Project 2: ReadMe
Fareen Pourmoussavian
Neel Patel

How to run:
Compile with: $gcc -pthread sorter_thread.c -o sorter

Assumptions made:

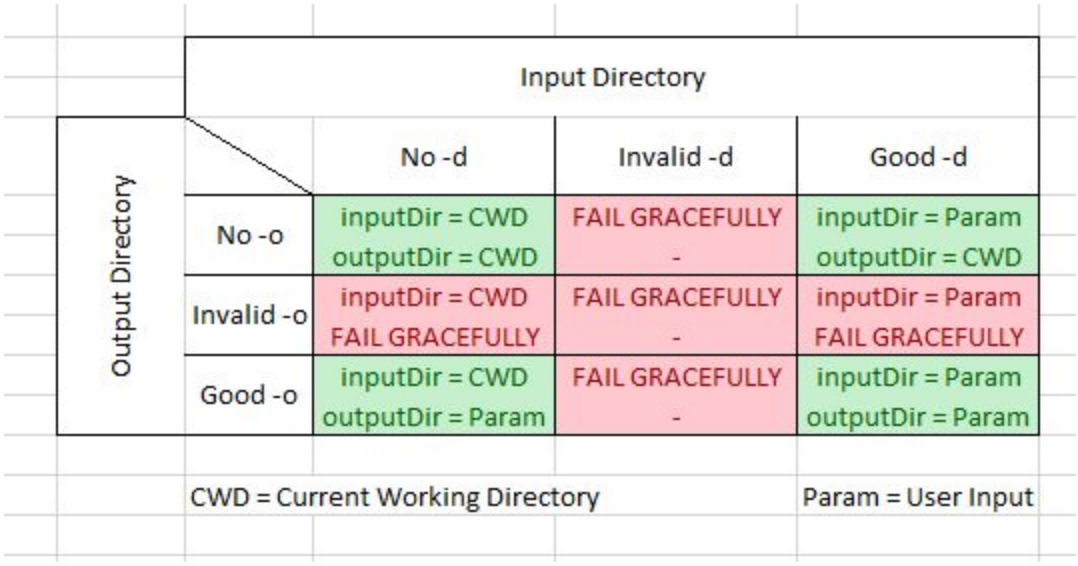
- Professor said on piazza that ALL CSV files will have the SAME headers as movie_metadata. That means 28 columns, same order, same words. Example: if one csv has “duration”, ALL the others will have duration as well. Only the number of rows may vary.
- According to the professor, Any possible mistakes made by our mergesort are forgiven, as we were not provided with a 100% perfect merge sort code and forced to use our old, possibly incorrect one. This is because we were tested on mergesort already, and since we are building on top of the old project, it does not make sense to lose points for something we don’t know how to correct. (We had very minor mistakes according to our grader for project 0)

Error cases/Formatting:

- We cannot print the TIDs properly.
- Once we go over 930 files with about 100 lines each, there is a rare possibility of segfaults in our joinCSV method.
- Segfaults also occur when we have too many threads sometimes, but we are unable to trace how many threads we can handle safely.
- Our CSV output has an empty line between each row. Unfortunately we ran out of time to fix this. Ideally it has something to do with the \0 terminator at the end of the final token in the row.

Part A: Command line Flags

- Our program handles -c -d -o in any order given. It will proceed to work under 4 possible scenarios, given that a -c value is entered. (It will not run without -c)
    - If no -d or -o flags are given, it will assume the current working directory for both input and output of csvs.
    - If one or both are given, it will take the appropriate action based on this table.
    - If the -d and/or -o values are both non-existent, the program will fail gracefully.



Output Directory \ Input Directory	No -d	Invalid -d	Good -d
No -o	inputDir = CWD outputDir = CWD	FAIL GRACEFULLY -	inputDir = Param outputDir = CWD
Invalid -o	inputDir = CWD FAIL GRACEFULLY	FAIL GRACEFULLY -	inputDir = Param FAIL GRACEFULLY
Good -o	inputDir = CWD outputDir = Param	FAIL GRACEFULLY -	inputDir = Param outputDir = Param

CWD = Current Working Directory Param = User Input

Part B: File Structure

- Our program finds all the CSVs and attempts to sort them.
- We might have forgotten to implement a check for the correct format of CSVs, hence the assumption made.
    - The way we would have checked is by reading it in from standard input as always, but this time we compare every header with a static array that makes sure the headers match movie_metadata.csv.
    - We might have forgotten to check for correct format because we were using our own CSV with 10 lines and 8 headers for quick easy checking.

