CS 214 Systems Programming
Daniel Chen, danieche
Sharon Lee, sal285

Assignment 2
Sorting CSVs with Multithreading

Introduction

Previously in Assignment1, we took advantage of the computer's multiple CPU cores to conduct concurrent multiprocess sorting on CSV files using *fork()* and *wait()*. In Assignment2, we modified our sorter to use concurrent sorting using threads, *pthread_create()* and *pthread_join()* in particular. This allows us to work within a single process, working in the same memory and address space as other threads. This lead to a few issues we had to watch for when writing our code, particularly synchronization between different threads. To counter this, we used mutex locks to ensure that common data being used was serially processed in order to prevent race conditions and locks.

Design

The way the sorter works for Assignment2 is that it will search through the current directory and all subdirectories searching for CSV files. If the CSV file matches the IMDB-type file we would like to sort by, it will add its data into an array common to all threads. Upon finishing searching all CSV files with the correct format, the program will pass the data structure to a mergesort function that will sort the array and then write the final result to a single CSV. Threads are created upon each time the program encounters a new directory, and child threads are created upon each time a new subdirectory and valid CSV is found within that directory. Thread IDs are tracked by using a linked list that records the parent and child IDs within the stack frame. Upon successful completion, the program will display the initial thread ID, thread IDs of all child threads, and the total number of threads.

Assumptions

We assume that every csv must be ended with a newline

Sorter.h

Sorter.h contains the function and struct definitions used for our program. The three structs ar defined as below:

```
typedef struct{
    char *field_data;
    char *original_row;
} Record;
```

Struct Record the fundamental element of our sorted data, where field_data holds the item to be sorted, and original_row holds the original string.

```
typedef struct{
    char *sortTopic;
    char *currPath;
    char *outputDir;
} InputParams;
```
Struct InputParams holds fields from the user input parameters such as category, input path, and output path.

```
typedef struct node{
    char *path; //current path
    char *filename; //current filename if csv
    pthread_t p_tid; //parent thread tid
    pthread_t c_tid; //child thread tid
    struct node *next;
    InputParams * input;
    SortParams *sortInput;
} Node;
```
Struct Node holds the fields of a node for our linked list.

Most functions are written to clearly describe their purpose. Some of importance that are new from Assignment 1 include these:

```
Node * init(Node *next, char*filename, char *path, pthread_t p_tid, pthread_t
c_tid,InputParams * input,SortParams* sortInput);
```
This function is used to create a new node for a thread linked list in order to track parent and child IDs.

```
Node* append(Node *add,Node *head);
```
This function is used to append a new node to the linked list upon creation of a child thread.

```
int count(Node *head);
```
This function returns the number of nodes within a given linked list.

```
Node *removeFront (Node *head);
```
This function is used to remove the head of a linked list and return the next node as head.

```
void push(Record ** csvData,int index);
```
This function is used to lock the central data structure and push a node into the central data structure.

Instructions

In order to run our program, first make sure sorter_thread.c, mergesort.c, sorter.h are within the same directory and compile with the command:

gcc sorter_thread.c -o sorter -lpthread

This will create a binary “sorter” executable that will be run. To run, you can input multiple flags in any order that will dictate the category to sort by, the directory to start sorting in, and the output directory to place the sorted combined file.

./sorter -c <category topic> -d <directory path> -o <directory path>

Category (mandatory):	-c <category topic>
Start Directory (optional):	-d <directory path>
Output Directory (optional):	-o <directory path>

If no input and/or output directory is provided, the default location will be in the same directory as the program itself. If an invalid category is provided, the program will gracefully close after notifying the user.

Output

Note that the output is printing out metadata for each thread; each thread outputs the number of threads it created along with their threadIDs (labeled as Initial PID). At the beginning of the program,Also, at the end of the program, a global count of the total number of threads spawned over the entirety of the program is printed.

