Bash Project 1 Summary

The goal of this project was to make a Facebook-esque program in Bash. It started off with making four scripts which correlated to four user activities in Facebook – creating a user (create.sh), adding a friend(add_friend.sh), writing a message (send_messages.sh) and displaying your wall (display_wall.sh).

The first script was to create a user. The first question is how does the system get the user name? I set it to an argument where id="$1" – the first argument or string written after calling the script. Then we had to check a few things. Firstly, if there is nothing written after calling the script, then there is no argument. The system checks this with an if statement which checks if the number of arguments is equal to zero. If this statement is satisfied, then the system must echo "ERROR. No user provided." and then Exit 1. In this case we use exit(1) because an exit with a non-zero value means that the program was terminated with some sort of error. We also must check if the user already exits. This is done via another if statement where the script checks the directory for the first argument (the user name). If one is found, then the system echoes "This user already exists." And exit 1. Finally, the system will check via an if statement if the argument, the user id, is not already in the directory (! -d). If this is the case, then it makes a directory with the name of the given argument. Within this directory with the touch command it makes a wall and a friends file and then echoes "User Created". Then the if statement closes and exits 0.

For the add_friend.sh, we need to tell the system two bits of information – the id of the user and the friend they want to add. These labels are added to the values of the first two arguments. Again, like in the last script, the system must check if the user doesn't exist via an if statement, but this time, if true instead of creating a user, the script sends the message "ERROR. This user does not exist". And exit 1. We also must check if the person we're trying to if the friend doesn't exist with the same sort of if statement but instead of using argument 1 ($id) we use argument 2 ($friend). If both arguments already exist in the directory, we must check for one more error and that is if the two people, id and friend, are already friends. I knew about the grep command – essentially a sort of control+f – but we were given the full command to check and see if the second argument, the friend, was already listed in the first argument – the id's friend file. If true and the same name is found, then the script prints out an error to this effect. Finally, if none of these errors come to fruition, then the second argument "$friend" is echoed >> into the first argument "$id"'s friend file. Because I wanted to synchronize the friendship process like in real Facebook, I also wrote a command that if there were no errors, then the first argument "$id" is echoed >> into the second argument "$friend"'s friend file.

The third script, post_messages.sh takes three arguments – a sender, a receiver, and a message. The sender and receiver are argument 1 and 2 respectively, but the message argument poses an interesting problem. In its simplest notation, only one string can be assigned to an argument. These strings are usually determined to end with a space. However, we don't want just one word messages. So we have to change the notation of the third argument. So instead of having sender="$1" receiver="2" message="$3, I changed the notation to ${@:3} where ${} means parameter substitution and can be used to concatenate variables with strings, @ means the parameter is a quotes string and that the parameter is passed intact without interpretation or expansion, and 3 is the third argument. This allows us to print a multiword message to a person's wall.

This third script checks that both the sender and the receiver existed and printed errors and exit 1 if they did not. It also checks that the two users are friends before messages can be posted. For checking if someone was friends or not, I tried another approach by grep (to find) -fx

“~/facebook/’$sender’/friends/’$receiver’ “ where f is for fixed-strings and interprets a pattern as a list of fixed strings (in this case, I was hoping that would correlate to letters/words) and x where x stands for line – regexp which selects only those things that match the whole line, which, in the way that the friends files is set up, would be an entire name. These commands came from the man page for grep. In this case, I was trying to find a specific user name – that of the receiver, in the sender’s friends file. That way, if it didn’t exist, I could implement the echo ERROR. You must be friends to send someone a message. This method did not work very well as the servers was failing to recognize a valid exit code (even though the exit code was, as usual, at the bottom of the script. I just decided to repeat it the way outlined in the second add_friends.sh script. If the two users both exist and are indeed friends then the script echoed the sender’s name, “$sender”, and sent a message. That message was passed into the receiver’s wall file.

