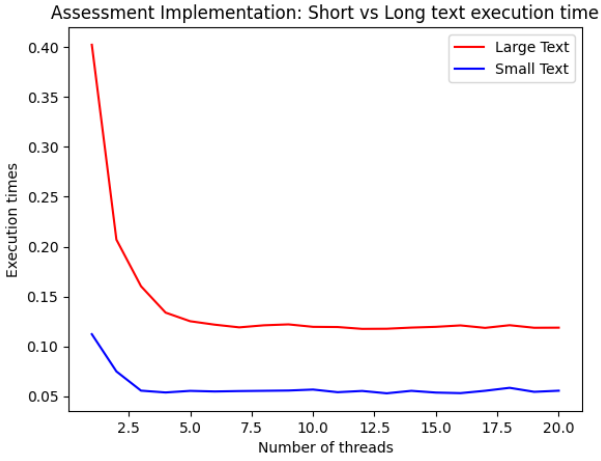
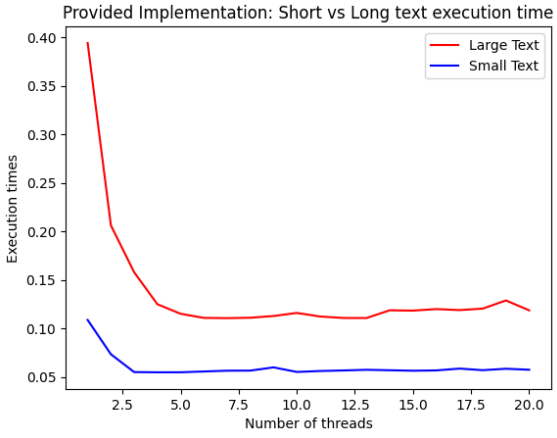
The algorithm above uses the dispatcher/worker pattern (mainly, this is explained below); as we have shown above, the initial thread that the user creates, acts as a dispatcher allocating tasks to a pool of worker threads, the method of allocation is assigning the work to the `todo` queue, where the worker threads get woken to perform the work of downloading the data.

One performance fault found in the program is where the dispatcher thread is writing the files; at this point in the program, the worker threads will remain idle as the dispatcher thread continues

to work, which will cause a bottleneck. It is also important to note that this does not match the dispatcher/worker method. The dispatcher is responsible for completing work rather than specifically allocating the work to the pool of worker threads.

Performance analysis

Below is a comparison of the provided binary vs the assessment implementation, these were analysed using the Lab computers. The comparison was run using a large download test and a small download test *(test files are included in `./report`)*. These were done using the provided python script (`./analysis.py`) to run each test-case three times and take the average across a range of threads $T = \{1 < t < 20\}$, where T is the set containing test cases where t is the number of threads used in that test case. The results can be seen in the figures below:

Assessment Implementation	Provided Implementation
	

As we can see from the figures above, both the `provided implementation` and the `assessment implementation` have an exponential fall-off of execution times that appears to approach an asymptote where the number of threads is no longer having a positive impact on speeding up the download times.

The figures from the `provided implementation` and the `assessment implementation` have almost identical curves, and we see slightly more variation in the provided implementation, however, it is important to note that these results are subject to change due to a number of factors, including connection speed and reliability, the available resources of the machine in use at the current time and the computer's specifications, this is likely to be the cause of slightly different run times.

Finding the optimal number of threads From the figures shown earlier, it appears that the optimal number of threads before significant falloff is between 3-4 threads. This is due to the fact that between 3-4 threads, the gradient of the above graph is approaching 45 degrees. The optimal point is in this range of threads because we are getting the maximum ratio of benefit for the least resources used. One possible reason for this could be to do with the number of available cores the lab machines CPU's have. A machine running in perfect conditions (meaning no external processes running) would have an optimum (non-blocking) thread count when the number of threads used is the same as the number of cores on the CPU. As our data seems to indicate that around 3-4 threads is where our optimum point lies. This could be a contributing factor in our findings as the lab computers are quad-core computers.

While this may be the case from our data, it is important to note that this may not be the case on other machines. This is because different machines will have a different number of CPU cores available. If the given CPU has more cores than the lab computers, its optimum thread count will likely be higher than that given by our data.