Part C: File Sorting

- Plan
    - The plan was to use multithreads to traverse the directories and sort all the CSV files. After sorting each CSV file, we used a Queue to store them instead of printing to a file like before. Once all of the CSVs are in the queue, we wait for all the threads to return so that we ensure all appropriate CSVs have been accounted for.
    -
- Difficulties

    -encountering a lot of segmentation faults and allocating enough memory for the structs. Learning threads in general was difficult because you had to remember what was happening. It all happens in the same process and they share the same memory so you can run out of memory if you're not careful.

Part D: Meta data

-The PIDs do not print out properly because of a few possible bugs within our threading. We isolated the issues to two possible cases: either bad joining or non-thread safe directory traversal. To make our program faster, we could implement multithreading into our joinCSV() method where we dequeue two csv struct arrays and enqueue a joint form of it.

Part E: Analysis

-The comparison in run times is not a fair one for the following reasons and more:

1. Project 1 is dealing with forking and the other is dealing with threads. One has multiple processes and the other has multiple threads which is a lightweight process. So in forks, it copies everything, but in a lightweight process the memory is shared with the parent. Project 1 just needs to print out the same files just sorted but project 2 needs to combine all these files while locking and unlocking so they go through one at a time. And then we need to sort the big file which consumes a bit of time.
2. Runtimes can only be compared effectively when both sides have the same background resources running. Because of the way the task scheduler works, there is a chance it interrupting or pausing our executions for various higher priority reasons.

-They will have different times because one is dealing with multiple processes that copies everything over and the parent and child in a fork process don't share the same memory, but threads do share the same memory.

-It's possible to make the slower one faster. If there are recursive steps that are taken to traverse directories that could be done iteratively in order to reduce time. Another way to reduce time is to remove any unnecessary print statements, implement a better sorting structure than mergesort, and proper coding techniques that require fewer lines in the Assembly portion of the compilation process.

- Project 2 may also be slower because Project 1 does not deal with joining multiple struct arrays into one large array and call merge sort on it multiple times. All project 1 does is call Mergesort on small files and print it out individually in a directory.

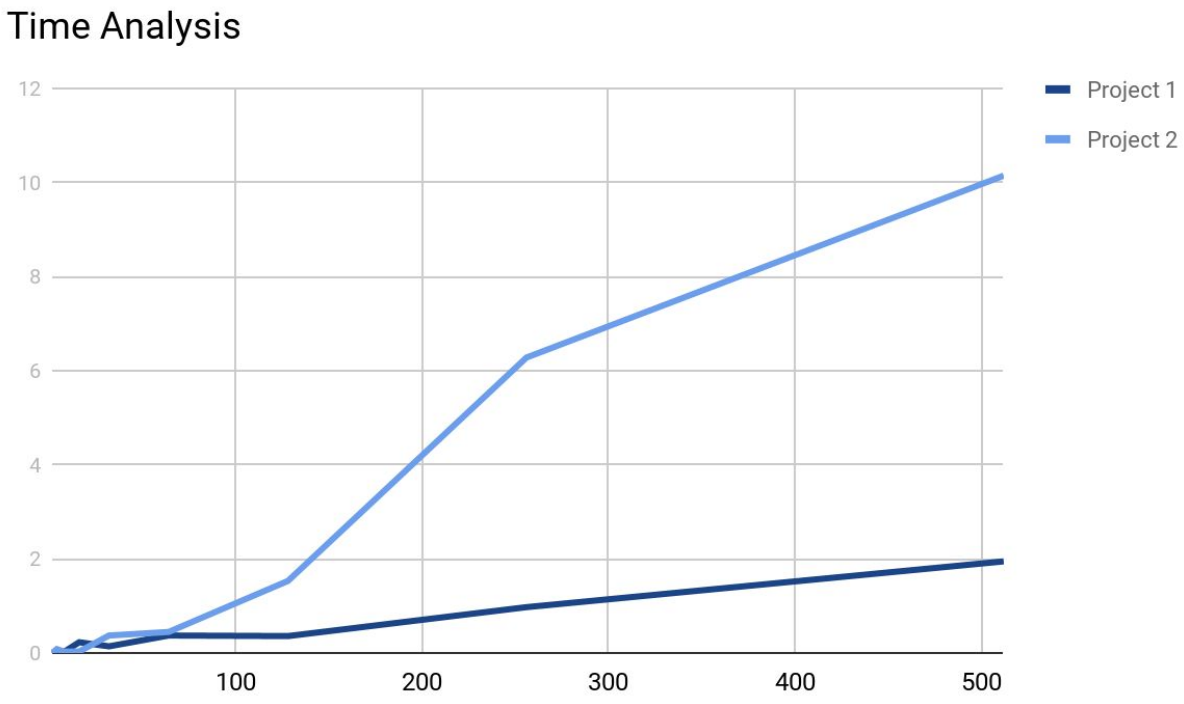
The times are calculated in the following manner:

