Module 8: Portfolio Project

Working with Big Data using Multithreading

Nolan Byrnes

CSC507 – Foundations of Operating Systems

Colorado State University – Global Campus

Professor Joseph Issa

January 15th, 2022

Module 6: Portfolio Project

I created a program that would read the numbers of a text file which contained one billion random integers each, summed the two integers together for each matching line, and wrote the results into a separate text file. Since both huge files had a size of about 6.5 GB, the files could not be read into memory at the same time, so the data had to be read in chunks. Reading both files one line at a time would also take ages to process. In this project, I used two different approaches to increase the efficiency of processing the large amount of data. In the first approach, I used multithreading to process the workload, and in the second approach, I split the huge text files containing one billion integers into ten separate files containing one hundred million and used multi-processing to run the multithreading solution in parallel with the split files. This paper explains the methods used to accomplish the task, as well as an analysis of the results of the methods used.

**Multi-threading Solution**

Stallings explains that multithreading refers "refers to the ability of an OS to support multiple, concurrent paths of execution within a single process" (Stallings, 2019, pg. 150). In my multithreading solution, I had two threads that would read the large text files, two threads to add the data together, and one thread to write the data to the resulting text file.

**Reading Thread**

In the reading thread function, I had the thread open up one of the text files, and perform a loop to grab the data in chunks. After grabbing the chunk of the file, it would append the data to a list, and if the amount of indexes matched the amount of rows for each write, it would send the data into the respective priority queue, with the iterator as the priority number. Each time the thread places data into the priority queue, it increases the iteration (i) by one. this function will exit the loop close the text file and finish when there is no data being returned from the text file.

**Process Thread**

The Process thread is the one that would take the data from the read\_queue\_one and read\_queue\_two priority queues, add the data together, and format it into a string array to be sent to the write\_queue. I had two dedicated threads to handle the number crunching while the other three threads handled the input/output operations. The process\_thread() function had a while loop, and would continuously search for the next queue item from read\_queue\_one until it received one. It would then do the same from read\_queue\_two. Since I only had one writing thread, and the data from both of the text files were being inserted in the priority queue, with each iteration, the matching rows would return, to be summed together. It would then merge the data together and send it to another priority queue for the writer to receive and process.

**Writer thread**

In order to keep the program thread-safe, I only had one thread that was dedicated to writing data to the resulting file. If I had multiple threads writing data to the file, then there would be data loss, and the order of the data would most likely be out of order. The writer thread function opens the resulting file, and will enter a loop, where it looks for an item in the write\_queue. As soon as it finds one, it will write the data to the resulting text file, and increment the lines\_written property by the amount of rows written to the file. The lines\_written property helps the function determine when there are no more lines to write to the file. Relying on if there are items in the write queue would not be accurate, as at any moment, there may not be any items in the queue for the moment.

**Multithreading results**

When I used the multithreading solution on the files with one billion rows, it would have taken a long time to process. Since I was using only one thread per text file to read, and one thread to write, the solution would only be as fast as the two threads could extract the data from the text files, and write to disk. When I had my multithreaded solution run on text files containing 10,000,000 random numbers, it took 31,556 ms. The reason why the multithreaded solution would work slower than you would expect was due to the python Global Interpreter Lock (GIL). Although the job of the GIL is to keep the Python interpreter thread-safe, it does provide additional overhead that can make a multithreaded program run slower than If you were to just use only one thread to execute all of the code (Skvortsov, 2021). This is because when you introduce multiple threads, for one of the threads to complete an action, it must have a hold of the GIL. Skvortsov explains that in order to “acquire the GIL, a thread first checks whether some other thread holds the GIL” , and if it is available, it immediately grabs it. If not, it waits for a switch interval, which is 5 ms by default (Skvortsov, 2021, para. 10). The context switching can get expensive, especially when grabbing and processing large amounts of data.

**Multiprocessing**

Skvortsov explains that multiprocessing “is the ability of a processor to execute several unrelated processes simultaneously” (Skvortsov, 2021, ). This means that each of the processes being started do not share any resources and are independent of one another. by using multiprocessing, we can break through the limitation of the GIL, so there “is no restriction of executing the bytecode of one thread within our programs at any one time” (Tutorialspoint, n.d., para 6 ). With more threads being utilized to read and write the data, the processing time increased significantly. For the multiprocessing approach, I had the text files split into 10 separate files of 100,000,000 random numbers each and had a queue for the 10 files to be processed by the processes, and after they completed, I had the files combined together to give us the final result.

To split the text files into ten separate files, I used the python module filesplit, with the function .split().bylinecount(), which will split text files by the line count specified (PyPI, 2022). By using this module, it was able to split both hugefile1 and hugefile2 at a decent speed, at 627,371 ms.

After I had both text files split, I created a list of 6 processes, calling the multiprocess\_job() function, starting the processes, proceeded with joining the processes to end the task. All ten split files were included in a queue for the processes to process and ran the Multithreading\_solution.execute() function. After all the processes had been completed, I had all the resulting documents combined into one, to give us the resulting text file.

With each process having its own resources and GIL, I was able to use more than one core when running the programs, and even though I was using more threads that would context switch, there were 6 processes, which also meant 6 GIL in total. It took my multiprocessing solution 365,369 ms, which was a significant improvement over the multithreaded solution. You can view the execution results in figure 1 below.

Figure 1.

Execution Speed Results

Text

Description automatically generated

Note. This figure displays the amount of time it took for my program to process two large files using multithreading, and multiprocessing using multithreading. The multithreading solution processed 10,000,000 rows of data, whereas the multiprocessing solution processed one billion in each file.

**Conclusion**

Using multithreading is good for input/output operations, but you still need to keep in mind that context switching can decrease the performance, especially with Python’s GIL. The information that I learned throughout this course has helped me throughout this project because when I was developing the solution, I kept in mind how the computer operates, and having a better understanding of how it works internally can help come with more efficient solutions.

**REFERENCES**

PyPI. (2022). Filesplit. <https://pypi.org/project/filesplit/>

Skvortsov, V. (2021, September 22). Python behind the scenes #13: The GIL and its effects on Python multithreading. blog | TenThousandMeters.com. <https://tenthousandmeters.com/blog/python-behind-the-scenes-13-the-gil-and-its-effects-on-python-multithreading/>

Stallings, W. (2018). Operating systems: Internals and design principles. Pearson.

Turing. (2022, May 28). Python multiprocessing vs Multithreading. <https://www.turing.com/kb/python-multiprocessing-vs-multithreading>

Tutorialspoint. (n.d.). Concurrency in Python - Multiprocessing. <https://www.tutorialspoint.com/concurrency_in_python/concurrency_in_python_multiprocessing.htm>