CT213 Computer Systems & Organisation

Assignment 1: BashBook version 2.0

Team – Maxwell Maia & Cathal Lawlor

Maxwell - 21236277 Cathal - 21325456

Introduction

We were tasked with this project of BashBook v2.

BashBookv2 – Building upon the previous BashBook we had built but employing synchronisation and pipelining to allow multiple users to use the platform at once.

We took an approach of programming segments of the code in the computing systems labs. We found this was the most productive as we could get a solid understanding of the blocks we'd made, and having manageable developments to explain to each other, while not overwhelming ourselves.

Maxwell had a very good grasp in the synchronisation side of bash and so developed that part of it. He also laid down the foundation for the pipelining where Cathal developed the pipelining to suit exactly what was going to occur along the $id.pipe (user pipe). Cathal then also handled the error handling for making server errors human readable.

Organisation

The program's server server.sh is run separately (but still locally for the ease of demonstration), to the clients client.sh script which is all based locally on the machine. After starting the server and client scripts, all commands for the user are run through client script with a more user-friendly type interface. Note: When started the client.sh script creates a pipe for the id provided, when the script is exited the pipe is deleted.

A user passes a request and arguments into user client.sh interface. It checks the request is well formed, where it is then sent down the server.pipe for the server to read it.

The server interprets this request and then finds the suitable case. In the case it checks if the user has passed in their own ID, for example to display their own wall. It then does the suitable method for either two different ID's or the users ID for both fields.
It passes the necessary ID's and arguments into a script, which it passed down the pipe of the user. This is called $id.pipe, with $id being the user's ID.

The server script that can call on the rest of the scripts to do operations within the program. When used in conjunction with the client.sh script this makes the user interface / commands a lot more uniform for the user, e.g. no need to keep typing new filepaths, but also allows multiple users to employ the servers functionalities at once.

All the files are in the one directory, making it much easier for us to work with. Our bash scripts and user directories are found in the "BashBook" directory. Each user has their own directory in the "BashBook" directory. When in use, unique user pipes will appear as well as locks pertaining to the user when an operation is being carried out.

This organisation approach is not scalable at all, but it works currently for its desired purpose.

Features

Script - ./client.sh.

The command “./client.sh Maxwell” will run the client.sh script.

Arguments: [user id]

It creates a pipe $id.pipe with the $id being Maxwell. This is for the server to have a pipeline to feedback the outputs it generates to the user and only the user. At this stage the directory for the id specified could be already exist or this could be a new user which means that the user directory doesn’t exist yet.

The client script then goes into an infinite loop allowing the user to type commands in much more friendly manor than typing server commands. It also automatically passes the id of the user (passed in from the argument earlier) to the server when needed. This will allow the server to know who to send a response to.

Requests

Create a user – create
Add a friend – add [newFriendID]
Post a message – post [receiverID] [message]
Display a wall – display [id]

Script – ./server.sh.

“./server.sh”

Arguments: none

No conditions are needed.
It stays in a constant loop, checking for any new requests being passed down the server.pipe. If there is a new request it acts upon it and returns an output down the specified $id.pipe (id is obtained in the request).

Specific features of the program:

In the server script:

A previous input check to prevent the server from executing the same request over and over in a loop. Maxwell.

Avoiding a deadlock. If the user wants to view their own wall, attempt to add themselves as a friend or display their own wall, the server cannot lock the same id as it would be acquiring the same lock twice (deadlock). So a check is added to create a lock on only one id. (more clear in the commented code). Maxwell.

In the client script:

A previous input check that will send a special refresh command to refresh the previous input variable of the server. This allows the client to send the same request multiple times in a row without an error occurring due to the server not recognising that the same request is instead actually, a new request. Maxwell.

A validity check makes sure that each request that the user types is valid. The first word of the request must be either post, add, display or create. Create needs no more than the word “create”. Post, add and display need to have further arguments. If these arguments are present the request is valid enough to send to the server. Further validity checks are done in each command’s script. Cathal.

Cathal did most of the pdf.

Synchronisation – using acquire.sh and release.sh

We tackle the issue of synchronisation by creating a lock for every client/user $id that is involved in a particular operation. Some operations will only lock one user though. Details included below.
Locking is done by calling the acquire script with a specific id as an argument before the operation script in the server. After, it's followed by calling release script with the same argument after the operation.

Using this method means that each operation for a user is completed by the server atomically and result is returned to the user when the server has completed it.

