CS214 Project 2: Basic Data Sorter - multithreaded

Abstract

This is the third in a series of projects that will involve sorting a large amount of data. In this third phase, you will modify your previous phase to be multithreaded rather than multiprocessed and examine the differences between these two methods of parallel operation.

Introduction

Keeping with the spirit of the project, differences between this phase and previous will be relatively minimal besides the methods of parallelisation.

Your code will read in the same movie metadata CSV files and sort the data in them with mergesort. Your code will also need to scan through the directory it is called on and sort each file in that directory and all files in all subdirectories. The major differences are that your program will instead spawn a new thread to search each directory and sort each file rather than a new process. Since all your computation will be in the same address space, you need not make separate files. The sorted output of all files will go to a single output file. Be sure that in the case of any bad input or a bad status code that your program fails gracefully, closes files, frees memory and exits threads. Under no condition should your code crash on a segmentation fault. As your code runs your threads should output similar metadata to STDOUT as the previous phase. Each thread should output the number of threads it created along with their threadIDs. Finally, you should study the differences between threads and processes for multicomputing. The metrics you should compute are detailed below.

Methodology

a. Command line flags

Your code will read in a set of flags via the command line. They are '-c', '-d' and '-o'.
Note: 'optional' and 'mandatory' below refer to your code's operation only. You must implement all three flags. Mandatory flags must be present for your code to run. Optional flags may or may not be present. If they are not, your code should take the specified default behavior. You must support all flags.

'-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:

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

$ ./multiThreadSorter   -d thisdir -o thatdir -c  movie_title

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

.. are all valid and would operate identically.

b. File Structure

CSV file structure remains unchanged. Records will be stored in CSV files in the provided directory. As mentioned above, directory structures may have multiple levels and you must find all CSV files. Your code should ignore non-CSV files and CSV files that do not have the correct format of the movie_metadata CSV (e.g. CSV files that have other random data in them).

c. File Sorting

Your code will be reading in and traversing entire starting directory and each contained subdirectory. Your code should open and sort each CSV file found that contains move data.

Your code's output will be single CSV file outputted to a file named:

AllFiles-sorted-<fieldname>.csv.

Your code should create a new thread to handle each subdirectory found, and each file found. Each file thread should read in and sort that file using Mergesort. You are welcome to implement another sorting algorithm, but it must be written entirely your self. Once sorted, Instead of writing out the sorted version, your file threads should write their data in to some common data structure in the heap. Once all CSV files have been sorted your code must take the individually-sorted files and integrate all the results to create a single sorted file containing all the records read in.

It would be advisable to insert the per-file data in to a globally-accessible sorted data structure. Be sure to keep this data structure propertly synchronized, however. You will need to make use of synchronization mechanisms to be sure that your updates are handled in a consistent and coherent manner.

d. Metadata

Your threads should write metadata to STDOUT while running. It should be in the following format and order:

Initial PID: XXXXX

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

Total number of threads: ZZZZZ

e. Analysis

Lastly, you will use the time utility to time the execution of your program:

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

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

You should run your Project 1 sorter on the same files/directory and record the differences.

You should build a variety of different directory structures and file sizes and test your current sorter and Project 1 sorter on both.

The goal is to see how multiprocessing compares in speed to multithreading. See how the two compare with:

- long subdirectory chains
- lots of files in a single directory
- combinations of the two

In particular, see how the time differs as the number of files and/or directories increase. For instance, collect timing information for both programs with a single directory and 1, 2, 4, 8, 16, 32, 64, 128 and 256 files. Plot the time taken vs number of files to see if there is a definite trend for both programs.

It would be a good idea to run each test multiple times and average the results. Other users, the speed of the hard drives, the OS scheduler, the difficulty in sorting the files, and any number of other factors may alter your results. It would be a very good idea to automate as much as you can. You may want to write some simple code to automatically create N files to be sorted, to run your code and collect the time results. It may sound like unnecessary extra work, but running everything manually by hand will likely take much more time than automating it. Let the machine do the boring work. Welcome to Computational Science!

Results

Submit your "multiThreadSorter_thread.c", "multiThreadSorter_thread.h" and "mergesort.c" as well as any other source files your header file references.

Document your design, assumptions, difficulties you had and testing procedure. Include any test CSV files you used in your documentation. Be sure to also include a short description of how to use your code. Look at the man pages for suggestions on format and content. Do not neglect your header file. Be sure to describe the contents of it and why you needed them.

Include a file called analysis.pdf that describes your findings from executing the time utility. Discuss why the results are the way they are. Be sure to answer the following questions:

Is the comparison between run times a fair one? Why or why not?
What are some reasons for the discrepancies of the times or for lack of discrepancies?
If there are differences, is it possible to make the slower one faster? How? If there were no differences, is it possible to make one faster than the other? How?
Is mergesort the right option for a multithreaded sorting program? Why or why not?

Extra Credit

a. (30 points)
Implement the fastest (wall clock time) program. Your program will be compared against all other submitted programs in the class. The programs that are among the top 10% fastest implementations will receive the 30 points. The time that will be used is the real time reported by time. To achieve this, you may need to re-examine your data structures and algorithms to optimize your code for speed.

b. (20 points)

Implement a fast (wall clock time) program. If your program is not within the top 10% but is within the top 30% of timed finishers. You will receive 10 points.

*In the event that all groups submit programs that run with times that are not different in statistically significant ways, we will give everyone 5 points of extra credit.