Multi-Threaded Sorter

Kislaya Joshi (Sec. F4) & Shreya Bajpai (Sec. F2)
Professor Tjang
Project 2

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

        real: 0mXXXs
        user: 0mYYYs
        sys: 0mZZZs
```

The comparison between runtimes on my machine and on the iLab machines is different. For example, we get an average runtime that is faster than the average runtime on the iLab machines. This might be due to the number of users present on the machine, the programs they are running, and of course, the specifications of the machine itself. While there might be discrepancies in the run times, they become normalized when we take the average (despite there being obvious outliers skewing our real runtime data).

What we expected was generally true as evidenced by the runtimes. For example, when we had long subdirectory chains with many files, it took slightly longer, possibly due to the amount of threads that had to be waited on, the size of the files, and the number of files in each subdirectory. It might be possible to make the slower one faster by decreasing the amount of files or the data in those files. Possibly, those who used many locks will be facing the overhead it requires to create and destroy our locks.

We did not change our sorting algorithm. We read a paper (Bozidar & Dobravec, 2015) in which researchers performed a comparison of parallel sorting algorithms. The featured sorting algorithms are the following: *bitonic sort, multistep bitonic sort, IBR bitonic sort,* ***merge sort****, quicksort, radix sort,* and *sample sort.* We learned more about mergesort (divide and conquer as opposed to the bitonic sorts), specifically that if the algorithm for merging sorted sequences is stable, then the whole algorithm is also stable. We decided not to change our sorting algorithm as we started facing complexities in establishing multi-threading. However, we learned a lot and it was well worthwhile!

***Multi-processing* vs. *Multi-threading*.** To understand the “best choice”, we have to consider our program and the workload it requires. In a **multi-threaded** program, there is *less overhead* to create and terminate a thread than a process because little memory copying is required (since threads share all memory, except their respective thread stack).There is also faster task-switching when using a thread, since the active CPU can switch between different processes (CPU caches + program context can be maintained b/w threads in a process, rather than being reloaded in case switching must occur). The obvious benefit to using threads in our assignment is the data sharing that occurs with other threads since all our threads share a pool

of memory (our record * *movies*). There are **disadvantages** to multi-threading as well. For example, the synchronization overhead of shared data requires us to implement a mutex lock to ensure that data from multiple CSV files isn't being read while written nor written by multiple threads at the same time. Since we have a shared process memory space, if something goes wrong in one thread, it can impact data corruption of access violation in others and leads to complete corruption. The *hardest* thing is debugging (*_*) since synchronization issues and accidental data corruption can cause issues that we never would have imagined if working with a single-threaded program.

Upon doing more research as to why we got lower average runtimes in a multi-threaded implementation of our basic data sorter as opposed to our multi-process data sorter, I (Shreya) found that processes are typically beneficial for workloads where tasks take significant computing power, memory, or both (e.g., rendering or printing complicated file formats, PDF).

Implementation.

In our implementation we first parse through the stdin and compare the flags set by the user, if an incorrect flag has been set then an error message is displayed.
After we get the flag parameters we create a pointer to a number of pthreads which would store all the thread ID's as they are created, this is a global variable "allThreads" . We iterate through this array of thread ID's using a "threadIndex" that is also a global variable and appends as a new thread is created either when a subdirectory is found or when a valid csv file is found.

We figured out that to pass parameters to void * functions using threads we would need to create a struct of the parameters and pass a pointer to the structs as a parameter to our threaded functions, we store all the values of our parameters in a global array of type struct variables which contains the necessary parameters for directory and the sorter. The main parameter here was the path of the directory or csv file.

After we parse through all the directories and CSV encountered it stores them into a global pointer to our movies, after we have this huge structure of all the movies that were encountered in the traversal, we run mergesort on them. Now once our mergesort presented us with an updated data structure containing the sorted movies we get the output directory flag and create the "allfiles-sorted-<fieldname>.csv" and print it into the file in the appropriate manner.

To run the program in terminal please use

```
Gcc -o sorter -pthread sorter_thread.c mergesort.c
```