Each thread prints out metadata in the following format:

```
 Initial PID: XXXXX

                TIDS of all child threads: AAA,BBB,CCC,DDD,EEE,FFF
(... etc.)

                Total number of threads: ZZZZZ
```

Program sample output:

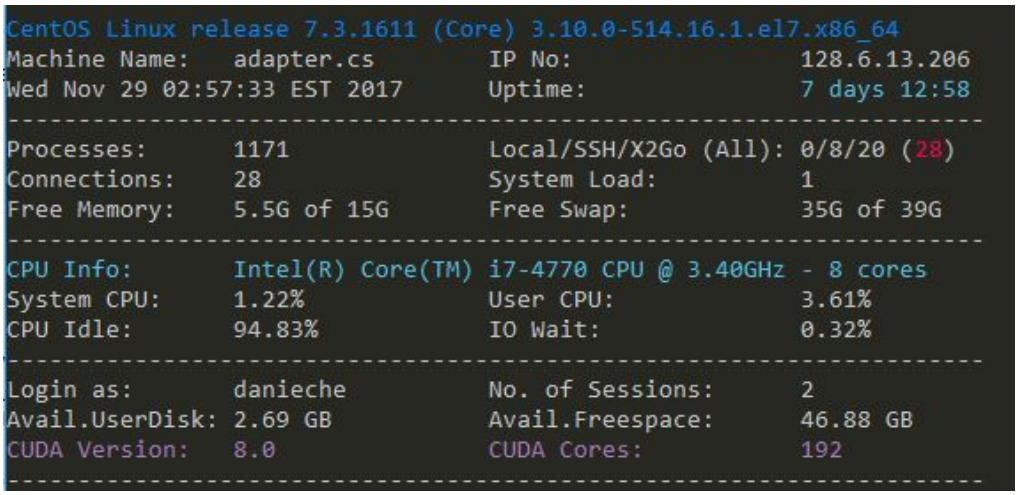
```
[centos@n7225-owv122c06 Code]$ ./sorter -c movie_facebook_likes
MAIN PID:140089761556288

INITIAL PID:813262656
TIDS of all child threads: 140089753200384,
Total number of threads:1

Total number of threads at program end:2
```

