CT213: Bashbook submission.

Iarla Roche (23747889) & Ethan Aguocha-Njoroge (23480984)

Introduction

We were tasked with creating a simple server-client communication system using bash scripting. The goal was to simulate a basic social media platform called BashBook, where users could create accounts, add friends, post messages on their friends' walls, and view their walls. We implemented a server that listens for client requests and processes them, ensuring that operations are synchronised and executed correctly. We faced several communication, concurrency, and synchronisation challenges throughout the project, which we successfully resolved through teamwork and learning.

Implemented Features

- Create a User Account: Users can create an account by providing a unique identifier. If the account already exists, an error message is returned.
- Add Friends: Users can add other users as friends. An error message is returned if the user or the friend doesn't exist or are already friends.
- Post Messages: Users can post messages to another user's wall, but only if they are friends.
- Display Wall: Users can view their wall, showing all their friends' posts.
- Locking Mechanism: Ethan implemented a symbolic link-based locking mechanism to avoid race conditions and ensure that only one client can perform critical operations simultaneously.
- Auto-Cleanup of Named Pipes: Ethan ensured that the named pipes used for communication were automatically deleted when a user quits the script, preventing leftover pipes and errors in future executions.

Description of the Named Pipes and How We Chose Their Names

Named pipes were used for communication between the server and clients. Each user has a unique pipe named $id.pipe. This allows the server to send responses to each client through their individual pipes. The server also uses a central pipe called server.pipe to receive client requests.

- Server Pipe (server.pipe): The server uses this pipe to receive all client requests, acting as a centralised entry point for all commands.
- User Pipes ($id.pipe): Each user has a separate pipe named with their user ID (e.g., 123.pipe). This ensures that responses are sent to the correct client.

The names were simple and descriptive, as each user's pipe is tied to their unique ID. This makes it easier to route messages to and from individual clients.

Description of the Locking Strategy for Each Request

The server.lock is a mechanism that prevents multiple clients from performing conflicting actions simultaneously. Here's how it works and why it was given this name:

Before a client can perform a critical operation (such as creating a user or posting a message), it must check if the lock is already in place. If the lock file (server.lock) exists, another client is currently performing an operation, so the requesting client must wait. If the lock file does not exist, the client

creates the server.lock file. This indicates that the client has acquired the lock and has exclusive access to the server's resources for the duration of the operation.

Once the lock is acquired, the client can safely perform the desired action (e.g., creating a user or posting a message) without the risk of conflicts or race conditions with other clients.

After completing the operation, the client deletes the server.lock file, effectively releasing the lock and allowing other clients to acquire it and perform their own operations.

- Why the name server.lock? Ethan chose server.lock because it indicates it controls access to the entire server. It prevents race conditions by ensuring that only one client can perform a critical operation (creating a user or posting a message) at a time.
- Where we used it? The lock is applied when a client wants to perform any operation that could conflict with another client's request, such as creating a user or posting a message.

Before sending any request to the server, clients must first acquire the lock using acquire.sh. If the lock is already in place, clients will wait until it is released. Once the operation is completed, the client releases the lock using release.sh, allowing other clients to proceed. In the incident that the client terminates their session before the server can process their request, their lock is automatically released along with deleting their pipe.

Challenges Faced

1. Message Passing Issues Between Server and Client:

   Initially, we struggled with getting the server to send clear success or error messages like "SUCCESS: user created!" or "ERROR: user exists" to the client. The problem stemmed from the server incorrectly echoing responses directly to the appropriate user's pipe, and the client couldn't effectively retrieve the required information. This was resolved after the pipes were reworked.

2. Maintaining the Infinite Loop in the Server Script:

   A significant issue was ensuring the server could process multiple requests without shutting down after handling just one. The server needed to remain operational to listen for new requests continuously, but initially, it would stop after processing a single request. The fix ensured the server script was set to run in an infinite loop, allowing it to process client requests indefinitely. Additionally, we learned that the server and client had to run in separate terminal windows, with the server in the background. This resolved the confusion and allowed the server to handle requests continuously.

3. Locking Mechanism and Handling Client Requests:

   Handling concurrent client requests was a critical challenge. We had to ensure that multiple clients wouldn't interfere with each other when interacting with the server, especially when accessing or modifying shared resources. To avoid race conditions, we implemented a locking mechanism. When one client interacted with the server, it would have exclusive access to the resources, preventing other clients from accessing them simultaneously. This ensured the server's resources were accessed sequentially and safely, preventing conflicts.

4. Server-Client Communication and Synchronization:

Ensuring reliable communication between the server and clients was another challenge. At first, it needed to be clarified how to route responses correctly from the server to the client, mainly when multiple pipes were involved. To address this, we focused on synchronisation using the named pipes. Each client had a unique pipe, and the server wrote responses directly to these pipes, ensuring the correct response reached the right client. This allowed the server to handle multiple requests in sequence without losing any messages.

Extra Challenge Implementation

Ethan tackled the extra challenge where we ensured that named pipes were automatically deleted when a user quit the script (e.g., by pressing Ctrl+C), preventing orphaned pipes from remaining in the system. This was achieved using the trap command to catch the Ctrl+C signal and automatically delete the named pipe before the script exits. This cleanup mechanism was critical in preventing the buildup of unnecessary pipes, which could have caused errors in subsequent executions.

Features Not Implemented & How We Would Implement Them

1. **Enhanced Error Handling:** While basic error handling is implemented, we could improve it by providing users with more detailed suggestions or solutions when something goes wrong. For instance, if a user attempts to add someone as a friend already on their friend list, the server could suggest that they're already friends instead of just returning an error.
2. **Message Formatting:** Currently, posts are plain text, but it would be cool to post videos and pictures.

Conclusion

Working on this project was a valuable learning experience, especially regarding bash scripting and handling communication between a server and multiple clients. We learned how to implement a locking mechanism to prevent race conditions, how to handle server-client communication via named pipes, and how to tackle issues such as ensuring that the server remains active and reliable throughout. While there are additional features we would implement with more time, we are proud of the work we accomplished. By collaborating on both the technical and conceptual challenges, we were able to build a functional system that simulates the core features of a social media platform, even under complex scenarios.