David Tian (dt474) & Rahul Boppana (rsb172)

Design:

The first steps we took were to fix any issues with the original program from Assignment 0. For example, we added a check to make sure that every row had the same number of columns as the header. We also changed some malloced strings to arrays and removed their respective free() statements to make things simpler.

Next, we had to find a way to read through directories using C. We used the dirent.h library to open DIR objects on the specified input and output directories. The DIR objects can be iterated with readdir() to generate a directory entity, or dirent, for each file object in the directory. The dirent struct provides all the information we need, including the type of file object and the file name.

The last main topic was processes and forking. We were required to fork when we came across a nested folder within a directory and to fork to sort when we came across a valid csv. A lot of the functions were built into the linux system and c package but it was important to understand how to use them and how to take advantage of their return values as processes cannot communicate with each other. As a general description, every time we forked, we checked which instance we were in (parent vs child) and conducted the respective operations. To avoid fork bombs and the like, we structured the program to wait() every time a fork was called.

Implementation Basic Overview:

All of the sorting aspects of the program were carried over from assignment 0, except the process of sorting is now packaged in a method that gets called whenever a compatible csv is discovered. Our main method is now responsible for checking arguments and setting up initial directories. The program uses getopt() to read in the flags in any order and set the column and directory values. Only the -c option is required; if -d or -o are not specified, the default is to set the directories to current for both. From there the first process ID is printed and we call traverseDir().

TraverseDir() is responsible for all forking operations and returns the total number of new processes created. We check each entry in a directory and fork a process to handle it. If the file object is a directory, the child process recursively calls traverseDir() on that directory by passing in the calculated directory path location. If it is a file, it checks if it is a CSV and calls sortCSV() to sort the file. SortCSV() handles all the checks for format consistency and creates the output file in the correct location.

After every fork, the program is basically split up into 2 areas with instructions specified for the child process and instructions for the parent – mainly to wait on the child and check the exist status. It is important to note that we use the exit status to keep track of the total processes to

be printed in the end. With nested directories and therefore nested child processes, this allows us to keep track of the lower levels and return that data back to the initial execution instance.

Whenever a child is called, the process ID is printed out and on forking errors, the program gracefully halts execution. If a directory value was another directory, the child sets the path to the directory and passes that into the recursive call and the parent waits. If the directory value was a file, the file extension is also checked and the sorting method is called with the file name passed in while the parent again waits for a success or failure status. The global values are set during each fork to where that fork instance should be looking in the file system.

Headerfile:

The headerfile contains a lot of sorting structures and methods carried over from assignment 0. The new additions include some global variables to keep track of the process counts and directory paths/names.

The global variable inputDirPath keeps track of the complete path to the location of the process' input file, as does outputDirPath with respect to output. firstProc is simply for the sake of print formatting so the first process isn't printed with a preceding comma. We also included the global variable columnName to avoid having to pass it between functions.

The helper function endsWith() is a method to check if a string ends with a target suffix. We used this to check file extensions and to see if a file had already been sorted. The declarations for traverseDir() and sortCSV() are also stored here and are defined in scannerCSVsorter.c. All other fields were present in the prior assignment.

Testing Process:

We created a directory called input for testing purposes which contained several layers of subdirectories and our previous test CSV files at different locations. We ran the program with the -d flag set to "input", and then with the full path name, which gave the same results. Running the program without the -d flag parses through the current directory as expected. We then repeated this with the -o flag set to the output folder. For every test we also wrote stdout to a file called stdout.txt to be able to differentiate between stdout and stderr. The current contents of output and stdout.txt are the result of running

```
./scannerCSVsorter -c food -d input -o output > stdout.txt
```

The 9 processes are a result of the 7 files and 2 subdirectories within the input folder. Finally, the program works correctly with the options in any order, and prints a status message and the filename for any errors it encounters.

Instructions for Running:

Please follow the same instructions outline in the Asst1 assignment description. It specifies to run the program using

```
./scannerCSVsorter -c column_name [-d input_directory] [-o output_directory]
```

to run the file once compiled.