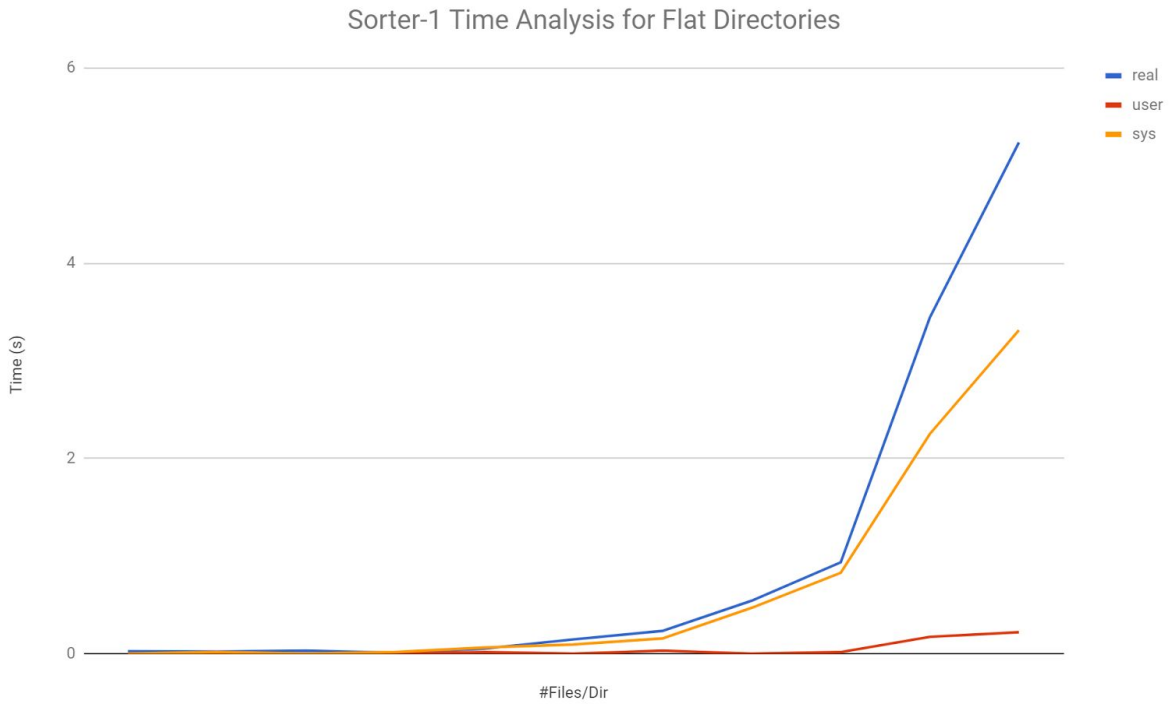
Analysis and Results

Attached are comparisons of the run times between Sorter-1 (multiprocessing) and Sorter-2 (multithreading) that have been timed using the *time* command before the run command. Some important factors that could have impacted our results are the speed of the drives being run (whether HDD or SSD), the processor (cores for Sorter-1, threads available for Sorter-2), the operating system, and even potentially the number of other users on the server could potentially factor into the results we have. In this case, we have been using the Rutgers iLabs machines which use an Intel i7-4770 CPU.



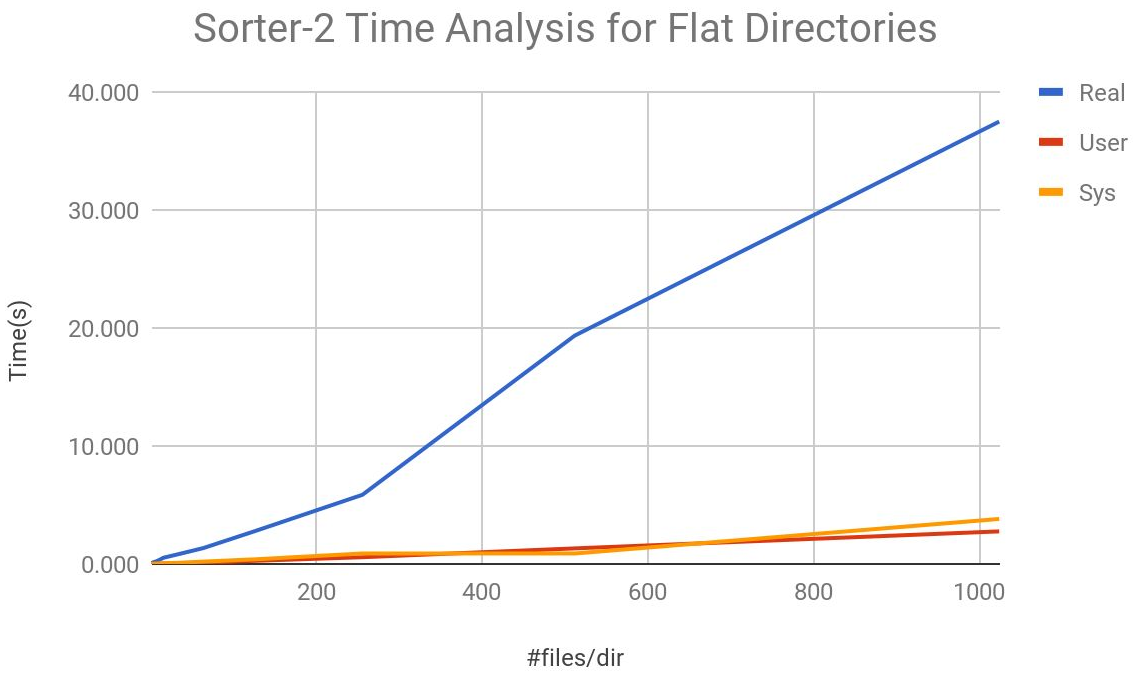
The results below for Sorter-1 are averages of three run times on the directory containing the specified number of files (i.e. 1,2,4,8….1024 files in one directory) in a flat directory.

