Project Password

Management System 2019

Introduction

My understanding of the intention of this project is to fulfil two goals. The first goal is to apply what I have learned from the theoretical element of this module in a practical way, in order to improve my understanding of key concepts such as process management, threads and concurrency and scheduling with a view to how they are applied in a real life context. The second goal is to improve my programming skills and knowledge for the command language/Unix shell Bash, through writing relatively complex (for a beginner such as myself) scripts that interact with each other.

Requirements

The purpose of this system is to function as a password management system for multiple users. The system takes requests off multiple users (often at the same time) allowing them to create and maintain their own passwords in a secure and convenient way. If a user inputs an incorrect request, whether they have provided insufficient or incorrect information then they system will inform them of the exact issue to allow them to provide a valid request. The system should have some flexibility in allowing users to amend files they have already created, create sub folders within their own user folder and delete files when required.

The system is also designed in a way to avoid information being corrupted or delays to requests due to user processes interfering with each other when accessing shared resource e.g. if two users attempt to update a new file with the same name using insert.sh. This was achieved using semaphores, locking a service while a process is using it to prevent others process from accessing it at the same time, then unlocking it when the original process is finished to enable other processes access to same.

Design

In terms of handling invalid requests, my solution has numerous checks to ensure any given request is valid or not. This comprises the checking if the number of parameters provided is correct based on the service involved, checking if a username or service name already exists and therefore rejecting the request and also checking if a username or service name doesn't exist when they are required.

These checks are performed **prior** to any operation being done, the logic being that there is no point going further if the initial input by the user isn't right. Consequently, conditional statements were used for this.

To prevent data corruption issues associated with concurrency and synchronisation where multiple users may try to access shared files simultaneously, I created two scripts P.sh and V.sh to enable locking at critical sections of the code. I defined a critical section as any area where sensitive information was created, changed or viewed i.e. login and password details. Therefore, these scripts were input into my init, insert and show scripts.

The P script ran directly before any critical section, locking the file to prevent other users from accessing it. The V script ran directly after the critical section, serving to unlock the critical section to allow other users access to same.

My scripts allowed users to create, amend and view data one file deep. By that I mean a user could create a folder within their user folder and then create files within that folder.

A critical component of each of my scripts is structure, in that the structure should be well laid with appropriately named variables and logical flows of control. This is as much for others viewing the scripts as was is for me; it made debugging that bit easier when I could tell when exactly in my code where the error occurred.

Server.sh design decisions

For my server script, I used a while loop to allow it to read multiple requests send through the server pipe from the client script. The request would be read as an array, where I could easily respond back to the client through the server pipe using index 0 in the array (being the client id).

The server pipe was created before entering the loop and only removed once a shutdown request is sent through. This may be an issue if a client shuts down the server before another client is finished with it but for the purpose of my pipes it worked correctly.

Client.sh design decisions

In terms of the update request which was the trickiest part of client.sh for me, I ran show in the client script to assign the output to a variable and then echo it to a tempfile to show the current content. I realise this may not be sound design in terms of the client script itself running a script instead of the server doing it, but I made a pragmatic decision to do this to allow the overall update request to work mostly as intended.

The client pipe was created at the beginning of the script and deleted whenever a service request was fulfilled i.e. at the end of the case statement for any given script.

Challenges

1. I encountered an issue when creating the insert file of recognising when the user had a directory name prefacing their service file e.g. Bank/aib.ie. My original way using delimiters was cumbersome and led to problems recognising the directory and service. I remedied this using command substitution i.e. base and dirname to allow my script to correctly identify the paths given.

2. Update service was by far the most difficult part of the client script for me. Having to create a new payload based on the user's changes made on a nano file. I spent a lot of time on this part of the script but alas could not get the update to display correctly split into two lines. If I had more time, I would have tried solving this using the cut command to create separate variables for each line.

3. The start of part 3 in creating the server script was another stumbling block for me. This was my first time using both case statements and arrays in bash, so it was one of the most difficult sections to do in the entire project. I had a bit of trouble understanding case statements initially in terms of them only checking for value and not being a loop, but once I experimented a little with them it became clearer. And as was the usual case for these sorts of problems, once I got the first case statement working for init.sh script then I was able to add the other scripts relatively quickly.

4. For arrays, what made things easier for me in understanding them was using a for loop to echo out each item of my array, such that I could know which item corresponded to the argument I wanted to send to my sub-scripts. This helped me solve parameter problems I was having when sending slices of arrays as arguments in scripts, where outputting array values separately as I currently had it fixed this issue. I used arrays primarily in my server to read in requests from the client side; this allowed me to discard unneeded parameters such as the client id for example.

5. I surprised myself with how quickly I was able to implement my pipes, given I struggled to understand them in the practical session covering them. I found literally drawing out the process from the client side sending the request over to the server with a pen and a piece of paper helped clarify what I needed to do to get the pipes working as intended. Unfortunately, this didn't stop getting stuck for a while not understanding how one file won't finish running until the other files reads its message through its pipe!

6. Semaphores were a tricky part for me; I was able to get them implemented relatively quickly on some scripts but took much longer on others. For example, I had issues with insert.sh constantly locking using the file as the thing to lock, so I amended it to lock the user folder instead. I placed locks and unlocks only at the blocks of code where data was being changed

or created. The same was done for init.sh, where only critical section responsible for creating a directory was enclosed with a P.sh and V.sh script. I also only inserted them into init and insert scripts, as these are the only scripts creating/manipulating data compared to the others.

7. Disappointingly I didn't have time to implement the encryption system as an added feature. If I was to do so I would try the following approach: only implement them with insert and update. As they are the only requests containing sensitive information in the request so I feel encrypting the logins and passwords there would be appropriate. I would have run the encrypt.sh script in my client file with my arguments, assigning it to a variable and sending that variable to my server.sh script. Then decrypt it on the server.sh using the decrypt.sh file, for the server to then send the request to the respective service.

8. I had issues with the show file duplicating the display of the file. I realised this was due to the way I was using grep, as it would read the entire line where I was using it. I also inserted -i after my greps to allow it to account for both upper and lower cases in the login and password lines.

9. Overall the key part for making progress in this project was patience, taking each section at its smallest level and including echo statements as much as possible to help me better identify where bugs were occurring. I did not find Bash a particularly intuitive language initially and was a bit worried when starting the project. However, as I made progress and began to understand how the concepts we learned in the lectures underpinned the design decisions of the project, things became clearer and my confidence improved.

Conclusion

As mentioned in the project brief, taking the project step by step was the key and making incremental process allowed me to easier identify where bugs were occurring and debug effectively (that and a lot of echo statements!)

While I found this project initially intimidating and struggled with some parts, I found it overall very worthwhile in completing and learned a great deal in applying the concepts of OS management in a practical context.

I would like to thank you all for the help provided during this module, the lectures were very well taught and the practicals well laid out with every tutor providing excellent guidance and help throughout the entire process.