Erick Galvez 189005643
Eric Zhang 172005801
Professor Francisco

Asst1 Documentation/ReadMe

Files:

scannerCSVsorter.c

listdir:

Forking Directories

When it came to forking it was hard to understand the concept of it and how to go about it without fork bombing that was our biggest concern. We initially wrote the code so that it only traverses the directory recursively and then implemented forking. Once we had a proper running recursive function all we had to do is right before the function itself recurses we fork a child and have it recurse through the sub directory for us. Then having the parent wait and print out the returned child's pid. At that point we had repeated values for child pid's when printing them out because of the recursive call that goes along with the forking. So to prevent that we used fflush each time we printed a child pid to make sure the stream didn't have repeated values of the child process.

Forking Files

Then we began working on forking when we find a .csv file at first we had to to fork only when it found a .csv file instead of forking each time it finds a file. When we first tried to implement the forking for all .csv files we were getting an infinite loop we had to stop immediately in case it was due to forking which could potentially be a fork bomb. Thankfully we were able to fix it since it was too deep within a loop and if statements. So in the end we were able to have it fork for both cases, each time it finds a sub directory and then each time it encounters a file.

Count of all pids fork()ed and printing pids

After getting that to run properly we then started the next phase of counting the total number of pids. To do this we used a pipe and with the pipe we noticed that if we open it within the traversing function it would create multiple pipes and splitting the total number of processes within each pipe created. So we opened the pipe in the main where the initial process is the only one in the main initially. From there we had the initial process print it's pid in the main after it checks all the flag cases. From there we head back to out recursing and forking function where each time we fork we have the child write one byte into the pipe. When we create a fork we have a depth initialized to zero however that is only the case when it is the initial process since each time we fork and the child process writes to the pipe it increments the counter. So every child will have a depth of 1 or higher. With that being said at the end of the function only when the depth is zero it will close the pipe and then read the total number of bytes written into the pipe and keep count of the number of times read giving the count of all the child processes that wrote into the pipe.

Main:

In the main, the first thing to do was the take the flag/input from the user and interpret it. To account for the user inputting the flags in any order, we set up a for loop to increment a counter starting from 1 by 2 until it was less than the total number of arguments inputted, or argc. From there, we used strcmp to check when -c, -d, or -o were inputted. Each time it evaluated to true, a counter for each flag was incremented by 1. If any were greater than 1, or if -c was counted less than once, then there would be a fatal error in the input format, as it would mean that there were duplicate flags, or that there was no column header name to be sorted on. If the above was not true, then the number of -c was counted would be 1, and -d and -o could either be 1 or 0, which is fine. If either -d or -o was 0, then they would be set to "." automatically, otherwise the main would record the path each one was inputted as to pass onto the listdir function.

scannerCSVsorter.h

This file remained largely the same as it was in the original Asst0 version, but we added a method to check if the CSV file we were looking at was a valid csv file by seeing if the number of columns in the header matched the rest of the rows. This was done by tokenizing the header line, counting the number of tokens it had, and then tokenizing and counting the number of tokens in each other line and comparing the two. If they were different, the file would be deemed as an invalid format for a csv file, and it would be skipped over and not sorted.

modcsvSorter.c

This sorting function remained largely the same as it was in the original Asst0 version, but a big difference was including a way to account for float values and sort them, something we forgot to do in our first submission. In addition for the new material, I passed in the path of the output directory, and used strcat to format the name name of the file to write to using fprintf.

mergesort.c

This .c file remained the same as it was in the original Asst0 version, with the only difference is adding another set of mergesort methods to sort through float values, something we forgot to account for in our first assignment.

<u>Makefile</u>

Finally, We ended with the Makefile. This was really tricky to understand because we didn't know what we needed to include during the compilation for the final sorter since other files would be included on top. It was a pain to figure out why our mergesort.c file gave us an error each time it ran. We did not understand why because it would compile just fine when we ran out assignment 0 code. In the end we came to the conclusion that since it is included in the top of our csvSorter file what it will compile properly along with it since mergesort.c does not contain a main function which was the error being printed.

<u>Test Cases:</u>

While working on the project, we used the movie_metadata.csv file from assignment 1 as our testing file. During the project, we would constantly make new directories/subdirectories and new csv files with different names (usually movie_metadata.csv named to testing1movie_metadata.csv, where testing1 is a new subdirectory). Then, we would output the path to stdout to see if we were properly concatenating for each new path created leading to either a subdirectory or file.
For example, if we inputted:

```
./scannerCSVsorter -c director_name
```

Using the methods explained above, we would obtain:

```
Sorting on: director_name
Searching in directory: .
Outputting in directory: .

Searching path to file: ./movie_metadata.csv
Outputting path to file: ./movie_metadata-sorted-director_name.csv
```

In the context of our program, this information means that the movie_metadata.csv file was found in the current directory, and was made to output a new sorted csv file in the current directory.

Or if we inputted:

```
./scannerCSVsorter -c title_year -d testing1 -o ./testing1/testing2
```

Where testing 1 is a subdirectory of the starting/current directory, and testing2 is a subdirectory of testing1, and movie_metadata.csv being the only csv file in the system located in testing1, using the methods explained above, we would obtain:

```
Sorting on: title_year
Searching in directory: tesing1
Outputting in directory: ./testing1/testing2

Searching path to file: ./testing1/movie_metadata.csv
Outputting path to file: ./testing1/testing2/movie_metadata-sorted-director_name.csv
```

In the context of our program, this information means that the movie_metadata.csv file was found in the testing1 subdirectory of the current directory, and was made to output a new sorted csv file in the testing2 subdriectory of the testing1 subdirectory.