#files/dir	Real	User	Sys
1	0.024	0.000	0.000
2	0.022	0.000	0.016
4	0.030	0.000	0.000
8	0.009	0.000	0.016
16	0.051	0.016	0.063
32	0.146	0.000	0.094
64	0.233	0.031	0.156
128	0.542	0.000	0.469
256	0.935	0.016	0.828
512	3.442	0.172	2.250
1024	5.235	0.219	3.313

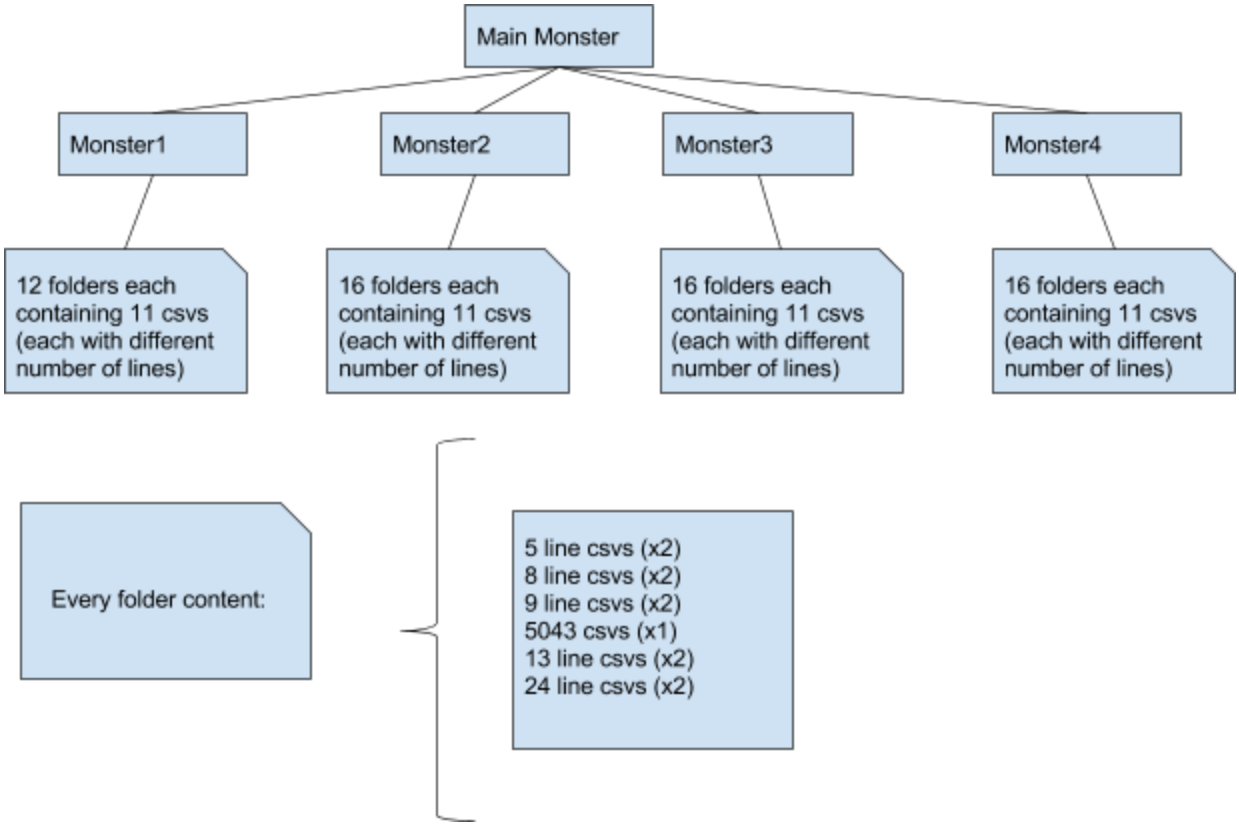


The results below for Sorter-2 are averages of three run times on the directory containing the specified number of files in a flat directory.

