Module 6: Critical Thinking

Evaluate real-time process scheduling

Nolan Byrnes

CSC507 – Foundations of Operating Systems

Colorado State University – Global Campus

Professor Joseph Issa

December 28th, 2022

Module 6: Critical Thinking

In Module 4, I created a program that would read the contents of a large text file containing random numbers on one million lines, double the number, and write the results to a separate file. When I increased the text file to contain ten million lines of numbers, it took an average of 3912 milliseconds. This paper explains the different approaches I took to increase the processing time, as well as the results.

Trial One

The next method I tried doubled the numbers of the input file in chunks, based on how many files we wanted to split the data by. The trial\_one() method first creates a queue based on how many times we wanted to split up the input file. I decided to use the first-come-first-served scheduling (FCFS) policy for the queue. With a FCFS policy, “when the currently running process ceases to execute, the process that has been in the ready queue the longest is selected for running” (Stallings, 2019, pg. 407). By having this policy, the queue naturally kept the next chunk of numbers from the file to be the next subsequent chunk, helping to keep the chunks of numbers in order. The value of each queue item increased by one, starting at zero. Then I had the function calculate how many rows would be in each of the split files and ran the queue\_consumer() function.

The queue\_consumer() function would take the next item in the queue and take a chunk of the data from the large input file based on the queue number, and how many rows of the text file it should extract. It then doubled each of the numbers and wrote it to a text file that matched the queue number (0.txt, 1.txt, etc.). After the queue consumer function completed, the combine\_files() function would execute, combining all the generated files from the queue consumer function. You can view the trial\_one() function in figure 1 below.

Figure 1.

Trial One Function

Text

Description automatically generated

Note. This figure displays the Trial One Function where I had one process split up the large file into chunks, and combined the results together.

I ran a speed test on this function where I had the input file split into 2, 5, 10, and 20 files, and with an average out of three trials for each one, I noticed that the average completion time increased as the amount of splits increased. See figure 2 for the results of the trials.

Figure 2.

Trial One Results

Text

Description automatically generated

Note. This figure displays the execution times of the original function which doubles all the numbers at once, and the execution times of splitting the file 2, 5, 10, and 20 times.

I expected the processing time to take longer as the amount of files increased because there were more steps to perform as the amount of chunks of the input file increased but I was also only utilizing one core to handle the workload.

To improve the processing time, I decided to run the same method, but taking a multiprocessing approach, which “can make a program substantially more efficient by running multiple tasks in parallel instead of sequentially” (Chung, 2021, para. 6). Since I did not have the files being combined until after the queue was completed, I was able to have multiple processes chip away at creating the files which contained the chunks of the file to be combined. In the multi\_processing() function, it generated the queue, calculated how many rows should be in each file, and created a Process for the amount of cores that I specified in the num\_cores input parameter. After generating the processes, and starting them, each of the processes created started to tackle the generated queue. After each process completed, it was important to use the .join() function because without it, generated processes would remain idle, and would not terminate (DigitalOcean, 2022, para. 5). After the completed processes are terminated, the multi\_processing() function combined all the generated files by using the combine\_files() function. You can view the multi\_processing() function in figure 3 below.

Figure 3.

Multi\_processing Function

Text

Description automatically generated

Note. This figure displays code used when using multiple processes to complete a queue to split the large text file into smaller chunks.

Testing the speed of the multi-processing approach, I split the large text file in 2, 5, 10, and 20 sections, with using 2, 3, or 4 cores to process the file. You can view the main function which displays all the different combinations attempted in figure 4 below.

Figure 4.

Main Function

Text

Description automatically generated

Note. This figure displays the main function which shows the different combinations of file splits and cores to process the large text file.

Out of all the trials that were performed, when we split the large text file into five chunks and had five processes working on the queue is where we saw the best results, with the task being completed with an average of 2803 milliseconds.

Based on the results shown in figure 5 below, splitting the large text file by five, and having five cores work on the queue gave the best results.

Figure 5.

Multi\_processing Results

Graphical user interface, text

Description automatically generated

Note. This figure displays the execution times of the different file splits with different amounts of cores being used. The milliseconds represent an average out of 3 trials.

Conclusion

When I used a queue with a first-come-first-serve policy, when only using one core, it performed slower than when I had the program process the entire text file, double the numbers, and write it to new\_file.txt. this is because there were more steps that the single process would have to take, by creating the queue, splitting the large file, and processing it. When I used multiprocessing to split up the data in the large text file and tackle the generated queue, the processing times decreased, since I had multiple processes working on chunks of the file at the same time. By combining all the files at the end, we were able to ensure that all the chunks of data that were processed were placed in the resulting output file in the correct order. When I ran my speed tests, the large file was able to be processed at the fastest time when I used five processes, with the text file being split in five sections.

**REFERENCES**

Chung, D. (2022, April 26). Multiprocessing in Python. Machine Learning Mastery. <https://machinelearningmastery.com/multiprocessing-in-python/>

DigitalOcean. (2022, August 3). Python multiprocessing example. <https://www.digitalocean.com/community/tutorials/python-multiprocessing-example>

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