Adithya Thiruvalluvan, Mitch Lew, Dhruvik Patel, Sagar Phanda

Systems Programming Writeup

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.

Client Design: To begin we start with the design of our client. We handle the arguments for the client, making sure that the user has input the directory, outputdirectory, hostname, and port number. This is done in our function arghandle.

1) ArgHandle

Arghandle: Our first objective handle the arguments inputted in the command line. Through main I declared 5 strings: column, directory, and outputdirectory, hostname, port number. Column representing the header column that we need to sort in each csv file, directory representing the directory we want to search through and sort the csv files, and outputdirectory representing the directory we wished to output our csv files. Portnumber is the port number that we want to connect to form the host, hostname being the host that we want to connect to. I also defaulted the value for directory to current directory or '.' Next we passed those the values of the five strings into arghandle along with the argc and argv. In arghandle we checked to see how many arguments were inputted into the command line. If they were less than 7 we terminated the program. The executable file, the column identifier –c and the actual column field we need to sort must be inputted for our program to work as well as the port number and host name. Following that we checked to see if any value for column, port number, and hostname was inputted. If they failed to input an argument for column the program would terminate. Also if they failed to have a proper column heading, port number or hostname to sort the file the program would terminate. We also checked to see if any values for "output directory" and, "directory we are searching through" were inputted and set outputdirectory and directory equal to those values respectively if inputted. It is also mentioned in the instruction specifications that the order for which the –c, -d, -o, -p, –h flags are inputted should not matter. As a result in our code we traverse the input argument array argv and account for when we see the –p.-h,–c, -d,-o flags. When we see the flags we take the array index immediately after that current index and set that equal to its respective flag. This allows the flags to be set in any order and for our program to output the same. Once these conditions are checked we have handled the arguments inputted through command line.

Once we have completed the argument handler function we create a strut "handarg" which we put all the arguments into so that we can pass them to our thread later when we create the threads. Then we call the function handy with the argument being the structure that we have inputted the arguments.

2) Handy

Through handy we would use directory pointers and struct dirent's to traverse the file directory and access their entries. First we set our directory pointer to the value of "fopen (directory)". If the directory doesn't exist we fail the program. We then determine in if the files within that directory are directories or a files. If the entry is a .csv file we spawn a thread to pass the .csv file to the server. If the entry is a directory we rename that directory (original directory inputted in main)/ (entry->d_name) and spawn a

new thread that calls handy. This allows for parallelism within multiple threads. As the main thread works on all the directories and files within the specified directory from the input arguments, the new threads work on the handling/sorting the .csv files or traversing the nested directories within that directory. However, we do not want the main thread to end before the spawned threads finish execution as that would terminate all the spawned threads before they are finish. As a result we must wait on all the threads associated with a specific directory to finish before we exit our handy function. To do this we must wait using pthread_join. Every time we call the handy function we create a linked lists storing all the spawned threads associated with that thread. Then after the directory traversal in a single thread we create a while loop that traverses the link lists and waits on the thread IDs in each iteration of the loop. For instance, I specify to search through dirA in my input arguments in the command line. Within dirA I have dirB, 1.csv, and 2.csv. In my main thread I will create new threads for dirB, 1.csv, and 2.csv. Upon their creation I store the spawned thread's TID into a local linked list. Then at the end of my directory traversal I create a while loop that pthread_join from the beginning of my link lists. So when the thread associated with 1.csv finishes the main thread allows the while loop to continue. When the thread associated with 2.csv finishes, the main thread continues. DirB will spawn its own linked list and wait for the thread associated within it to finish similar to the main thread. After all the threads are finished executing we can continue pass the handy function.

3) SearchRequest

Search Request begins the handshake between the server and the client. The original arguments that were inputted to the argument structure that I created was passed through handy and into searchRequest. Here in search request we use those arguments to connect to the right server via the port number argument inputted from main. To begin we create a socket for the client. We then connect to the server after the socket creation. Following this connection we open the .csv files and check to see if it's in proper format to sort, to make sure it's not sent to the server unnecessarily. If it is able to sort we write the first line to a buffer and send the buffer over to the server, waiting for a return message via the read statement, so that we keep synchronization with the server. After sending the entire file we close the socket.

4) DumpRequest

The is the last part of the project where all the sorted records are send back to the client and an output can be created. To do this, we need to create a socket again. Once that socket is established, we need a connection. Following the connection, confirmation from the server will be needed to ensure that it is able to send that data. Server will send a message to the client to start the data transfer and again we can do it by using buffer. To do this we write the first line to a buffer and send the buffer over to the client. We then wait to send the next line over until a signal is received from the client that it is ready to receive. We have to keep everything in synchronization to make sure everything is placed properly into the client. After sending the entire array we close the socket. Also in this request, it creates the final file from the buffers it receives from the server.

Design: Server)

To begin in main the server takes in a port number for it to bind to in its arguments. If a port number is not specified after the –p flag or if the port number is not inputted into the execution of the server than the server fails gracefully. To begin we create a socket for the server. We then bind the socket to the server address. Next we create an infinite while loop so that we can listen for as many connection as possible from the client through threading. In the while we listen for a client. Then we malloc space for the new socket on the heap. Accept a connection and store that new socket into the heap. This allows us to create a new socket in the server every time it accepts a new connection, preventing data to be overwritten when multiple .Csv files are being written to server at the same time. We create new thread after accepting and tell the thread to execute the conhandle function.

1) Conhandle

Connection handle handles the connection of each connection that connects to the server. In connection handle we begin the handshake with the client. Initially the client sends over the amount of lines in the .csv and column heading we want to store in the .csv. We then use a tokenizer to store these values in variables. We then check to see if we got a signal from client telling the server to perform a dump. (Dump explanation later) Next receive the .csv file line by line in a while loop. Within the while loop we add each line to an array via the addelement function.(to be discussed after) then we write back a message so that we the client and server fully synchronized. After we finish storing all the lines into the array we call the mergesort function which sorts the array of structs via the column heading. Then we call the addGlobal function (to be discussed). This function store the .csv file into a linked list in shared memory.

2) Addelement

The add element store the line of the .csv sent from the client into a structure Record, that we defined in our .h file. It takes the strings and comma by comma, we tokenized the string and stored them in our record structure for each column heading.

3) addGlobal

In our addGlobal function we added each sorted array of lines that constructed the .csv files into a global data structure (linked list). Utilizing the methods of linked list.

After the completion of conhandle we close the connection between the client and server.

Assumptions: All csv files with 28 columns have the proper column heading. If there were more or less than 28 column headings than the csv file would be improper.

Difficulties: A difficult in this project deals with the communication between the client and server. We needed to sync everything up in order for us to successfully get the program running. Also threading was a bit difficult too since we are threading to a server.

How to use code:

First determine which command window will be the client and which one will be the server. To compile the client you would use the flags –o –g and –pthread as such:

gcc sorter_client.c –o sorter_client –g –pthread

To compile the server you would do the same thing as such:

gcc sorter_server.c –o sorter_server –g –pthread

Here you would first call your server first to get that running. Since one parameter is needed for this program to run you would do the following:

./sorter_server –p <portnumber>

Now you can call your client to connect to that portnumber and also the server's hostname as such:

./sorter_client –c <column heading> -p <portnumber> -h <hostname>