Ronak Gandhi

Maxwell Mucha

Andrew Lowe

Kenneth Creer

####READ ME#####

Professor: Andrew Tjang

Possible Client Inputs (sorter_client):

- ./sorter_client -c <column name> -h <hostname>.cs.rutgers.edu -p <port number>
- ./sorter_client -c <column name> -h <hostname>.cs.rutgers.edu -p <port number> -d <input directory>
- ./sorter_client -c <column name> -h <hostname>.cs.rutgers.edu -p <port number> -o <output directory>
- ./sorter_client -c <column name> -h <hostname>.cs.rutgers.edu -p <port number> -d <input directory> -o <output directory>

* where **<column name>** may be any of the following fields:

director_name, num_critic_for_reviews, duration, director_facebook_likes, actor_3_facebook_likes, actor_2_name, actor_1_facebook_likes, gross, genres, actor_1_name, movie_title, num_voted_users, cast_total_facebook_likes, actor_3_name, facenumber_in_poster, plot_keywords, movie_imdb_link, Num_user_for_reviews, language, country, content_rating , budget, title_year, actor_2_facebook_likes, imdb_score, aspect_ratio, movie_facebook_likes

Possible Server Inputs (sorter_server):

- ./sorter_server -p <port number>

IMPLEMENTATION / DESIGN CHOICES:

- Server:
    - Prints out IP addresses for every connection made instead of only each individual client (our program treats each csv file as a connection when dealing with a client)
        - On Piazza, Professor Tjang stated that this is allowed

- Client:
    - Despite not needing to print anything out to STDOUT for the client, we have print statements that create a more friendly/informative interface for the user
        - Project instructions state that we don't necessarily have to print anything out to STDOUT for the client, but we believe that it is more helpful to the user if there are print statements that display the progress of the connections established, files being sent and sorted, etc.

Client (sorter_client.c & sorter_client.h)

Upon receiving the input parameters, the client first connects to the server (via sockets) given the port number and the IP address that is derived from the hostname using the function getaddrinfo(). The client first traverses through either the current directory or given directory, searching for valid csv files. If it finds a valid csv file, it will create a thread and will put the

contents of the csv file into a buffer that will be sent to the server. It will skip over and ignore any non valid csv file within the searched directory. When the client receives the sorted data (it is sent line by line in the form of strings) from the server, it then prints out this data into a single outputted file named “AllFiles-sorted-<fieldname>.csv” in either the output directory provided (if given) or the current directory (if output directory is not given).

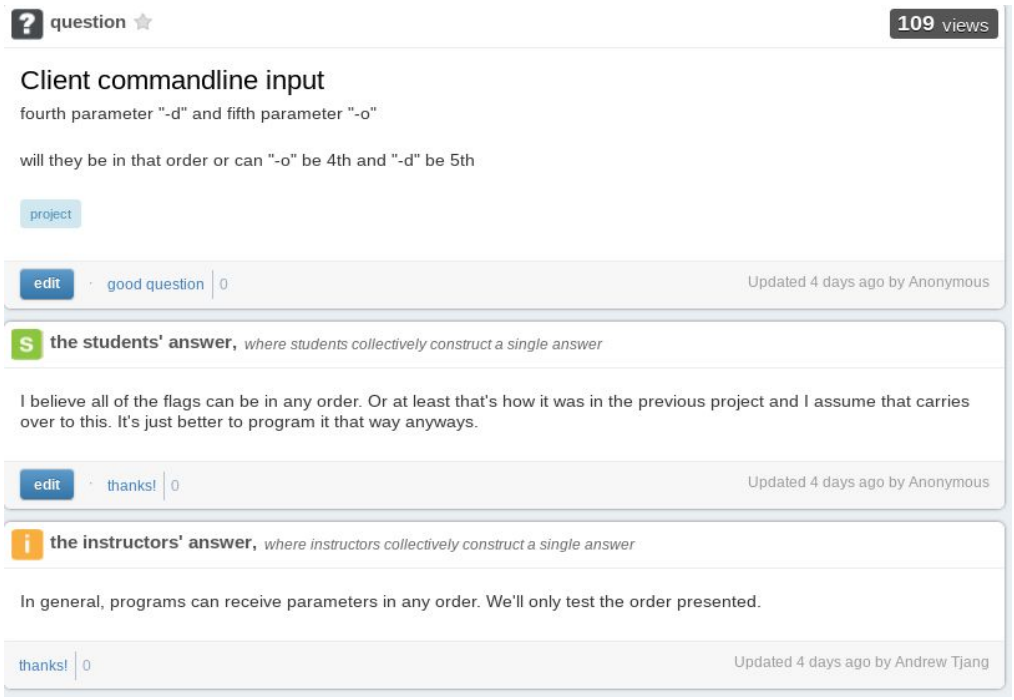
Server (sorter_server.c & sorter_server.h)