- 1x to 1024x files are produced in the same directory as the sorter file; 100 rows each, same format as movie_meta, sorted by Gross

-930x csv files are custom with multiple directory traversals and 100 lines per file. Sorted by Gross as well.

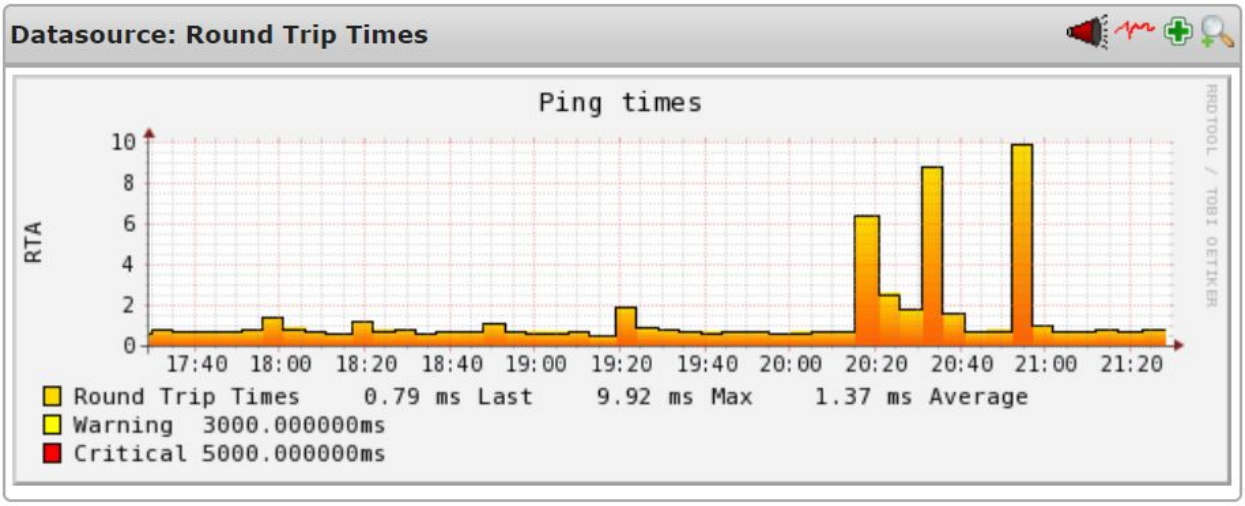
Number of files	Project 1	Project 2
1	0.014	0.012
2	0.022	0.036

4	0.028	0.084
8	0.051	0.081
16	0.074	0.092
32	0.141	0.371
64	0.371	0.447
128	0.359	1.531
256	0.974	6.28
512	1.947	10.145
1024	4.273	8.51
930(custom in depth file)	3.002	7.413



Host: template.cs.rutgers.edu **Service:** Host Perfdata

4 Hours 29.11.17 17:28 - 29.11.17 21:28



Some of these times were calculated during peak times which may have affected our average times values.

Neel was testing on ls.cs.rutgers.edu around 7pm, and for roughly 930 files of our custom directory depth and a total of 93000 lines took only 4.1 seconds max, with multiple averages coming close to 3.8.

However upon further testing during the night, the peak times of the network of iLab machines might have taken a toll on the real wall clock timings.

-Mergesort most likely isn't the fastest sorting algorithm to use for threads. Another sorting algorithm can be run parallel that conserves memory would be better. Maybe other sorting algorithms like quicksort, heapsort, or even radix sort.