Operating Systems Assignment – BASH DBMS.

Report

Kieran Curtin
18205921

COMP30640 Operating Systems

UCD School of Computer Science

30/11/2018

Introduction

Using BASH on the CS server and an Ubuntu app on my Windows machine, I have made a rudimentary Database Management System. Here clients can communicate with a server to create databases, create tables within databases, insert rows into tables, and select data from tables. I have also implemented a bonus feature, adding a "where" clause to the select function.

Requirements / Functionality

The system I have made is intended to be used by many clients. Initially a server script is executed; this is always listening for requests that can be sent in by any number of active users/clients. Responses are sent to a pipe unique to each client.

Clients enter their requests as text into the standard input, this is then sent to the server. The server receives the request and processes the commands (it will run the scripts with the arguments that have been passed to it via a named pipe from the client standard input). If the command has been successfully executed, or if there has been an error, an appropriate message will be sent to the client.

As requested in the assignment, to terminate the server the client enters "shutdown", and to terminate the client, "exit" is entered.

Architecture/Design

Create_database.sh

This is the first script I created. It takes exactly 1 argument – the name of the database. I have conditional statements to check the number of parameters is correct: 1) make sure that only 1 parameter has been passed to the script 2) make sure that this parameter is not an empty string. Then we enter the "critical section". Even though the database may not exist, we can still put a lock on it as we already have the name – the first parameter of the script. In the CS, I create a database (make a directory), if it does not already exist. A message will be sent to the client regarding the outcome of the command, and then we exit the CS and exit the script with exit code 0.

Create_table.sh

This script will create a Table (with column names/headings) within a database that already exists. 3 arguments are supplied to the script, database name, table name, and table headings. The third parameter, table heading, will be in the form "1,2,3,4" or "Name,Age,Address,Telephone" i.e. they will be comma separated values. As with the previous script, I have conditional statements to check that the number of parameters is exactly 3, and to check if any of them are empty strings. We enter the critical section as we check the validity of the

arguments that have been passed: 1) if the database exists 2) if the table exists. If the database exists, and the table does not, a table will be created. I use the "./" to access the current directory, and "touch" a file within the directory i.e. I navigate to $1 (which is database) and create $2 (table name). The third parameter of the script is the column headings. Once the file has been created, these are echoed/appended into the file. A message will be sent to the client pipe confirming the result of the request.

Insert.sh

This script will insert data into a table that already exists within a database. It takes exactly 3 parameters. As previous, I check to make sure that exactly 3 arguments have been given, and that none of them are empty strings. We then enter the CS and I check if the database and the table exist. I then create 2 variables:

$x = The number of columns that the user wants to select. I am taking the third parameter (the CSVs), echoing and piping it into a SED command where I am substituting all the commas for newlines (globally), then counting the amount of lines. Essentially, I am checking the number of columns that the user wants to insert.
$y = The number of columns that are in the "table", the file they are trying to add to. I use the same technique as above, but I am taking the first line of the table and piping it to the other 2 commands instead.

I check to make sure $x is the same as $y (clients are only able to insert into table if the insert data has the same number of columns as the table). If everything is OK, then I echo and append (>>) the third parameter into the database and send a response to the client.

Select.sh

This was the most challenging script to make and the one I spent most time on. I do 2 checks at the start to make sure that at least 2 parameters have been supplied to the script (the reason for maximum of 5 is explained below in the WHERE clause). After this, I check that the 2 parameters are not empty strings. Then we enter the CS so I add a lock to the database. If the database and table exist, then results will be printed. If 2 parameters are supplied, the entire CSV file will be displayed. If a third parameter is supplied to the script (the columns) then I must assign more variables:

$x = The number of columns that the client is trying to display.
$y = The number of columns in the Table in question.
$z = The number of lines that are left in the table after the headings/column names.
$fdem = Variable that will be passed in for the field(s) to be displayed in the cut command on the rows of the database.
$range1 = This variable exists due to lines 55 and 66 in the select.sh script. (description below).