Sorter-2	Average Time(s)		
# files/dir	Real	User	Sys
1	0.035	0.002	0.006
2	0.075	0.004	0.014
4	0.150	0.010	0.021
8	0.237	0.012	0.032
16	0.526	0.026	0.074
32	0.796	0.062	0.092
64	1.344	0.102	0.200
128	2.822	0.272	0.399
256	5.862	0.572	0.887
512	19.347	1.320	0.887
1024	37.482	2.760	3.812



Main_Monster directory structure (725 files)



Results from running Main_Monster folder with Sorter-1:

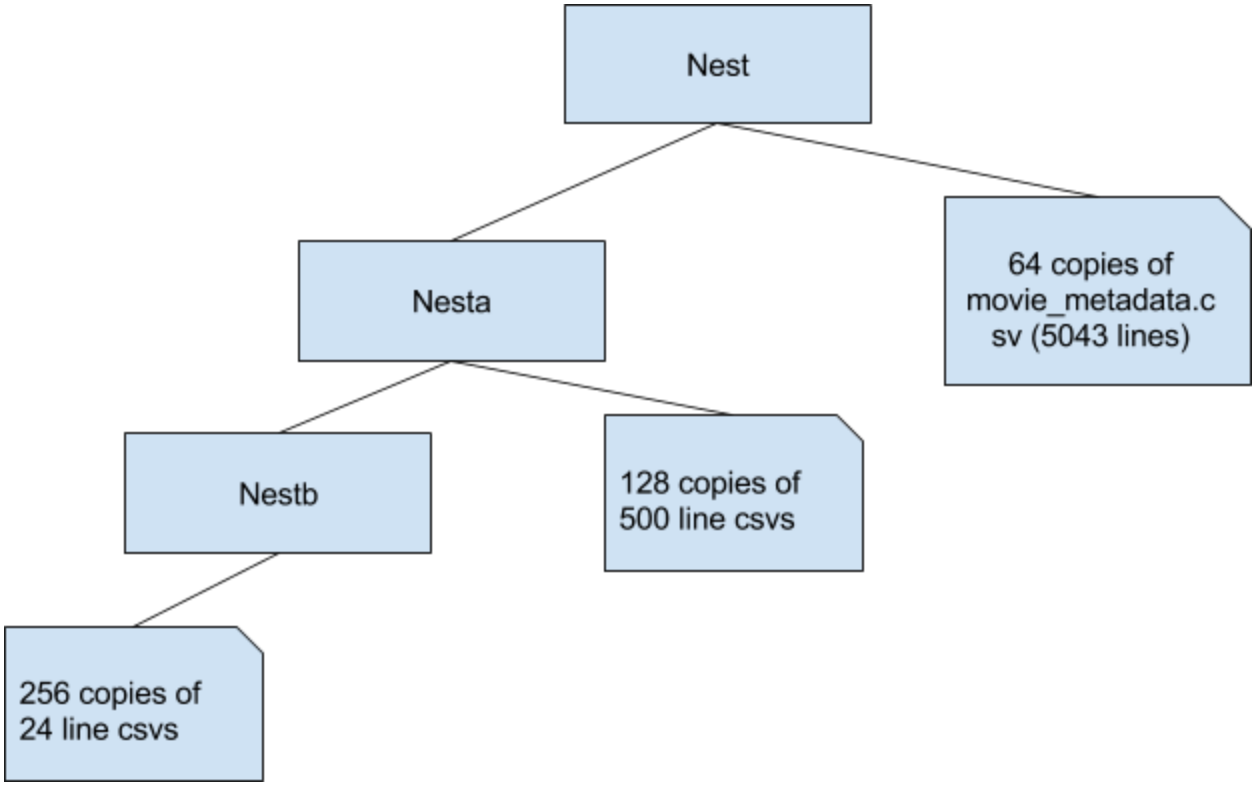
Sorter1					
Main_Monster					
Trial	1	2	3	4	
real	3.413	3.436	2.994	2.03	2.96825
user	2.525	3.203	3.058	3.282	3.017
sys	1.108	0.734	1.173	0.749	0.941

