Portfolio Milestone 4

Brady Chin

Colorado State University Global

CSC507-2: Foundations of Operating Systems

Dr. Joseph Issa

February 9th, 2025

Portfolio Milestone 4

This milestone compares different methods of processing large text files with numerical data using a Bash script and several approaches in Python. By measuring execution times, we can analyze the efficiency of each method and identify how different processing techniques-sequential reading, line-by-line processing, and parallel execution-affect it.

Results

These results show some contrast in run times for methods. Figure 1 shows the execution time of the Bash script. The execution time of a Bash script was 76 seconds.

Figure 1: Bash script execution time.

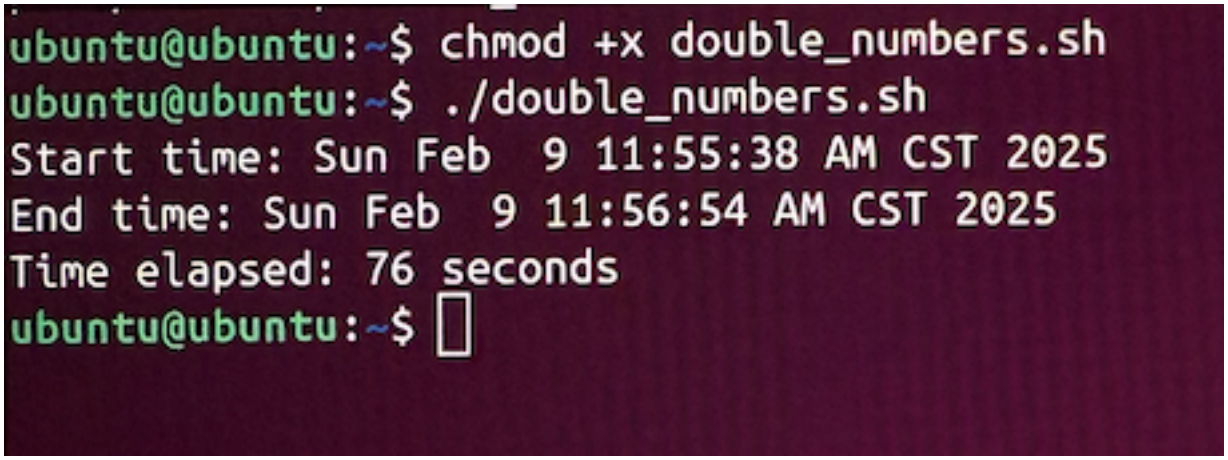
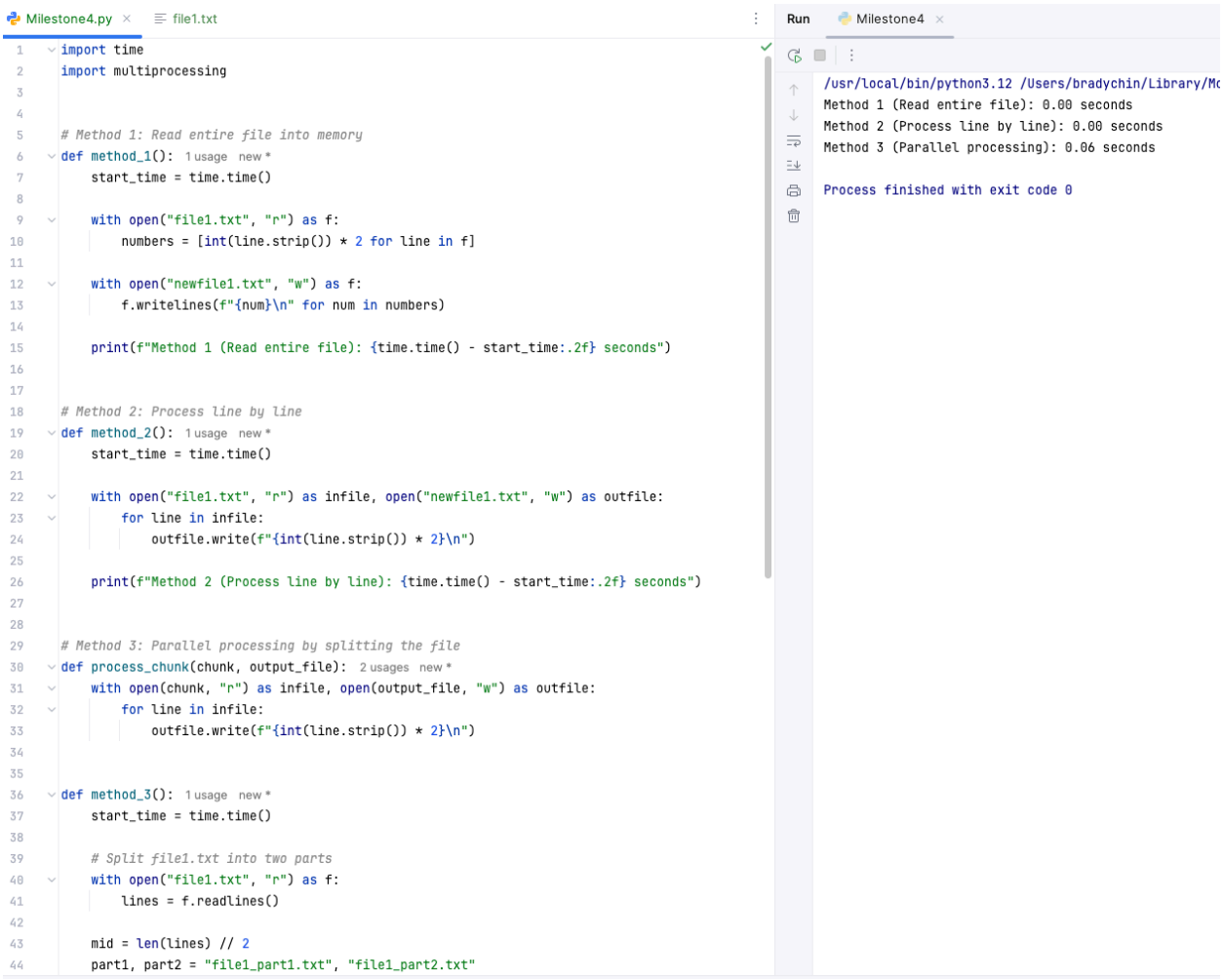


