BashBook Project Report

Group members

- Diarmaid Mc Keagney 22321376
- Peter Fitzgerald 22323303

Introduction

This is the project report for the BashBook project which was developed for CT213-Computer Systems and Organization. The BashBook project is a collection of bash scripts which implement the basic features of a social media platform. The projects server script ties all the other scripts together and allows a client, running the client script, to interact with the server and carry out operations such as, creating a user, adding friends, posting messages to friends' walls, and displaying a user's wall. Throughout this report we will outline the features we have implemented, the choice of pipes we made, the locking strategy used to combat synchronization issues, and any issues we ran into along the way.

Implemented Features

We implemented all the features that were listed in the assignment description except for the user-friendly error messages which we have addressed further along in this report. Below we detail all the features implemented and the design choices we made when implementing them.

Creating a user:

The create.sh file was written by Diarmaid and allows the user to create their "account" in the program. The create file simply takes in an id which will be used to identify the user and sets up a directory by the same name. In this directory the friends.txt file and the wall.txt file are stored which keep track of a user's friends and wall respectively.

Adding friends:

The add_friends.sh file was created by Peter and allows the user to add a friend to their friends.txt file, it is important to note that friendship is not mutual so only the original user will add the friends name to their friends.txt file not the other way around. Once two users are friends it allows them to post messages to each other's walls

Posting messages to a friend’s wall:

The post_messages.sh file was completed by Peter and allows friends to post messages to each other’s walls. The user can only post messages to another user’s wall if they are on the receiver's friends list. The messages posted will be stored in the receiver’s wall.txt file and will need to be printed to the terminal by the display_wall.sh file.

Displaying a user’s wall:

The display_wall.sh file was completed by Diarmaid and allows the user to print any users wall.txt file in the terminal. This feature allows users to see the messages that other users have posted on their wall or any other user’s wall file.

Client script:

The client.sh file was completed by both of us in a pair programming environment. By running the client script, a user sets up their terminal to communicate with the server. The script takes in an id that will be used to create the user pipe that returns the output of a command run by the user. Within the client script any command typed into the terminal by the user is sent to the server pipe to be processed.

Managing concurrent users:

A quick comment on the handling of concurrent users, we had to implement a locking system that would lock each of the possible commands individually if a user had entered that command. This is needed to prevent errors caused by the server receiving two requests for the same command at the same time and possibly scrambling the requests or outputs. This locking feature means that the program can handle large numbers of concurrent users without any of the users receiving a nonsense output. This feature is discussed in more detail further along in this report.

Server script:

The server script was again written by both of us using pair programming. The server works by checking the server pipe and reading arguments from it. These arguments are then checked to see what feature the user would like to use, i.e. create, add, post, display. Depending on which is needed the server calls the script and passes the users arguments to it. The result is then returned to the user pipe that corresponds to the user that sent the request.

Deleting pipes after user closes the program:

This feature allows the program to close the pipes that are created for each user once they press ctrl + c to stop the program. It does this by trapping the signal interrupt command and instead calling a function that removes the pipe and exits the program with code 0. We implemented this feature separately in the server and client scripts with Diarmaid doing it in the client script and Peter doing it in the server script, but we both did it in the same way.

Named Pipes

We implemented two named pipes in our project. The pipe used to send requests to the server is called the server pipe. All client scripts send requests to the server via the server pipe. The second pipe is a dynamically named pipe. It is created when the client script is first called. The pipe is named [USERNAME]_pipe. The username is passed in as an argument when calling the client script. The server sends the response back to the client via their specific pipe.

Locking Strategy

For our locking strategy we used a spinlock by trying to create a link to a file in our program folder. If the link could be created, then the user that created the link would acquire the lock and no other user could carry out the same operation until the original user was finished. Once the user completed the operation the lock was released so other users can now carry out that operation. If a user tried to use the program while a different user had the lock the program will just sleep for 1 second and then try and acquire the lock again.

This locking strategy performed quite well in practice but one possible issue with it is if users try and acquire a lock at the same time on user will get the lock and the other user will be forced to wait for at least one second before they can carry out their operation even though the first user might only have the lock for a few milliseconds. This could cause delays for users when carrying out operations although the likelihood of this happening is exceptionally low unless there are many concurrent users. This locking mechanism worked well in preventing synchronization issues, as in testing we had no issues arising from two users entering the same command at the same time.

We had four different lock files, one for each of the four operations that a user could send to the server, their names are: addlock.txt, createlock.txt, displaylock.txt, and postlock.txt. We chose their names this way, so that it is clear which feature they are locking, i.e., the create operation is locked by the createlock file. This naming system also made it easier to read the client script as the names of the lock make it clear which function we are checking for.

We used the locking mechanism in the client script just before the command gets piped to the server. We chose to have the lock here as the critical section is the sending of the information to the server, if we have two users send information to the server at the same time the requests will be merged or only the second user will get a response, so we needed to control the flow of information to the server. The lock simply prevents a user that doesn't have the lock from sending information to the server but once the lock is released, they are free to acquire it and send their request.

Challenges We Faced

We faced many different challenges when coding this project, more than could be written down in this section. However, here are some of the major challenges we faced.

1. When developing our locking feature, the acquire.sh script had a hard time finding the lock file. We tried using "/locks/[specific lock we wanted].txt" as the address, but this didn't seem to work. How we fixed this problem is we used the pwd command to get the current working directory and saved that address to a variable. Then we could use the full address of the lockfile by using "$currentDir/locks/[specific lock we wanted].txt"(note: we passed in the {/locks/[specific lock we wanted].txt} from the client script instead of hardcoding it)

2. When we were developing the post_messages script, we ran into an issue where if the message the client wanted to send contained spaces, then the server would only send the first word in the message to the target wall.txt. This was due to the way we handled input from the user. The users input was seperated into an array with the delimiter being a space. This made it easy to handle arguments sent by the user, but made it difficult to send an entire sentence to the server. We fixed this by first turning the message from a series of separate words in an array, back into a single string using a for loop. Then we were able to send that string to the desired wall.txt.

3. When developing the display_wall script, we ran into an issue where only the first line of the wall.txt would be displayed. This was due to how the client script read the response from the server. It would only read the first response from the pipe. To fix this problem, we changed the client script to include a while loop that would run as long as there was still text in the pipe. The while loop would continuously echo the contents of the pipe until the end of the text.

Unimplemented Features

Due to time constraints, we had only one feature which we were not able to implement into our project. This feature was the user-friendly responses from the server, we decided that this wasn't an especially important feature and as such, it had a low priority in our work. We implemented the most prominent features first and we ran out of time to implement the user messages. If we were to have implemented them, we would have had a case statement in the server file that looks for the response from each of the feature scripts. Then we would simply convert each of the messages into a user-friendly format when we caught the script error message.

Conclusion

In conclusion we ran into a lot of issues with this project but overall, this was a fantastic opportunity to work in bash and to solidify the learning we have been doing in lectures throughout this semester. We managed to get the project polished, and we learned a lot about bash and how it handles different operations over the project. Overall, we are happy with the result of the project other than the omission of the user-friendly messages, and if given some more time we could have solved that issue also. Thank you for reading this report about our project, and we hope it clears up any questions about our design choices throughout the project.