Vineeth Puli
Abdullah Shareef 11/29
Systems Project 2: Multithreaded Sort

**Design:**
We designed our multithreaded sort of the movie data CSV files by keeping much of our logic from our multiprocess sort, but instead of creating a new process every time we reach a new directory or CSV, we create a new thread with the respective directory or CSV function. If it is a directory, we go through all the files in the directory and repeat the same process, and if it is a CSV, we create a thread with the CSV function, which sorts that individual CSV and prints the output to that file itself. In addition, we use a mutex locked critical section of code which places the name of the sorted CSV file in a global array and increments the global counter of the total number of threads.

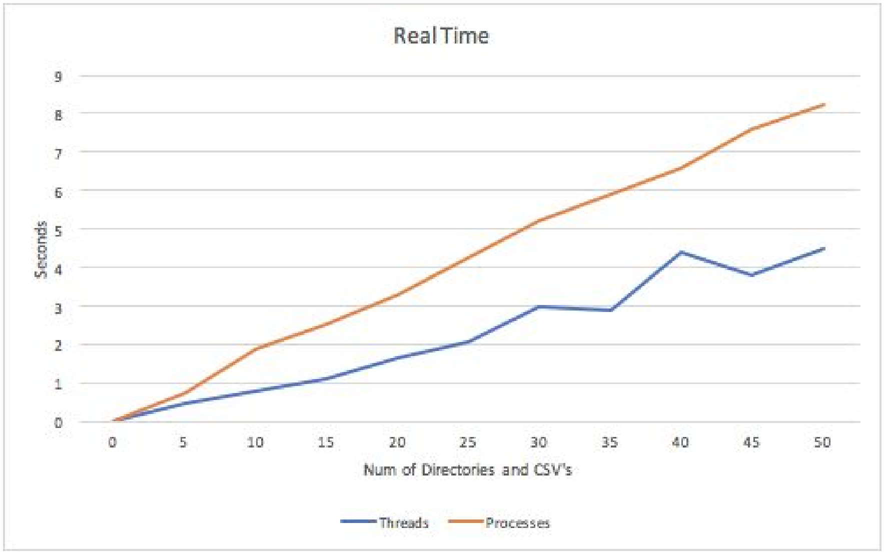
After all the files in each directory are checked, we join all the threads that were created, whose TIDs we save in an array. In the main method after all the threads are joined, we then call a function called megasort, which creates a linked list of movie structs. We use a while loop to go through each sorted file saved in the global array, and add the movies in each file in order in the sorted linked list. After all the files are traversed, we simply print out the movies in the sorted linked list to the output file in its proper output directory.

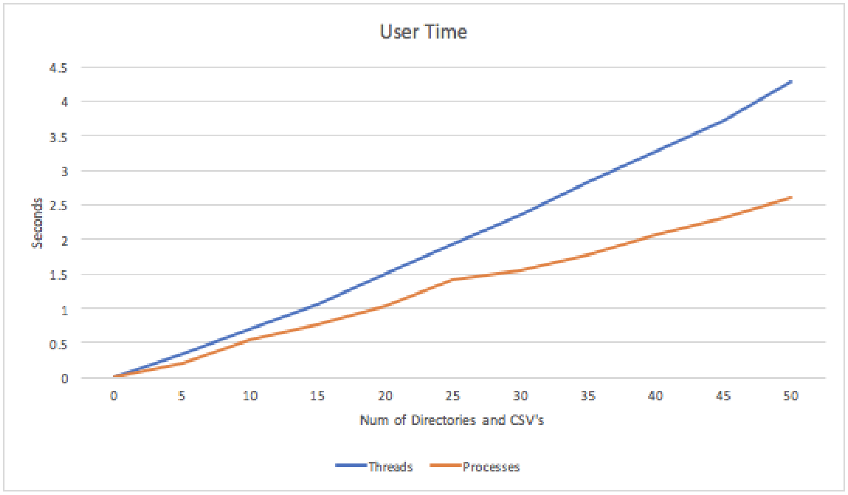
**Assumptions**
We assumed that the fastest way to achieve the final sorted output file would be to sort each file in its own thread, and then merge them at the end in the main thread, as opposed to merging during the threading as it would require more locking.

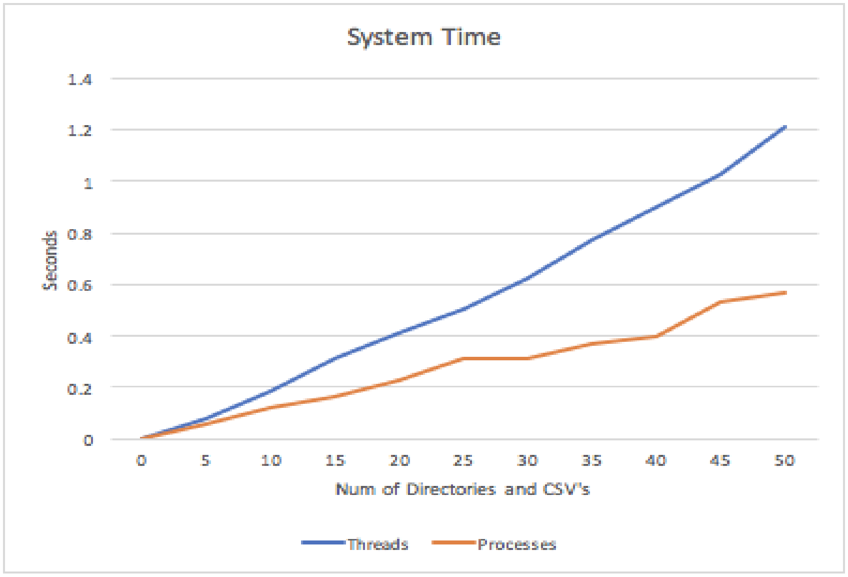
**Difficulties**
The difficult part of this project was not actually the threading, since it was pretty similar to the design of multiprocess sorting, but it was figuring out the optimal way to combine each sorted file into a large data structure. We finally ended up choosing a linked list as it was the simplest way and would require the least memory reallocation.

**Testing**
We tested by creating a Bash script which creates an inputted number of directories, copies an initially chosen CSV into each directory an inputted number of times, and sorts the inputted directory on the inputted sort value. We tested on increments of 5 files and directories upto 50 files and directories, and tracked our system time, user time, and real time, for each run, which we displayed in graphs in the following results section.

## Results: Multithreaded sort vs Multiprocess sort







## **Analysis**:

Upon analysis of our run times, we found that the real time was faster for threaded sort, but the user and system times were faster for the multiprocess sort. This comparison is not exactly fair, due to the fact that our multithreaded sort was required to do an extra action not found in the multiprocess sort, which was to merge all the sorted files into one large output file. This obviously added to the computations that were required to be done.

Threading's time taken was on a stronger upward trend than processing. And than is probably because the time taken to spin up a thread is simply a significant amount more than it takes to create a process. Our hypothesis is that the task that we coded wasn't that memory intensive so the act of copy the entire stack and heap to create processes wasn't that expensive vs creating a new thread. But we could see how as the task requires more and more memory that the action to create a new process would take longer than the threads.

Though we never implement threading inside the actual mergesort function. We held all the entries in a file in a linked list and therefore mergesort was the simplest to implement. From a threading perspective mergesort could in theory have the recursive steps happen in different thread which at a superficial glance seems like it would boost performance. But as we have seen from our analysis that the cost of creating a thread can be significant and bring down an algorithms overall performance if used recklessly.