Figure 2 shows the script and the execution times for all three of the Python methods. Methods 1 and 2, which read the whole file into memory and process line by line, respectively, finished in 0.00 seconds. This is indicative of how efficient Python is at handling such operations, further aided by operating system-level optimizations such as disk caching. Surprisingly, Method 3, parallel processing through chunking, was a bit slower at 0.06 seconds.

Figure 2: Python script and execution times.




```python
import time
import multiprocessing


# Method 1: Read entire file into memory
def method_1():  1 usage  new *
    start_time = time.time()

    with open("file1.txt", "r") as f:
        numbers = [int(line.strip()) * 2 for line in f]

    with open("newfile1.txt", "w") as f:
        f.writelines(f"{num}\n" for num in numbers)

    print(f"Method 1 (Read entire file): {time.time() - start_time:.2f} seconds")


# Method 2: Process line by line
def method_2():  1 usage  new *
    start_time = time.time()

    with open("file1.txt", "r") as infile, open("newfile1.txt", "w") as outfile:
        for line in infile:
            outfile.write(f"{int(line.strip()) * 2}\n")

    print(f"Method 2 (Process line by line): {time.time() - start_time:.2f} seconds")


# Method 3: Parallel processing by splitting the file
def process_chunk(chunk, output_file):  2 usages  new *
    with open(chunk, "r") as infile, open(output_file, "w") as outfile:
        for line in infile:
            outfile.write(f"{int(line.strip()) * 2}\n")


def method_3():  1 usage  new *
    start_time = time.time()

    # Split file1.txt into two parts
    with open("file1.txt", "r") as f:
        lines = f.readlines()

    mid = len(lines) // 2
    part1, part2 = "file1_part1.txt", "file1_part2.txt"
```

Surprises

Surprisingly, the above results show that Method 3 was not the fastest of the three Python approaches. Intuitively, it seems that parallelizing the splitting process should pay off for larger files, but in practice the single-threaded approaches were fast enough that their simplicity won out over the more complex multi-processing version. This suggests that Python's built-in file handling is already highly optimized for such a simple task as reading and modifying numbers in a file, and parallel processing at this scale is quite unnecessary.

Another aspect that was surprising was the extreme difference between Bash and Python. The Bash script took more than a minute since it reads line by line, modifies, and

appends in a new file, which implies high I/O overhead (Yesmin, F., 2018). While Python handles the operations of a file much better, especially reading great bulks of data. This experiment really reinforces that for tasks involving heavy file manipulation, Python is the far superior choice due to its speed and built-in optimizations.

References

Yesmin, F., (2018) *How to read file line by line in Bash script.* Linux Hint.

https://linuxhint.com/read_file_line_by_line_bash/