Chris Mathew
Jacek Zarski

# Client-Server: Multi-Threaded-Sorter

1. Design
    a. Client
        i. Header
            1. Defined Various sizes for different buffers, calloc's, strings, etc.
            2. Small is 1024 bytes
            3. Medium is 2048 bytes
            4. Large is 4096 bytes
            5. Max is 8192 bytes
            6. Defined a general-purpose port as a string -> “3490”
            7. Two structs record for csv rows and middleware to hold the file path information
        ii. Source Code Algorithm
            1. Start with argument checking in the main
            2. Use it to get a hostname, port number, field to sort for, parent directory, and output directory
            3. Throws errors if incorrect usage
            4. Move on to thread mutex initiations with errors if they failed
            5. Then calloc thread array and a file path holder array
            6. Then I get the full path for the parent directory and if it’s not null we move on to the recursive directory traversal
            7. In traversal, we check for two things if it’s a directory or if it’s a file
            8. If it’s a directory we spawn a new directory thread, which store the file path into the global file path struct array and continue traversal
            9. If it’s a file we then check if it’s a csv else we ignore
            10. If it’s a csv, we spawn a new thread, store the file path into the global file path struct, and begin parsing and sending the csv file
            11. In order to parse and send the csv we first must create a socket and connect it
            12. Once connected we then begin the parsing where we get the headers, check if it’s a valid csv (has 28 columns), and if so we get the target column in integer form and send that over to the server
            13. From there we go forward and signal to the server that we are going to begin reading line by line the csv storing it into a buffer and then sending it to the server and the signal the end of the csv
            14. Once all threads have done that we begin joining the threads, and then we begin creating the output file at which point we start the receiving master file process
            15. We open up a new socket, connect it and then signal that we need the master file
            16. Once we start receiving the file we immediately write it to the output file

17. Once that's over, we close of the socket, file stream, destroy the mutex's, free calloc'd information and then exit out of the program

b. Server
   i. Header
      1. Same defines as client
      2. New struct for client info which contains socket file descriptor and its respective sock address struct with their client's information
   ii. Source Code
      1. Check arguments from command line to see that everything was done correctly if not throw error.
      2. Create a socket, set the socket options, and then bind the socket.
      3. Next, we create the mutex locks, calloc the global structs for threads, client information, and record array.
      4. We set up a SIGKILL and SIGINT callback handler to exit out of the server program
      5. Then we begin the receiving of information by opening up the socket to accept connection
      6. Once we get a connection, we spawn a new thread to handle the client.
      7. The new thread will then begin by receiving information the first of which should be the target column to be sorted in integer form
      8. After that we should receive a stream of data for a series of csv lines.
      9. As we get the csv lines we will tokenize the line and store them into a global struct in a form that get rid of special character and delimiters/
      10. Once the file stream has ended we will receive a signal from the client stating end of CSV's
      11. We then perform merge sort on the target column, and then once we receive the signal from the client we begin sending the client the sorted column
      12. Once it's over we keep the server running until we receive a SIGINT or SIGKILL

2. Assumptions
   a. We could send an entire record of a csv and receive it back in the same form rather than in small packets of information
   b. We assumed our tokenizer function would work here as it had in the past
   c. We assumed connecting to the iLab machines would be the same as connecting to local host but that was not the case
3. Difficulties
   a. Reading bytes from the client and storing them into a global struct
   b. Getting the socket to connect to the server's socket
   c. Memory allocations and memory leaks -> solved by using calloc and other functions like bzero, memset to zero out memory

    d. Figuring out the algorithm to our signaling so the server knew when the client was sending the target column, the csv files, when the client was done sending csv files, and when the server was done sending the sorted column.
4. Testing Procedures
    a. We created our own test directories containing various csv files in sub directories, and other test directories that just contained a lot of csv files to see if the code held under pressure
    b. We worked from the ground up first getting the sockets to work, then getting a client to connect to the iLab, then configuring how to receive and send information, etc.
    c. We had a lot of trouble with getting the server to read the lines from the client and from there to tokenize and store the data into a global struct which we would then later perform merge sort and send back to the client
5. Instructions
    a. First have all the code files in one directory.
    b. Make sure to have the directory where you want to sort be in the same directory as the code.
    c. We used a makefile to streamline compiling and testing so we recommend you use it as well.
    d. To compile the client in your command prompt or terminal once you have changed directories to the directory containing the code run make cli to create the client.
    e. Edit the client command args in the makefile to your liking.
    f. Once make client has compiled, you then want to run make serv to create the server again changing the args in the makefile to your liking.
    g. Make sure the ports are the same in both the server and the client
    h. After that the server should be running in which case you want to run make ilab to run the client using the ilab args again changing the args to your liking.
    i. Now let the client and server do its things and the client should exit out on its own.
    j. Server will need a control – c to exit out otherwise it runs forever.