Results from running Main_Monster folder with Sorter-2:

Sorter2					
Main Monster					
Trial	1	2	3	4	Average
real	16.416	15.373	18.15	12.471	15.6025

user	1.196	1.119	1.136	1.106	1.13925
sys	1.096	1.26	1.466	1.113	1.23375

Nest Folder Directory structure (451 files)



Results from running Nest folder with Sorter-1:

Sorter1					
Nest					
Trial	1	2	3	4	Avg
real	3.379	3.601	2.774	2.603	3.08925
user	3.837	3.875	3.958	3.967	3.90925
sys	1.329	1.336	0.644	0.631	0.985

Results from running Nest folder with Sorter-2:

451 files in total					
Sorter2					
Nest					

Trial	1	2	3	4	Avg
real	20.232	4.14	4.388	3.488	8.062
user	1.451	1.45	1.568	1.208	1.41925
sys	0.932	0.604	0.655	0.562	0.68825

Analysis questions:

Q: Is the comparison between run times a fair one? Why or why not?

A: Intrinsically, the comparison between the run times is not a fair one because the assignment outputs are different (in other words, it would be a fairer comparison if both programs were outputting to a centralized sorted file). Based on the algorithm of Sorter-2, there is additional overhead of compiling everything into a centralized data structure (where the pointers of the centralized array are pointed to the contents of the array created to store a single csv's data), sorting all the contents of the data structure after all of the rows are entered into the data structure, and transferring all of the csv data into an overall output file. The main sources for this discrepancy in time is explain in the subsequent question.

Q: What are some reasons for the discrepancies of the times or for lack of discrepancies?

A: One of the largest differences between multithreading and multiprocessing is that multiprocessing creates its own processes with its own resources and memory, whereas multithreading creates threads that share resources within a process. If two threads share the same data structure or resource, the threads have to serialize their access. In Sorter-2, each discovered valid CSV is added to a centralized array on the heap of type Record** which is a serialized process. In Sorter-1, each file is individually sorted in its own process and outputted, which is closer to a true concurrent process.

Q: (a) If there are differences, is it possible to make the slower one faster? How?
(b) If there were no differences, is it possible to make one faster than the other? How?

A: The multiprocess sorter, Sorter-1, is faster than the multithreading sorter, Sorter-2. This is due to the way Sorter-1 and Sorter-2 were required to sort the CSVs and output the results. In Sorter-1, each CSV would give its own sorted CSV file upon completion, whereas Sorter-2 had to combine the sorted output of all CSVs into a single CSV file. In Sorter-2, all CSV data was stashed within a data structure that was sorted at the end. This created threads for each new directory and CSV, which had to be processed linearly. Using a thread pool, we can reduce the overhead of adding to the combined data structure. Another solution would be to have threads work within multiple stacks, which would isolate threads from potential sync conflicts with each other until sorting the combined data structure in the end using semaphores.

Q: Is mergesort the right option for a multithreaded sorting program? Why or why not?
A: Mergesort is the correct option for a multithreaded program in this specific scenario (writing to a single file) because it retains its run time of O(n-logn) when acted upon a single array. This is because the bottleneck is the serialized combination of thread data into the combined data structure, which can not be avoided through other sorting methods.

Difficulties

Some of the difficulties we faced was the synchronization on our previous Sorter-1. Due to some previous assumptions, neither of our Sorter-1 worked for both flat and nested folders. In order to resolve this, we used each Sorter-1 that worked for their respective purpose and timed those outputs for comparison. This is okay for our analysis because we are timing the computation, which already is skewed based on how the Sorter-1 and Sorter-2 work fundamentally. It suffices however as a tool for comparing multiprocessing and multithreading given their originally designed purposes.