The for loop on line 33 deals with a varying number of column requests by the client ($3 can be any number of columns). Here I am taking each column and saving the value to a variable, but I also want to have access to the variable outside the for loop. I have used export to allow this. This might be overkill because I do not need access to the variable from *other* processes outside this script. However, declaring variables within the for loop and exporting them allowed me to compare/check their values later…

For each variable (column) declared in the above for loop, I then check to make sure that this value is not less than 1, or greater than the number of columns in the Table being queried. (I am still in the CS because it is possible to reach this stage without executing a V.sh and exiting the script.) If the client is selecting more columns than exist in the Table, an error message is sent advising to optimize their query (for example, if columns 1,2,3,1,2 are selected from a table that has only 3 columns, they will need to remove duplicate requests).

Next step is to print results. I am entering a for loop and will be iterating through each line of the file. However, because I want to start from the second line and exclude the "headings" line from the Table, my range starts at 2. Therefore, I have created the $range1 variable, which is [(number of lines left in the file after the heading) +1]. This ensures that the last line of the Table gets "selected" and echoed out. I declare a new variable in the for loop to accept the result of the "select" for that line. I am using Stream Editor to access each individual line of the Table file, cutting on the comma, and selecting using the $fdem variable declared earlier.

Select.sh (where)

This script adds extra functionality to the Database that was originally described for the project. This is the reason that I am allowing up to 5 parameters to be supplied to the script. Select queries are made through standard input as suggested in the project PDF and follow the form "./select $database $table $columns $specific_column $condition". Here is a sample of the BASH query translated into SQL (keeping the variable names supplied to the script), assuming the query is being made on "$database":

Select "$columns"
From "$table"
Where "$specific_column" = "$condition".

To do this I am declaring a new variable:
$a = This is the specific column that the "where" clause will look at.

Then I compare the $condition against that column for each line in the file and check for equality; For each line in the file, if column $a is equal to $5 (5th parameter), then print the line. This is an example of using the select with where clause:

Select TestDB TestTable 1,3 1 Cork

Semaphores / Locks

I encountered a few problems when implementing locks onto my database. First, I was locking the database at the start of what I believed to be the CS, and was unlocking the database at the end of the CS. However, I was only considering situations where a database / table was created successfully and did not think about situations where the script would terminate with an exit value of 1 and not complete the request. I went back and had to change my scripts to unlock the database every time the script terminates e.g. make sure that there is a V.sh before every exit 1 / exit 0.

Also, I struggled with creating locks on directories as hard links are not allowed. To get around this, I created the hard link using the script as the link, but the name of the database for the lock. Originally, I had also locked the script instead of locking the database, this would have led to delays for clients as in that scenario, only one client would be able to have access to the database/table/insert/select scripts at a time.

Reading Pipes and Standard Input

I spent a lot of time trying different ways to successfully read data from standard input and from pipes. While I'm not sure if this has been done in the most efficient or correct way, the client and server are still able to communicate with each other effectively. When reading commands from the client to the server through the server.pipe, it was easiest to save this input as an array variable. I could then index different elements of the array to build the commands for the server to pass as parameters to the main scripts. It was easier to read responses sent from the server to the $client.pipe for the first 3 scripts, as they will only ever send 1 of 2 answers; "OK: …" or "Error: …". I read a word/answer from the pipe and then echoed this out. However, for the "select" script, reading out multiple results from the pipe was more difficult. I tried checking that the first line was "start_results" and then printing every line until you reach "end_results", then printing the final line. I was not able to achieve this, so instead I set up a while loop, and was reading from the pipe and echoing the result, while there was information still in the pipe.

Reading from named pipes, I was getting errors when trying to use the name $variable.pipe (e.g. $1.pipe). Instead I renamed the newly created pipe to a variable "pipe" and then read from the pipe using " < $pipe".

Case statements / Server & Client

I had not used this before starting the project. It is a very clear and simple way to incorporate pattern matching in BASH. I found the example given to us in the PDF easy to use, so I modelled both my server and client scripts off this template, and it seems to work well. I have split my error checking up into 2 areas. The individual scripts will check the parameters and make checks to see if databases/tables are valid. The client checks that the requests are in the correct format before sending them to the server, or else will tell the client that it is a bad request.

Reading responses from the server can probably be done more elegantly. I put in a sleep 1 to give the server enough time to process the request, run the script, and then send a response. I should have a clause in there that states "while there is no word in the pipe, sleep 1, check again and if word in pipe, echo

word". However, I found this difficult to implement. Once I was getting correct communication between the server and the client, I moved on to other issues as I had to focus my time on improving functionality in other areas.

