## Assignments

## Project 3: Basic Data Sorter - server/client - Returned

| | |
|---|---|
| **Title** | Project 3: Basic Data Sorter - server/client |
| **Student** | Lawrence Yong |
| **Submitted Date** | Dec 14, 2017 11:20 pm |
| **Grade** | **100.00 (max 100.00)** |
| **History** | Wed Dec 13 18:52:49 EST 2017 Lawrence Yong (ly172) submitted<br>Thu Dec 14 23:20:09 EST 2017 Lawrence Yong (ly172) submitted |

### Instructions

# CS214 Project 3: Basic Data Sorter – server/client

**Abstract**

This is the fourth and final part in a series of projects that will involve sorting a large amount of data. In this fourth phase, you will write a multithreaded C server and C client program to sort a list of records of movies from imdb alphabetically by the data in a given column. You will make use of the concepts learned in lecture including file/directory access, pthreads, and synchronization constructs (locks/semaphores), and sockets

It is recommended (but not required) that you join with another team of 2 – this way you and your partner can work on either the client and the other team can work on the server or vice versa.

**Introduction**

File formats for this part of the project are the same as in the first. The CSV file with movie metadata will remain the same. The sorting algorithm will also remain the same. If you properly modularized your code in Project 2, you should be able to reuse almost all of your code. The major difference is that you will have two blocks of code: a client and a server. The client will read in CSV files from multiple directories and instead of sorting them itself, it will connect to a server, send the files to it, and the server will sort them and send the files back.

Imagine a situation where there is one machine that is very powerful and flushed with resources. Then imagine a tiny, embedded system that must accomplish CPU-intensive tasks. Wouldn't it be nice if the tiny system could send all of the work to the powerful system and get the results back in a jiffy? In this project, you will:

0. Design a search request/response protocol for your client and server
1. Implement a multi-threaded server to respond to client requests using your protocol
2. Implement a client program that will send requests to the server using your protocol

The protocol for communication between client and server is an important agreement that must be adhered to by both teams. Consider this protocol carefully prior to implementation. Each client must be able to send to the server, at minimum:

0. The data to sort
1. The column to sort the data on
2. A request for the fully sorted list

Remember, the server is just reading bytes from a socket. It does not know intrinsically where data ends and commands begin. Your protocol must allow it to, at minimum, separate the data to sort from the column name to sort. Beware of using special characters to denote commands or separators. For instance, you could use 'X' to indicate the end of a block of a CSV file, but what if the character 'X' is part of the data *in* the CSV file? You need to be sure that, however your protocol is built, commands and\or separators can not be misinterpreted as data.

Be careful with control characters. Control characters like '\n', '\t', '\r' do not transfer well. They are special "characters" that are not printable, but are codes that cause the terminal\a file descriptor to do certain things. They can seriously gum up or break a socket connection. Do not send control characters with your data. Your protocol should have some way to either substitute a code for a control character or a way to work around them.

**Methodology**

a. Parameters

Your code will read in a set of parameters via the command line. Records will be stored in CSV files in the provided directory. Directories 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).

Remember, the first record (line) is the column headings and should not be sorted as data. Your code must take in a command-line parameter to determine which value type (column) to sort on. If that parameter is not present (?-> throw an error, or default behavior). The first parameter your program will be '-c' to indicate sorting by column, the second will '-h' to indicate the hostname of the server, and the third will be '-p' to indicate the port number the server is listening on. ALL of these parameters are required for the program to run:

```
./sorter_client -c food -h grep.cs.rutgers.edu -p 12345
```

Be sure to check the arguments are there and that they correspond to a listed value type (column heading) in the CSV.

```
./sorter_client -c food -h grep.cs.rutgers.edu -p 12345
    -d thisdir/thatdir -o anotherdir
```

The fourth parameter to your program will be '-d' indicating the directory the program should search for .csv files. This parameter is optional. The default behavior will search the current directory.

The fifth parameter to your program will be '-o' indicating the output where the sorted CSV should be written to. This parameter is optional. The default behavior will be to output in the same directory as the source directory.

The server has only one parameter: ‘-p’, and is required for operation:

```
./sorter_server -p 12345
```

b. Operation

*Client*
Your client should traverse all subdirectories under the directory it is given. For each file it finds, it should spawn a thread. Each thread it spawns should read in the file, construct a search request, connect to your server, send the request to your server and then wait for and read back the server’s response to be sure the file was sorted. Once the client has sent all files under the search directory to the server, it should send the server another request to give it the sorted version of all the files it was sent. When the server responds and sends back the sorted version of all the files, the client should output a SINGLE CSV file named:
AllFiles-sorted-<fieldname>.csv.

The client need not output anything to STDOUT.

*Server*
The server will open a port and wait for connection requests. On connection, it will spawn a service thread to handle that connection and go back to waiting for requests. Each service thread should read in a client request, if it is a sort request, it should perform the sort and store the results at the server. If it is a dump request, it should merge the current collection of sorted results into one sorted list and send the result back to the client. You may want/need to make use of synchronization constructs like mutex_locks, semaphores, and/or condition variables in your implementation to prevent memory corruption.

The server will run until stopped by a SIGKILL (i.e. kill <pid of server>).

To STDOUT, output a list of the ip addresses of all the clients that have connected: (?-> when?)

```
Received connections from: <ipaddress>,<ipaddress>,<ipaddress>,…
```

c. Structure
Your code can use Mergesort to do the actual sorting of records. It is a powerful algorithm with an excellent average case. You are welcome to implement another sorting algorithm if you wish. You may not use a third party library for sorting purposes (e.g. you must write the code yourself).

**Results:**
Submit your "sorter_client.c", "sorter_client.h", “sorter_server.c”, “sorter_server.h” and "mergesort.c" as well as any other source files your header file references.

Document your protocol 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.

**Extra Credit**
Here are some extra credit options for you:
Socket Pool (10 points):
On the client side, implement a socket pool whose size is given as an input parameter to your program. E.g.

```
./sorter_client -c food -h grep.cs.rutgers.edu -p 12345 -s 4
```

Would create a pool of 4 sockets. All communications are limited to those 4 sockets. Every thread should request access to one of the sockets. If no sockets are currently available, that thread should sleep. (hint: semaphores are built for this kind of work). Document your design in your readme.

Server Sessions (10 points):
When a client connects a server for the first time, it should first request a session ID. Then, whenever the same client sends a sort request to the server, it should include its ID in the request. The server will keep the sorted data for each ID separate. When a client requests all the sorted data for a given ID the server can then deallocate that data, but must keep it until then. Be careful. This requires non-trivial extensions of your protocol. You should also handle error cases and malicious input, like a client asking for merged data for an ID that doesn’t exist, or a client trying to do a sort for an ID whose data has already been sent back and deallocated.

---

**Submitted Attachments**

- Sorter3.tar ( 480 KB; Dec 14, 2017 11:20 pm )

Back to list

---