BashBook: A Facebook-Like Social Media In Bash

Group Members:

- Ben Naughton
- Igor Pokrzywniak

BashBook is a bash focused set of scripts that are meant to simulate a social media. There are a total of 4 feature scripts, 2 scripts for handling synchronization issues, a client and server script along with 2 minor scripts for verifying the integrity of arguments. This project used the concept of piping to simulate an exchange of information of data between clients and the server. The exercise gave us great insight into how operating systems function and how applications can use basic commands to construct robust applications and software.

List of implemented features:

- **CreateUser** – The create user script takes in one argument which is the name of the user. It acquires a lock (to make sure two users with the same name will not be created at the same time) and verifies that the user does not already exist. Once it does that, it checks for the number of arguments passed in, and if it is correct, it creates a directory named after the user and creates a txt document for messages (wall) and friends. The user also adds himself to be able to message himself (post messages to his wall).

- **Add_Friend** – The add friend script takes in two arguments, the ID of the person that is sending the invite and the name of the friend that the invite is sent to. Once again, we verify whether both users exist, and we lock the friend's files. If both users exist, we can add each other to their corresponding friend list – we add friend1 to friend2's friend list (friends.txt document) and we add friend2 to friend1's friend list.

- **Post_messages** – The post messages script takes in three arguments, which are: The ID of the user that is posting the message, the receiver of the message and the message itself. The script verifies whether the sender and receiver exist, if they do it checks if both of them are friends. It uses the exit code of the check_friends script to decide whether to exit the code or to pipe a message in the format: {Sender}: {Message} to the receivers wall.txt file.

- **Display_Wall** - The display wall file accepts 1 argument which is the ID of the user. The ID of the user is passed into our server script through the client script, so technically it does not require any arguments to be passed in from the client side. It verifies whether the user exists. Next it sets the internal field separator to underscore (switches how bash handles word splitting). It initializes an input array to "Start of file". It reads text from the current user's wall and appends them to the input array, each line is split by an underscore. Once it is done, it appends "end of file". Lastly it expands our input array into one string where each element is separated by our IFS (underscore) and the output is sent into the pipe called after our user.

- **Client** – The client script takes in one argument which is the name of the user. The script checks whether the parameter has been passed in, it then checks for the ID, if it exists. If it does not exist, the script calls the createUser.sh script with that parameter (this is an additional feature we added). Next, it checks whether our server script is running, and starts the server up if it was not running previously. Once the ID is verified and the server is started a pipe is made named after the ID of the user, this way each user will have their own separate and unique pipe. This pipe is trapped, so that when the client is abruptly shut down the pipe is removed. Next, the login feature is called which increments our number of current clients accessing server and then it gets response from server, that our client has logged in. After all this is done, the client is prompted for a command to access one of the main functions of the **bashbook**. It sends the arguments to our server and then reads the return through the user ID pipe. The script checks the returned value against our grep to see whether it returned an error, a wall of text, an exit or something else and adjusts its output to the user depending on what it received. This is done in a loop, so the user can continue to be utilizing the services as long as he needs without having to restart the client.

- **Server:** The server script takes no arguments. It starts by creating a pipe called server and trapping the script so if it's shut down it removes the pipe. It initializes a counter to 0 which is the number of clients connected to it. The server waits for input to be delivered to the server pipe which would include type of request with arguments. Then it checks whether the request is valid and if it is, it runs its corresponding script. Depending on the output of the script it pipes a properly formatted response to the client side through individual clients ID pipe. The server also has the login feature that is called on the client's side when initializing the client. Lastly, the server has an exit feature that allows the server to be closed once there are no more clients connected to it. This is checked by incrementing the counter whenever user is logged in and reducing it whenever the client exists.

- **Check_friends:** This Bash script is used to determine the friendship status between two users in a system. It starts by assigning command line arguments to variables user1 and user2. The script first checks if both users exist in the system using another script check_Idss.sh. If either user does not exist, it exists with status 1. Next, it checks if user1 is in user2's friends.txt file, which lists user2's friends. If user1 is found in the file, indicating they are friends, the script exits with status 0. Otherwise, if user1 is not in user2's friends list, it exits with status 1, indicating they are not friends.

- **Check_Ids:** The check_Ids script checks for the existence of directories corresponding to user IDs passed as arguments. If no arguments are provided, the script exists with a status code of 1, signaling an error due to lack of input. When arguments are given, it loops through each one, checking if a directory with the name matching the user ID exists. If a directory for any user ID is not found, the script exists with a unique status code based on the sequence of the argument (starting from 2). The script successfully exists with a status code of 0 only if all specified user directories exist.

- **Acquire:** This checks for the existence of directories named after each provided user ID and exits with a specific status code based on whether these directories exist.

- **Release:** This script checks if a command-line argument is provided. If no argument is given, the script exists with a status code of 1, indicating an error. If an argument is provided, it uses the **rm** command to delete the file or directory named in the argument.

Piping:

- *Our program only has two types of pipes:*
    - The "**server**" pipe – the pipe that the clients input their information into and then is read by the server script. The server pipe reads the users ID, request, ID of their friend and any additional arguments (such as the message for **post_message.sh** script)

    - The **"ID"_Pipe** pipe – this is the pipe that is created by the client, once their ID is verified as correct, unique and the server has started. This pipe is responsible for delivering unique information to users through the server script. For example, a request for "display" will be delivered through a pipe called Tom_Pipe for user "Tom" and for user "Bob" it will be delivered though a pipe called Bob_Pipe. This is to ensure that no user gets information from another client and so that there will not be any synchronization issues.

The "**server**" pipe is self-explanatory, and we only need one. For the ID pipes we used a concatenation of the ID with "_Pipe" to make it simple to understand and unique. The names of the pipe were inspired from the diagram below.

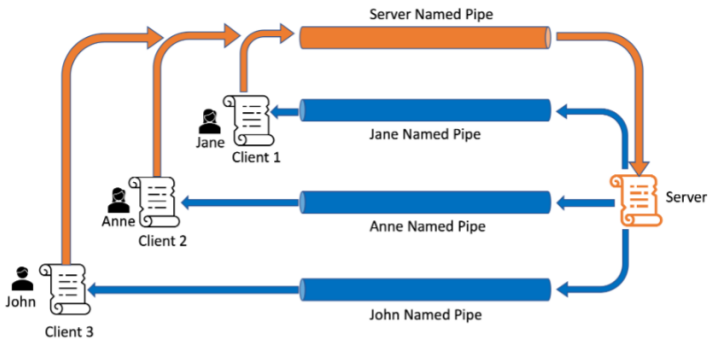

Figure 1: Overview of BashBook communication architecture

To conclude the BashBook project highlights the fundamental ideas of file handling, synchronization, piping, and interprocess communication using Bash scripting to simulate the building blocks of a social media. Using basic command-line tools and techniques such as acquiring-locking, data piping between

clients and a server, and user interactions through terminal, the project provides invaluable knowledge about building reliable applications in the Bash environment. BashBook demonstrates how simple commands may be combined to create useful and effective software solutions within a command-line interface with its user management, messaging, and display features.