Operating Systems Project: Database Management System

Patrick Kennedy
18205918

COMP30640: Operating Systems

UCD School of Computer Science
Date Submitted: December 2nd 2018

Introduction

In this report I will detail how I created a simple Database Management System using Bash. This system mimics other DBMS systems used today such as MySQL and Postgres SQL. Like other DBMS systems it allows users to create databases, tables and perform insert and selects on those tables. In addition, for bonus points, I implemented delete database and table functionality. This DBMS, just like others currently used today, allows for concurrent users.

Requirements

For our individual Operating Systems project we were tasked with creating a simple Database Management System that comprised a server working with one or more clients. These clients were to be able to access the server concurrently. The system itself was to allow for the following use cases:

1) Creating databases (a database is a directory in our rudimentary system).
2) Creating tables (a table is a csv file).
3) Inserting data into a table.
4) Selecting data from a table.

The system follows the client-server design pattern. The server was required to do the following:

1) Read from a named “server pipe” for commands from a client process.
2) When a command is received, if it is a valid command (create_database, insert etc.), execute it.
3) Send the output from that command to the requesting client’s named pipe.

Each client was to perform the following:

1) Send commands in the form ‘req args’ to the server.
2) Wait for the server to execute the commands and display the responses back to the user.

A further requirement was to ensure that multiple clients could access the same server process concurrently.

Design

As briefly mentioned above, the overall design follows the client-server design pattern. Within this pattern, the server script (server.sh) calls multiple other shells scripts that perform the required database operation. These scripts are:

1. create_database.sh
2. create_table.sh
3. insert.sh
4. select.sh

To gain bonus points, I also implemented two more scripts that I think add to the overall functionality of the system. These are:

1. delete_database
2. delete_table

I will cover how these work in the 'Bonus' section, later on in this report.

Database Management System Functional Scripts

create_database.sh $database

The create_database.sh script takes the name of the database that you want to create as an argument. In practice, this creates a directory with that name, if it doesn't already exist. The new directory is created in the same directory where the create_database.sh script is stored.

create_table.sh $database $table $tuple

The create_table.sh script creates a table (file) in the database (directory) with the column headings specified in the tuple argument supplied to the script. The tuple is supplied as a comma separated list of values.

insert.sh $database $table $tuple

This script inserts a comma separated tuple into the table (file) in the specified database (directory). It contains error handling to ensure that the number of values supplied in the tuple matches the number of columns in the table.

select.sh $database $table ($tuple)

This script selects values from the specified table (file) stored in the specified database (directory). If no tuple argument is provided, all rows will be returned. If a comma separated list of integers is supplied for $tuple (e.g. 1,2,3), it will return those columns (after checking that they exist).

server.sh

The server script was developed in an iterative manner. For the first iteration, the server script waited for input from standard input on the terminal after being launched. After a user-typed input into the terminal, the input was split into an array. A switch statement then checked the first value in this array against a list valid commands. If the command was valid, it then passed the rest of the values from the input array as parameters to the associated script. If a command is invalid, an error message is returned to the user. Once I had tested these all at the command-line successfully, I began work on using named pipes to send these same commands from a client process.

The second (and final) iteration of the server script reads from a named pipe, server.pipe, instead of waiting from input from the terminal from the user. The server pipe is created when the script is launched from the terminal. It is removed when the server script is stopped from either the client or by typing ‘CTRL+C’ from

the terminal running the server process. The client's send along a unique client ID along with the command and its arguments, so server script needed to adapted to accommodate this. What changed was that the input_array indexes needed to change and the output of the functional scripts were redirected into the correct client pipe, based on concatenating the client ID with '.pipe'. To ensure scripts can be run concurrently, each statement within the case block to run as a background process.

client.sh

The client script is how users interact with my database system. Functionally, it takes commands from standard input on the terminal and, if they are of the form 'request arguments', they send that command to the named server pipe. Before sending the request, the script inserts the client ID into the message sent to the server. This allows the server to send the response back to the correct named client. After sending the response, the client awaits a response from the server. It does so by reading from the named client pipe. After printing the response from the server to the screen, the client process awaits further input from the user. The client also includes the additional commands, 'exit', 'shutdown' and 'help'. The 'exit' command terminates the client process and removes that processes' named client pipe. The 'shutdown' command stops the server.sh process and removes the named server pipe. I'll discuss the 'help' command under the 'Bonus' section of this report.