Finally, the fourth user script’s purpose was to display a user’s wall file. In this case, the script must first check that the user does indeed exist. It gets the user’s name from the first argument wherein id=$1. It runs an if statement - if the id is not found in the directory, then an error prints and exit 1. Else, it indicates the start of the file by echoing so (thus displaying “start of file” on the screen. It then concatenates with the id in question (the first argument)’s wall file. This will print the contents of that file to the screen. Finally, it echoes end of the file so we know where the wall ends. Then it exits the if statement with “fi” and then exit 0.

For part two we had to make a server script in which the user via the client script and server pipe outlined later in this paper merely has to type the words create, add, post, and display to execute the scripts create.sh, add_friend.sh, post_messages.sh, and display_wall.sh respectively. I used a case statement to do this. I had the while loop read four arguments. At first, I had these arguments with a $ in front but this is incorrect because it’s just reading the variables as opposed to taking the values (in bash, if you want to assign something a value, you use the $). The read arguments were command, id, friend, and message. The names don’t matter, merely their position, but whatever you call what you’re reading, you must match those names to the parameters used later in the scripts because this loop will be passing whatever its reading into those scripts. Then I said case “$command” in, followed by a list of the four user commands that would cue the four different scripts – create, add, post, and display, respectively. This command is the first argument read because it is the first argument called in the loop and opens the other scripts. The other arguments correspond to the arguments read in order at the top and they are now being passed through the loop and being assigned and applied to the scripts. Finally, if a user typed in a command that didn’t match the previously outlined commands, then the server script would echo “Usage: $0 {create|add|post|display}” then exit 1. This is basically telling the user that their command was bad and that those commands in the curly brackets are the only ones recognized. Anything apart from those words was denoted in the script by a *. Now, this case statement must be closed, but it cannot be closed with “fi” like an if statement nor a simple exit 0. It is instead closed by “esac” which is case spelled backwards which I literally just realized. We also had to write in a statement that would catch a bad request and echo back “not okay. Bad request”. I attempted to do this in the same style as the previous four scripts checking if users existed and printing errors if they did not. I also attempted to do this via an if loop that checked if the $# (number of arguments) was equal to zero and then print an error statement – exactly like in create.sh. However, any way I attempted to implement it caused the script to have an error and stop running so I deleted all attempts and kept as was. I am almost sure that it was a small syntax error and, if I had more time, would have dedicated much more time combing through and tweaking the code to get the error to run. As is though, it works well enough.

The next step was putting in the locks. We use locks when we need to prevent two or more copies of a script running simultaneously. This is pertinent to this project because this server script is running multiple processes concurrently with various clients accessing the server. The problem with concurrent execution is the possibilities of inconsistencies – like what happens when two commands attempt to access the same file simultaneously. The commands could fail or corrupt data. To prevent this, we put locks on the necessary repositories and files. The locks I used were semaphores with script p.sh to allow a process into a critical section. The p.sh script would be put right before the start of the critical section. The script that allowed a process out of a critical section is v.sh. This script is placed right after a critical section of another script. These scripts are called within scripts. For example, In the fourth display_wall.sh script, the critical section of the script – where it all happens – is where it echoes the start of file, concatenates the $id/wall file and then echoes end of file. Right before those commands are listed, I inserted a ./p.sh "$id"_lock and, right after, a ./v.sh "$id"_lock. The ./ executes the following named scripts and I used the "$id"_lock because the argument I'm locking is the id – two ids cannot be displayed at the same time for the same client, and lock because that is the function of the code. The p.sh and v.sh were given to us in class.

Locks were used in all of the first four written scripts. In create.sh, p.sh was implemented before the critical section starting with the line that makes the directory with the id name in the facebook folder, and v.sh was placed after the friends file in said folder had been created. The argument locked was the first and only argument – the user id.