Upon receiving the buffer containing the contents of the csv file, the server turns puts this raw data into a char array called data. After the whole csv file is loaded into data(char array), it calls loadArray() which populates the array of movie records(structs). Then it appends this array of movie records to a global array (which locks when copying data from array to global array and when receiving a dump request in which it obtains the data from the global array) where it will be merge sorted in the end. It goes through this process until the traversal in the client is finished. When traversal is completed, the server will wait for a dump request, where once heard, it will sort the global array of appended movie records and send it back to the client in the form of strings. The server also outputs a list of the IP addresses of all the clients that have connected to it. Finally, as per the project’s instructions, the server runs until stopped by a SIGKILL (CTRL + C on keyboard).

Testing/Debugging

Upon testing, always run server before running client, so that the client can properly connect to a running server. If input parameters are incorrect for either client or server, program will give an error message and exit. Input and output directories may be either relative or absolute paths, but if the directories contain spaces in either folder or file names, they must be

surrounded by quotations in order for argc and argv to function properly (our program will remove the quotes to access the directories).



- - - - - - PAST ASSIGNMENTS - - - - - -

--PROJECT 2 INFO---

Design Process:

After receiving all possible given parameters to the program, we initiate by making a thread to start looking through directories and spawn new threads if we see a directory or csv

file. To spawn a new thread on a directory, we made a void * getFile function that can we can recursively call to spawn threads on each directory we saw. Additionally, we used a DIR pointer similar to what was used in Part 1 to traverse through the directories. If we saw a csv file, I would call a void * sortFile function that received a file path, then would open that file and mergesort on the file. Since the project notes all the csv files must be merged into 1 singular csv file in the end, we made a global array and combined the newly sorted array into the global array at the end of each void * sortFile function, and called mergesort in the end, ensuring that all of the records in the global array was sorted in the end.

Additionally, at the end of the sortFile and getFile functions, I wrote a pthread_exit(0); to exit out of the thread after it was finished to return to the calling thread. Additionally, I print out all of the TIDs as i end the threads. Also at the end of void * getFile, I wrote a loop that calls pthread_join to wait on all of the tids of the threads made (Note: I made an array of pthread_t tids so I can store and refer to the tids of all of the threads when joining and spawning threads). At the end when I have one big global array containing all of the combined csvs in sorted order, I insert them into an appropriate output File if specified otherwise, in the same directory as sorter_thread.c if not given an output File.

--PROJECT 1 INFO---

Design Process:

We use a loop to handle forking. The loop works by opening a directory, and going to the next item (skipping . and ..), and forking on each file to process it. In the case of directories, the forked child reopens the new directory and runs through the loop. In the case of .csv files, they are sent into the sorter. The output of our sorter was slightly changed, instead of using

STDOUT, we instead write to a newly created file buffer. Waits are handled by counting the number of valid children of the starting directory, and the children of every directory below that, then waiting for each child. These subdirectories also return their number of children, which is how we count the value of PIDS. We reject all improperly formatted arguments. We skip all files with a "-sorted-" in the name. Sorter.h contains the struct needed to store our movie data, as well as a number of function declarations. The structs and functions are the ones that are called in both files, sorter.c and mergesort.c like mergesort or mergesortHelper, or ones that could be useful for other projects, such as trim, compareStrings, or printCSV.

---PROJECT 0 INFO---

Design Process:

We chose to use an array of struct pointers to store our data, as this made for an easier time swapping the arrays. We chose to use strcasecmp, as using strcmp put lowercase letters after all uppercase letters, which put them out of dictionary order place.Memory leak free according to valgrind. We reject all improperly formatted arguments. All empty number fields are set as "-1", to differentiate between an empty input value, and a value of 0. All empty strings are left as is.