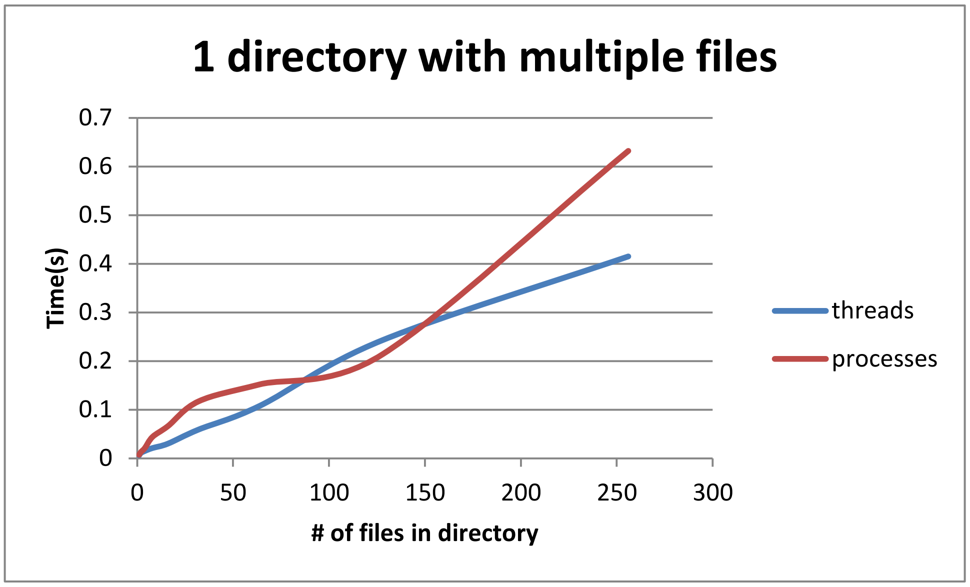
Jon Cobi Delfin and Jeffrey Pham
Systems Programming
Project 2: MultiThreaded CSV sorter

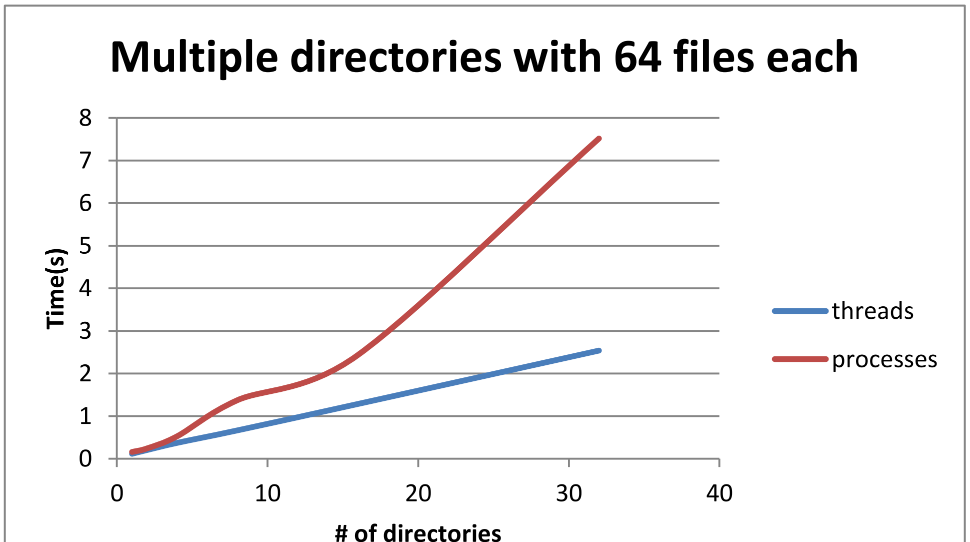
Analysis

Tests and findings:

First test we did was to have 1 directory with multiple files. We started with 1, then did 2, 4, 8, and so on until it contained 256. We ran the tests 10 times for each program and averaged the system time that came out. We found that threads and processes were pretty close up until 256 files, where threads were significantly faster than processes. This makes sense as it's faster for threads to switch from one to another than processes to switch between one and another within a program.



The second test we did was to have multiple directories with 64 files each in them. We ran this test 10 times as well. We only went up to 32 directories as we were doing it manually as opposed to with a script or program, so it took a long time to make the directories and set it all up. This test had a much more significant time difference. Threads were a lot faster than processes. This also makes sense as threads are basically lightweight processes. In Asst1, processes were created for every csv file we ran into, but it still had to run through the rest of the code that traversed through directories and everything, whereas our threads in Asst2 are only called upon to do the function they are needed for, which is to write the csv file to the AllFiles-sorted csv file.



The times between each testing fluctuated a good amount as the tests.

Questions?

Is the comparison between run times a fair one?

Comparison between run times are not necessarily a fair one as the two programs from the assignments do different things. Asst1 does not require the processes to sort a large csv file, which would possibly take a lot of time depending on how large the csv file is. On the other hand, those processes from Asst1 do have to sort each and every individual csv file, while the program from Asst2 does not have to sort them at all and just has to write them to the final csv file. Outputting the metadata for processes and threads are a bit different too, as processes print them as they are created and threads print them at the end once all the threads are done. Additionally, as the number of directories and files go up, it makes sense for processes to need more time since processes are slower overall due to the amount of resources required to fork and execute everything.

What are some reasons for the discrepancies of times or for lack of discrepancies?

The lack of discrepancies occurs at a lower number of files and directories as threads aren't able to take advantage of the fact that they're essentially lightweight processes. The file contents themselves weren't that large, so that contributed towards the lack of discrepancies too. Once the number of files and directories got large enough, the time discrepancies became very apparent as threads were the obviously faster option. Again though, the two programs do two different things in a sense so it's hard to compare.

If there are differences, is it possible to make the slower one faster? How? If there were no differences, it is possible to make one faster than the other? How?

It would be difficult to make the slower one faster as the program itself is very minimal, and since it uses processes it must fork and run through the whole code again to perform its function.

Is mergesort the right option for a multithreaded sorting program? Why or why not?

Mergesort is the right option for the multithreaded sorting program. In our particular case, mergesort is the fastest as our final csv file may have a large amount of unsorted lines and mergesort returns the best time. Even if our program performed sort on each individual csv file, mergesort would still be the best option as combining multiple sorted parts just makes mergesort even faster.