Project: Database Management System

Introduction

The purpose of this report is to document my progress throughout this project, to describe my understanding of the tasks at hand, to explain my software design and to highlight any challenges I encountered along the way and how I dealt with such issues. The task at hand to design and implement a database management system was incredibly complex, even daunting at times, but nonetheless was essential to improve my understanding programming concepts in Bourne Again Shell and using Linux systems more effectively.

Requirements

The purpose of this project was to implement a database management system which uses comma-separated value file formats which would use SQL style commands to create, edit and output data from such files. It would require various individual components (scripts) that would eventually operate in conjunction with each other to render this system usable by the end-user (i.e. the client). Communication between these components should be considered vital. Thus, the inter-process communications aspect of this project is what would make or break its functionality.

The client should never be required to access these components directly and instead, simply send their requests in the expected format to the server, which would internally handle these requests and carry out the desired functions.

Additionally, unprotected database systems are prone to malfunction and errors which arise from unsupervised read and write access to files (concurrent access to the same file by two separate clients, for example). In lieu of that, another key aspect to the functionality of this system therefore requires a method of protection for these files while they are being accessed by the server.

Database systems should also only carry out exactly what is needed, not more and not less. This means that should a client request specific information (e.g. columns 1 and 3 of our CSV files), we should not receive all data contained in the CSV file. Therefore, it is important to build a mechanism which only finds the data requested. In a larger system, this would be important as too much excess data being continuously accessed and output is computationally expensive.

ARCHITECTURE

Building from the ground up, we must first begin with the scripts that will handle our manipulation of the data. These are the core aspects to our system. We must construct these individual components first and ensure they function as required, before building the broader system of implementation. The four scripts which are essential to our database management system's functionality are:

- The ability to create a database (in our case, a folder)
- The ability to create a table which stores information within the database (in our case, a CSV file)

- The ability to amend the data within these files (insert new data as needed)
- The ability to query the database for information we require.

After our core scripts are built, we move onto the server script. The server script is designed to constantly listen for pre-defined commands. It does this by creating a named pipe and reading the inputs from that named pipe so long as the server is live. The server listens for these requests (create_database, create_table, insert and select) and expects varying parameters with each command. If the server deems the command acceptable, it forwards the command to the scripts. The scripts then carry out the functionality requested (if it matches the required format). The server forwards the standard output from the script execution back to the client, again, through a dynamic named pipe that is created when the client logs on (i.e. the client script is executed).

The client script is responsible for keeping the client session online. Without named pipes, we would not be able to contain the data and only display to each user who requested it and thus would not be much use. As mentioned, when the client logs on, a named pipe with their ID is created. This pipe is used for receiving information from the server. The client sends requests to the server's pipe, along with its ID, so the server knows where to send the information back to. It then reads its own pipe and output the information back to the user.

IMPLEMENTATIONS

This section will discuss the scripts which were difficult to implement and required some research and understanding before achieving functionality.

INSERT

Insert was the first script that caused difficulty, as I was unsure in how to verify that the information, we were attempting to append to the database in fact matched the database. I needed to identify a way in which we could ensure the uniformity of columns with column headers.

We are able to do this with the “tr (translate or delete” and “wc ” commands. We do this by assigning the table header count to a variable (replacing the “,” with “ “ and using wc -w) and doing the same with our argument $3 (also separated by “,”). If these match, we know that it is safe (at least in terms of column uniformity) to insert the data. We carry on with the function to insert the data in its traditional form (with the comma separated format) to the CSV file.

As a small footnote, in this section I also utilized printf instead of echo. I was experiencing some issues with the formatting in my CSV file and printf gave me more control over the formatting[1].

SELECT

This query was one of the most difficult and required the most amount of time researching. I was unsure how to implement this at the beginning. However, using knowledge from other languages, I figured the easiest way to do this section was to use arrays. It seemed like the most straight forward way to access pieces of information individually from a single parameter ($3), where each value related to a separate column.

[1] https://unix.stackexchange.com/questions/58310/difference-between-printf-and-echo-in-bash

This script requires much validity checking. First, we want to make sure that the request is inbounds (i.e., ensure if the user requests 4 columns that 4 columns exist). Interestingly, this was difficult to solve when working with arrays and the cut command. The cut command begins at index 1, while almost everything else begins at index 0. In order to ensure mathematical equality, we needed to -1 from the final index value of the user request, in order to effectively compare it with the table header values which begin at 0 (and thus would be -1 at each position) [2].

By altering the final index value of the user request, now we can verify if that column indeed exists in the table header. For this, I used a Boolean counter which is set to false by default and only changes to true if it finds a match between the table header index and the fixed final column value. Another thing to note here is that I could not -1 from the final column value unless I preceded it with the keyword "expr", since without it, it's considered a string.

Some additional error checking was done in this script, to ensure that user requests start at 1, and not 0. Finally a sed operation iterates through the table header indices and replaces the table header value with i+1, giving us the requested columns.

LockDB

A minor issue with getting this semaphore to work was that linking on directories wouldn't work. I identified that adding -sT allows us to treat folders as files when trying to us link (ln) command, effectively allowing us to also lock directories.

[2] https://unix.stackexchange.com/questions/177823/linux-cut-command-with-f1

SERVER

Again, reading the input from the server pipe into arrays was necessary in the server since single arguments can contain multiple values. It also allowed for error checking (such as whether the number of arguments in the array passed to the server match the number that the clients take).

As long as one does not get lost with the correct index references to the array, this section was fairly straight forward. However, some mistakes with the indices when passing the values from the client to the server caused some unexpected results until these indices were fixed.

Further, it is important to send all messages produced by the scripts from the server back the ClientID that requested them. Therefore, in each case statement, after validity checking, we pipe the standard output back to the “ClientID.pipe”.

CLIENT

The biggest issue I faced when developing the client section was my misunderstanding that “input” was a keyword for the read statement. I understood that “read input < pipe” was necessary in order to receive results from the pipe. However, since I used “read input” earlier in order for the client to listen to the stdinput, the variable associated with $input was not returning any information I desired from the server. Using “read output > server.pipe” effectively resolved this issue.

A key element with the client and using read on a pipe was that it only reads one line at a time. Our queries sometimes require that we display a wall of text. In order to resolve simply receiving the first line of a query (i.e. start_result), after printing the first $output, I added an ‘if’ statement which checks if $output is equal to “start_result”. If it is, then we enter a while loop which continues to read $output until $ouput is equal to “end_result”. There are plenty of other ways we could have done this (e.g. with another “case in” statement), but this approach was simplistic, and it appears my error handling for the other sections have resulted in not unexpected errors being returned by $output.

CONCLUSION

This was an extremely complex project which required many hours of research, learning and experimentation. Though, for the advanced programmer, this might be simplistic, grasping the concepts as a beginner with little experience was a challenge.

The project required the interaction between various components, including the scripts to build database directories, files, amending data, having a server that is constantly listening for pre-defined commands, a client system which continues to list to user requests, an inter-process communication system (named pipes) which carry the requests from the user to the scripts and back, with the server acting as the intermediary.

I have learned to use the tools provided in the learning environment in a practical setting and feel supremely more confident in my ability in using linux and the bourne again shell.

Some of the biggest issues faced in this project were the dissecting of arguments, using arrays to pass information around and reading inputs and outputs from pipes. Now, with it all complete, how the system works together makes a lot of sense and I believe that I can see how one would implement these tools in a commercial setting.