Exit codes and redirecting output

Having exit codes made it easier when building and checking scripts, as I could see if it had exited correctly without needing to constantly add echo statements to see what was happening. For communication between pipes, I am sending and receiving standard output. For some cases I was getting error messages printed to the server. I tried to suppress these error messages "2> /dev/null" but I was not successful as some errors are still printed to the server (most, if not all, of these errors were actually resolved later as I relocated where my variables were being declared in the select and insert scripts). This is a solution to make the program "cleaner", but I do realise that in a real-world situation, it would be difficult to debug/make changes/improve the system as you would not be getting any error messages!

Challenges

Select.sh script|: My main problem was accessing the value of a variable after I had exported it within a for loop. I was trying to have a "double" variable within the *next* for loop, as I wanted to access $column($i) : the variable $column, but with the number variable of the for loop at the end. This was made possible by creating a new variable before trying at access the value (line 40). I did this with help from the TAs and using the syntax ${!variable}. It was complicated, but eventually I was able to check that each column request made by the client was greater than 0 and not greater than the number of columns in the Table.

Placement of P.sh and V.sh : I had initially placed my scripts on either side of the section that *created* the table or database, so my critical section was quite small. However, if it remained this way, then 2 clients could still try and create the same database/table through context switching… They would both be able to check and confirm that a database does not next, and then both move on to trying to create it. In my solution, the "writing" part of the file was locked for one client at a time, but not the logical checks beforehand. I had to alter my scripts and move the P.sh script up a lot further. For example, my CS is now starting before checking that a database/table exists, instead of starting just before creating the new database/table.

Appending to and overwriting files: A simple error. When inserting new tuples into the table I was using > instead of >>. This was easily solved.

Checking validity of parameters passed to the script: There may be too many checks here, as I am always checking the number of parameters, then checking if they are empty or not. It caused problems in the select script when I was allowing *either* 2 *or* 3 parameters to be passed to the script. I added the checks to that script, and then went back and updated all the other scripts as well.

Declaring variables: I already mentioned the problem I had in creating a variable within a for loop and accessing it later. Apart from this, I was trying to make my scripts easier to read/navigate by declaring all my variables at the start of the script. However, I came into problems with this, as some of my variables involved using information from the third parameter… I was trying to use $3 before checking if it existed! I had to rearrange my scripts to declare variables *after* checking the script parameters first.

Modifying database : I got my database working quite early, but continued to try and optimize and make changes to improve its efficiency/concurrency etc. I regularly made small changes that actually impacted much more than I thought and had a huge knock on effect. This made me really analyze each line of code and ultimately gave me more understanding of what each line did and how the client and server communicated with each other.

Using the CS server / Ubuntu Application on windows: I started the project by using ssh to access the cs server. Not too long after this I realized it would be much easier to continue if I had the files saved locally. It is easier to run multiple terminals and check scripts. Also due to the high activity on the server it was not always reliable regarding performance. It took me longer than I thought but I used scp -r from the windows command line and copied the files. Finding the ubuntu file system was another awkward job, but I was able to move these files into the canonical local state root filesystem. Then I had to revisit the early practical's and change the permissions of the directory and its files, so I had access. It was worth it in the end.

Multiple users: I tried running the sample test script on several different terminals at the same time. I did not encounter any errors, but I am still not 100% sure of the concurrency of the system. I re-checked the location of all the P.sh and V.sh scripts and ran multiple clients.

Conclusion

Implementing this DBMS through BASH was challenging but it highlighted important steps when undertaking a large project. It is important when communicating between a server and multiple clients to "prototype" your system on paper, it can help to visualize what is happening when scripts are being executed. Decomposition of the problem into smaller more manageable tasks also makes it easier to approach. The project taught me a lot about creating and accessing variables and to consider their local/global scope. Using semaphores in a lot of different scripts helped me with my understanding of the critical section.

The project also demonstrated how versatile BASH is, as I know many of my colleagues are achieving the same result using completely different techniques.

During my own testing, the database is working correctly with create_database, create_table, insert and select scripts all executing as expected when invoked from client.sh. I have areas of code that can be made more efficient and more error handling could also be added. I have tried to suppress all undesired error messages and make the system as user friendly as possible.