The following tables show the running time in milliseconds. I modified the hardcoded file path when compiling and executing the program for each file.

|  |  |  |  |
| --- | --- | --- | --- |
|  | File 1 | File 2 | File 3 |
| 1 Process | 1.06 ms | 4.55 ms | 11.30 ms |
| 2 Processes | 1.03 ms | 3.24 ms | 9.14 ms |
| 4 Processes | 1.15 ms | 2.75 ms | 8.30 ms |

I initially expected the program to run faster every time I increased the number of processes for each file. However, this wasn’t always the case. The most apparent example of this is how the program had a longer execution time when there are more than two child processes for file 1. This is because the overhead CPU cost of creating each process ends up slowing down the program more than speeding it up.

File 2 and file 3, however, didn’t have the same issue that file 1 had when increasing the number of processes. This is because these files are much larger and the time it takes to cover every line outweighs the overhead cost of creating a bunch of child processes. File 1 is too small to gain any advantage from multiprocessing.

The way my program works is by first counting the total number of lines in a file. Then, the program gets the total number of child processes from the user, and it divides the total number of lines for each process, a timer is also started. After that, the program forks into each child process and handles each portion of the file based on the starting line that it’s given and the number of lines that it’s given. Finally, the child processes write their totals to the pipe and the parent process reads and sums up and prints the totals from pipe, the total execution time from the timer is also calculated and printed in microseconds.