1. Program explanation

This program runs in a multi-threaded fashion.

Thread is generated when search the below folder and sort each csv file

Command line arguments is the same with original program.

Searched each file is converted to the file sorted along with a csv file's field name.

If input folder is not provided then search all files in execution file's location directory.

If or not then search all files in indicated directory.

This program converts original csv file to a single sorted csv file along with indicated field name and displays process id and all ids of the threads and the number of the threads.

1)Program user guide

Code will read in a set of flags via the command line. They are '-c', '-d' and '-o'.

'-c' indicates sorting by a column. The files read in by your code should be sorted by the column name that immediately follows '-c'. This flag is required. If it is not present, your code should print an error message, usage information and return.

'-d' indicates a starting directory. The program will start at this directory name immediately following '-d' to look for CSV files to sort. This flag is optional. If this flag is not present, the default behavior of your code should be to start searching at the current directory.

'-o' indicates the output directory for the sorted file. The program will store the sorted file to the directory name immediately following the '-o'. This flag is optional. If this flag is not present, the default behavior of your code should be to store the sorted file in the current directory.

These flags may be defined in any order. For instance:

```
$ ./sorter -c  movie_title -d thisdir -o thatdir
$ ./sorter   -d thisdir -o thatdir -c  movie_title
$ ./sorter   -d thisdir  -c  movie_title -o thatdir
```

Code's output will be single CSV file outputted to a file named: AllFiles-sorted-<fieldname>.csv.

NAME

sorter - sorts all csv files in a directory lexicographically and outputs

SYNOPSIS

sorter -c [ARGUMENT] -d [DIRIN] -o [DIROUT]

sorter -c [ARGUMENT] -d [DIRIN]

sorter -c [ARGUMENT] -o [DIROUT]

sorter -c [ARGUMENT]

DESCRIPTION

Sorts all text files in an optional [DIRIN] with 28 specific header columns lexicographically by passing the name of the column as an argument, the sorted file are outputted as [FILE]-sorted-[ARGUMENT] in an optional [DIROUT] directory. Default [DIRIN] is the current directory, default [DIROUT] is the source file. Prints initial TID, all child TID, and total number of TIDs to stdout.

OPTIONS

-c color

Sorts by color of movie.

-c director_name

Sorts by director name alphabetically.

-c num_critic_for_reviews

Sorts by the amount of critics who reviewed the movie.

-c duration

Sorts by runtime.

-c director_facebook_likes

Sorts by amount of faceboook likes for the movie.

-c actor_3_facebook_likes

Sorts by amount of facebook likes for main actor 3.

-c actor_2_name

Sorts by main actor 2 name alphabetically

-c actor_1_facebook_likes

Sorts by amount of facebook likes for main actor 1.

-c gross

Sorts by gross income.

-c genres

Sorts alphabetically by genre.

-c actor_1_name

Sorts by main actor name alphabetically.

-c movie_title

Sorts alphabetically by movie title.

-c num_voted_users

Sorts by number of user votes.

-c cast_total_facebook_likes

Sorts by total cast facebook likes.

-c actor_3_name

Sorts by main actor 3 name alphabetically.

-c facenumber_in_poster

Sorts by number of faces in poster.

-c plot_keywords

Sorts alphabetically by plot keywords.

-c movie_imdb_link

Sorts alphabetically by URL.

-c num_user_for_reviews

Sorts by number of user views.

-c language

Sorts alphabetically by language.

-c country

Sorts alphabetically by country.

-c content_rating

Sorts alphabetically by MPAA rating.

-c budget

Sorts by budget.

-c title_year

Sorts by release year.

-c actor_2_facebook_likes

Sorts by amount of facebook likes of main actor 2.

-c imdb_score

Sorts by IMDB score of the movie.

-c aspect_ratio

Sorts by aspect ratio of the movie.

-c movie_facebook_likes

Sorts by likes on facebook page of movie.

Header file:

the header file keeps track of of the members of each function.

It was easy to organize every variable based on the function it was being used in.

It makes the variables more readable and less arbitrary because you know the significance of them. It also initialized the arrays for the headers.

the header had the static variable to keep track of the process count

---

Design:

A lot of the code was reused from the previous projects. However, I had to make a lot of changes as my project 1 did not pass many of the tests. My code assigns a tag to the column that the user wants to sort when it is provided from the command line. I create a struct from thread_args and assign the input directory, titles, and column numbers to it. I then create a thread, make print statements to print them

out using pthread_self() and implement pthread_create(&tid, NULL, process_csv, args) where process_csv is the function that does the sorting. In function sort_csv, I lock the thread and then print it.

sort_csv function:

If there is no file, I unlock it. I then check if the file is an IMDB file by comparing the columns to what they should be. If they are what they should be, I unlock the thread and return. After that I prepare the allfiles-sorted-< >.csv name. Next, parse the file how I did in the previous projects, trimming whitespace, trimming quotes. I then assigned a token to each header to be sorted later after setting that token equal to the member in the struct. (Note: all the headers are stored in a struct). I create a new struct and set it equal to the first struct. All of this is done through a while loop. After the loop parses the file until the end of it, I close the file. I create a new struct called sortCol and mergesort the original struct based on the column number. I then create a new csv with the appropriate name.

printData function:

I print everything in the csv file by calling my print data function. To do this I create a new struct and print out the headers. Since the struct colSorted is taken as an argument, I transfer the contents from that struct to the new struct titled printData in order to populate the new csv file.

proccess_csv function:

This is where I traverse through the directories. I first make a while loop with the condition that the directory isn't empty. We continue if the file is a .csv file. I implement pthread_create(&tid, NULL, sort_csv, s_args) where sort_csv is the function I previously described and s_args is the column number. I then do pthread_join(tid, NULL). If were are handling a directory and not a csv file, we open the directory, and do pthread_create(&tid, NULL, process_csv, argsl) where argsl is rhe new directory. I print out the TID for that and then do pthread_join(tid, NULL). Else: the current file is not a csv file. Close the while loop and then we exit

Challenges:

Keeping track of locking and unlocking threads was key. Figuring out how multiple threads worked together was the hardest part. I also struggled with dealing with incompatible csv files and non csv files which is something I failed to do last time. Making sure I could output the sorted csv file to any folder was something I had to make sure that worked as well. And most of all, optimization. Trying to get the program to be as fast as it can was frustrating because this is the first multithreading assignment I've ever done, so I am not directly familiar with speed and optimization of it.