In add_friend.sh there were actually two sets of locks because in my program, as previously mentioned, friendship is synchronized. Here a p.sh "$id"_lock was implemented before the script echoed "$id" into "$friend"/friends, but also a "$friend"_lock corresponding to the second argument was put in place before this critical section because right after $id is echoed into $friend/friends, then $friend is echoed into $id/friends. Both parameters must be locked because both are doing the same thing so both could suffer problems if they were simultaneously accessed. Again, ./v.sh with both previously stated locks is called after the echo confirms that "$id" and "$friend" are now officially friends.

Finally, the lock setup of post_message.sh acts exactly like the previously outlined add_friend.sh in that there are two arguments that need to be locked - $sender and $receiver. The p.sh script locks are put in place before the script echoes the sender's name and message into the receiver's wall file, and the v.sh script locks are put in place right after. The other parts of the script such as the other if statements do not need to have locks because they are not the critical sections, merely error checks.

The final part of the project involved making the client script – client.sh. This is where the user would enter their commands and receive corresponding output to and from the server. This script takes one argument and assigns it to the variable client. It then calls a trap function. A trap function basically allows a user to exit an endless loop in another manner apart from control+c. In this case, we set it to exit 0 in which typing 0 would exit the loop. The skeleton was given to us in class. In the function, I had it remove client pipe and then exit 0. Then the script must check if there is a user provided. As in create.sh, it checks via an if statement if the number of arguments is equal to 0. If so, it prints an error and exit 1. It does the same thing if there is more than one argument, in this script, defined as clients, because more than one client cannot run the same script. If this is attempted, the script checks and then will print an error and exit 1. Otherwise, if there is one and only one argument/client, then the script will make a pipe called $client.pipe (named after the argument). This pipe is temporary, as ensured by the trap function, and will be deleted as soon as the client ends client.sh.

The pipe is made with the command mkfifo. The script then implements a while loop and reads the command that the client inputs (must match the previously outlined commands in the server.sh) and the provided arguments. Then it will echo the command, the client, and the arguments (all with $ because the values are what's being echoed) through the server.pipe. It also reads lines that come through the $client.pipe (read line <$client.pipe;) and then echos the values of those lines. Those lines are coming from the server and are output generated by the server from this client's and other clients' commands.

Server.sh also has a trap function and a pipe. In this case, the trap function removes the server pipe via an if statement and exit 0. As far as creating pipes, the server.sh script requires a pipe to run properly and therefore I have implemented an if statement that checks if there is a server pipe and, if not makes a server pipe. Then, the case statement, where server.sh reads the different parameters, the server then receivers those parameters through the server pipe (read command id friend message < server.pipe) and also sends the echo or output of each client-called command through that client's own pipe for example, for the add command echo $./add_friend.sh $id $friend) > $id.pipe.

Pipes work as such: As a client, a command is entered into the prompt and it goes through the server pipe to the server. The server script server.sh then reads the server pipe – the command and arguments from the client and takes these arguments to run the commands. This server script then outputs whatever output it gets from running the command and sends it back to client.sh via the client pipe. Client.sh then reads the output and prints it to the terminal. In theory, you could have 6 client scripts open and working with one server and the server will go through them and give each individual output because each client has his or her own access to the server and own pipe from the server and there are locks on the parameters that require them. The most difficult part of crafting the pipes was melding them syntactically with the pre-existing server and client scripts. Determining the parameters the client script was sending through the server pipe was particularly challenging. To overcome this, I had to map out the logic and do loads of trial and error. And, as for everything, syntax was challenging – a missing curly bracket or a space between "$id" and _lock resulted in many errors, specifically the dreaded "syntax error unexpected end of file". Eventually, using the tools we learned in class and a lot of googling, these errors were overcome.

In conclusion, this project taught me a lot and was a very interesting and challenging way to implement and apply many of the codes, scripts, and concepts we learned in class. I found the most challenging part to be the syntax, of which some errors would set me back hours, but hopefully this will improve with practice.