Concurrency

As there can be multiple clients sending commands to our functional scripts via server.sh at the same time, I needed to include a mutex to lock the critical sections of these scripts. For each functional script the critical section start and end is highlighted by comments. In brief, the critical section begins when we query if the database or table exists, and ends when we execute either writes or reads on the directory or file in question.

The script P.sh creates a linked file (the lock) using itself as the source file and the parameter passed to it as the prefix for the name of the lock. For instance, P.sh database will create a linked file, 'database-lock'. The lock will only be created if it does not already exist. The script V.sh deletes the lock associated with the parameter passed to it. That is, V.sh database will delete 'database-lock'.

Challenges

This is the largest programming project to date in this course, and as such it presented many new challenges. Not least of all was in determining what to tackle first. I have learned through this project though that it is important to break down any problem into its smallest parts, and if possible, break those down further. Even the largest projects are just the sum of their parts. I've highlighted some particular challenges that stuck out during the making of this project below.

Issue Reading Multiple Rows from Named Pipe

This was by far the most perplexing and infuriating issue I came across during my project. It was one that after discussions with both Anthony and Thomas, neither could come to the bottom of themselves. I'll discuss the issue in brief below, and then describe the workaround.

The bug was as follows. The first time I used a client to select data from a table, it would print each line from the client pipe successfully. However, subsequent reads would only print a variable number before stopping. The 'end_results' line would not be printed. I had thought it may have been a process synchronization issue but it would seem to have been ultimately an issue with using a while loop within my client script to read multiple lines from the pipe. Instead of spending more time and resources on this one issue, I decided to approach it from a different angle.

To resolve this issue, I changed my select.sh script to send back what was initially a multi-line response as a single-line response. I then format that single-line response into the expected multi-line result in my client script. As I was now only reading once from my client pipe, I had a workaround for my issue.

Error Handling (Client-side vs Server-side?)

We were tasked with providing adequate error handling for our system. In the requirements for the client, we are told it can take arguments in the 'req args'. As the server and functional scripts are already performing the majority of the error handling, I decided that my error handling would increase in specificity from client,

to server, to functional scripts. That is, users would only be prevented from typing one word commands (excluding exit, shutdown and help). All other commands would be sent to the server and attempted to be executed. This method allows for new commands to be added to the server (such as delete_database, delete_table) without code needing to be added to the client. I can therefore add new functionality without asking clients to update their scripts.

Interrupted System Call Errors on macOS

This issue occurred when testing locally on my macOS. My server script would print 'Interrupted System Call' errors when reading from my named FIFO pipe, server.pipe, unless I added a sleep 1 command to each of my server response cases. This was in fact only an issue on macOS and not on the 'production' Ubuntu server I tested all of my scripts on. When I uploaded my code to the Ubuntu server and tested without those sleeps, it worked as expected. I've removed the sleep commands from the submitted version.

Bonus Work

For my bonus work, I decided to add some functionality that I felt made the client more user-friendly, and I also added two new database features, delete_database and delete_table. To the server I added a useful debug mode that I made when developing and felt was a fun addition.

Client improvements

The smallest change I made that I feel adds to the client is the inclusion of a prompt ">>>". It's a visual indicator that the client is read to accept input. I also added welcome messaging, including a nudge to use the added 'help' command if required. Typing 'help' opens the README.txt and provides a full list of commands to the user.

Added Database System Functionality

I added two new functional scripts, delete_database.sh and delete_table.sh. These scripts function as follows:

delete_database.sh $database

This command deletes $database, provided it exists.

delete_table.sh $database $table

This command deletes $table from $database, provided $table exists in $database.

Server Debug Mode

Server debug mode prints what messages it receives from the client, along with a timestamp displaying when that message was received. To enable debug mode, start the server from the terminal with the DEBUG parameter: ./server.sh DEBUG.

Conclusion

In this report I have detailed the requirements and design of the simple server-client based database management system I have developed in Bash. I also spoke of some of the challenges I faced in its creation and how I resolved those. For me, this project goes to show how the basic building blocks of programming and operating systems can be built up to form a complex project, one that the developer can feel proud of.