Mitch Lew, Dhruvik Patel

Design Synopsis:

The objective of this assignment is to write a multi-process C program to sort a list of records of movies from IMDb alphabetically by the data in a given column. Utilizing the Fork and File pointers we were able to accomplish this.

Using functions to operate our code we created 3 new functions and edited our old functions from our last project in order to accomplish our task. The 4 functions we created are

1) arghandle
2) handy
3) handleFile
4) sorter

We also created another function that determines how many processes are being run. The function is called:

5) numberOfProcesses

1) **Arghandle**:

Our first objective handle the arguments inputted in the command line. Through main I declared 3 strings: column, directory, and outputdirectory. Column representing the header column that we need to sort in each csv file, directory representing the directory we want to search through and sort the csv files, and outputdirectory representing the directory we wished to output our csv files. I also defaulted the value for directory to current directory or '.'

Next we passed those the values of the three strings into arghandle along with the argc and argv. In arghandle we checked to see how many arguments were inputted into the command line. If they were less than 3 we terminated the program. The executable file, the column identifier –c and the actual column field we need to sort must be inputted for our program to work. Following that we checked to see if any value for column was inputted. If they failed to input an argument for column the program would terminate. Also if they failed to have a proper column heading to sort the file the program would terminate.

We also checked to see if any values for "output directory" and, "directory we are searching through" were inputted and set outputdirectory and directory equal to those values respectively if inputted. Once these conditions are checked we have handled the arguments inputted through command line.

2) Handy: //(HandleDirectory)

After completing the arghandle function we have our values for "directory", "outputdirectory", and "column" so we call the function handy. Handy takes the three strings as arguments: Directory, column, and outputdirectory. Through handy we would use directory pointers and struct dirent's to traverse the file directory and access their entries. First we set our directory pointer to the value of "fopen(directory)". If the directory doesn't exist we fail the program. We then determine in if the files within that directory are directories or a files. If the entry is a directory we rename that directory (original directory inputted in main)/(entry->d_name) and fork a child processes which will recursively call handy so that we can check if there are directories within directories. If that is the case, we will fork a child process every single time we encounter a directory. This allows us the access the file-path to a file and use it to rename our files later so we can access them directly through their strings. If the entry is a regular file type we check to see if it is a csv file and fork a child process that rename the file appended with the file-path and takes it the sorter.

With the check to see if each entry is a file, we check to see if we have a csv file. We do so by check the last 4 letters of the filename. If it is not a csv file we ignore. If it is a csv file we fork a child process that appends the filepath to it and calls our next method, handleFile.

At the end of this method, we put all the PIDs of the child processes into an array, kept a count of how many entries are in the array. We can print the PIDs of each child processes using a for loop and using count as a limit.

3) **handleFile:**

Handle file takes in 3 strings, the new filename, the column name, and the outputdirectory. Once we've reached handle file we point a file pointer to our new filename. We open the csv file and check to see whether the csv file is properly formatted with 28 columns. If not we fail the program. If the file check succeeds we can now sort that csv file. We call the function sorter from handle file.

4) **Sorter:**

Sorter is taken from our first project. Sorter is passed three string arguments. The new filename, the column we wish to sort and the outputdirectory. Here in sorter we added a few check to determine where we should write out the sorted file to. First we check to see if our outputdirectory we passed through all our functions is set to an empty string, because we initialized it to empty string in main. If it is still an empty string than we know arghandle did not set a new value for output directory. In this case we want to write our sorted file into the same directory as the original file. Our filename at this point should already have the path appended to it. The only thing we need to add is –sorted-<column>.csv. we append that along with our filename to a new variable called sortedname and write our sorted file to that new filename.

If an outputdirectory is specified using string manipulation we rename the file outputdirectory/<original file name>-sorted-.csv we then write our merged csv file to this filename.

Following the sort we return to handy and search for additional csv files that are suitable to sort.

5) **numberOfProcesses:**

This method basically does the same thing as handy where it would traverse through the given directory and count how many .csv files are there. It also counts directories within directories and counts that as a process since we need to fork a child process every time the program encounters a directory. The count process also increments when a .csv file is encountered because the program also forks a child process to sort that file. We needed to use a global variable to achieve this task because count would be set to 0 on every loop.

Assumptions: Assumptions we made: All csv files with 28 columns have the proper column headings. If there were more or less than 28 column headings than the csv file would be improper. Also we made the assumption command line arguments can be handled in any order. Anything after the identifier –c, -d, -o are represented as the column heading, directory we are searching through, and outputdirectory respectively. If the user inputted things incorrectly our program fails gracefully.

Difficulties: One of the problems we faced when doing this project is that we had to account for all the different input test cases. The arghandle function takes in all the values put in by the user and determines how the program should be ran. Since it could be in any other, we had to use argv[i] and loop through all the inputs and achieve what we were trying to do. Another difficulty deals is fork(). The concept behind fork() is somewhat hard to understand and how we had to keep track of all the different processes can get overwhelming at times, but we managed to do it. Figuring out when to fork might the hardest thing, but once understanding its concept, it was not that difficult.

Testing procedure: In order to test the code, we took the extreme cases from the movie_metadata.csv file and made a shorter .CSV file. Since we know it worked on the smaller file for all the extreme cases, we inferred that it would work correctly on the larger .CSV file as well.

How to use code:

In the command window. Compile Sorter.C using gcc. We can use–o to rename the a.out to sorter such as:

```
gcc sorter.c –o sorter
```

Here you would just call the executable file along with the input arguments as follows.

```
./sorter –c <column> -d <directory> -o <outputdirectory>
```

-d and –o and what follows is optional, but the column we must sort is mandatory.