Description of locking strategy for each request

Create

Description: The create command creates a directory for the user and then a wall and a friends file. Once the user directory is made, other commands can attempt to access the wall file or friends file, which may not be present.
The whole create command needs to completed before another command can interact with this user, so we need to lock the client before the create command is run. At create exit, lock is released.

No other command should be run which accesses the resources of the client i.e., the wall or friends list should not be accessed by another command during the creation of these files. The entire client needs to be locked.

Performance: Good, considering that other users can be created concurrently because the lock is id dependent.

Correctness: This lock ensures that errors do not occur when other programs attempt to access a user being created. The files will be created properly as no one will access them. They will be correct.

Name: "$id"_lock - Why that name: it is a client dependent name which means that other clients can have their requests processed concurrently.

Why that level: all files in the client directory needs to be locked.

Add

Description: The add command adds a friend's name to the "friends" file of the client's user directory.

Example hypothetical error case. The client is Jamie, where there is a user named "Max" who is not a friend of Jamie. Max should not be able to post to Jamie's wall because "Max" is not in Jamie's friends.

Let's say that 2 requests are made at the same time.

Request 1: Jamie adds Maxwell to Jamie's friend file. Jamie is in the process of writing "Maxwell" to his friends file with an add function.

The program commences writing the characters of "Maxwell". It writes the characters M, a, x but then a context switch occurs and request 2 is allowed to execute before returning to finish request 1.

Request 2: Max posts to Jamie's wall. "Max" WOULD be able to post into Jamie's wall even though Max is not a friend of Jamie. This is because the add function hasn't finished writing Maxwell. The string "Max" was in the friends file when the post command checked if Max was a friend of Jamie. Therefore, the add function is the critical section, necessitating why we need to lock the client in the add function.

This prevents the special case where users that aren't friends with Jamie are able to post to Jamie's wall during the add function's operation.

In hindsight we could’ve have just locked the friends file, as it is the only file that has any changes made in it with the add function.

Performance: Good, because other users can use the add function, also poor, due to the case where, no one can do any requests involving the client’s wall which is an unnecessary thing to lock (hindsight).

Correctness: Ensures that the friend is added, and the post command doesn’t retrieve incorrect information.

Name: "$id"_lock - Why this name: Lock the client so that the friends file can be updated.

Why that level: In our approach, it is simpler to code a lock for the whole client whose files are being changed.

A better approach would be to make the name of the lock “$id”_friend_lock which would mean that we would have to code locks for both the friend and the wall.
The advantage of that upgraded approach would be that requests that access the wall of the client may be run whilst the friends file of the client is locked. The downside is that the system is more complex to design.

However, this was only realised post completing the coding so… next iteration perhaps :)

Post

Description: The post command checks if the sender is in the receiver’s friends file. If so, it concatenates a new message to the receiver’s wall. Resources of the receiver are being accessed (friends file) and updated (wall) so the whole receiver needs to be locked.
The sender’s client is also waiting for a response from the server, so (like all scripts) the client id needs to be locked too. Create a lock for every client/user $id that is involved in a particular operation

Performance: Potential for improvement if more code and information in requests were implemented, allowing requests to be returned to the client in any order [then we wouldn’t need to lock the client, allowing other users to perform operations with this client concurrently]. This would be at the cost of more lines of code for the server to run.

Correctness: Ensures that the updated wall is correct (no two post commands can run on the same wall due to the receiver lock). Ensures that the server responses are returned in the correct order to the client (due to the sender [client] lock).

Name: "$id"_lock and "$id2"_lock

Why that name: We want to lock the sender (the client, id1) and the receiver (the friend, id2). We don’t want any change in the friend list while we are searching through it with the grep function, necessitating locking the receiver. We don’t want two people to write into the wall at the same time.

We want the server to return the request responses in the right order to the client (hence “$id”_lock).

Why that level: For the sender lock, it prevents other commands from running since all commands acquire the client id. For the receiver lock, both the friends file and the wall need to be locked.

Display

Description: The display command returns the wall of a user to the client. We don’t want the wall of the user to update as we send it to the client. We want the server to return this request responses in the right order to the client. Create a lock for every client/user $id that is involved in a particular operation

Performance: Good. But more complicated code can allow the friend list of the user to be accessed concurrently.

Correctness: The data from the wall is correct. No other request’s output will be jumbled with the wall.

Name: "$id"_lock and "$id2"_lock
Why that name: We want the server to return the request responses in the right order to the client (hence "$id"_lock). We want to make sure no one writes in the user's wall as we pipe it (hence "$id2"_lock).

Why that level: For the client lock, it prevents other commands from running since all commands acquire the client id. For the other user lock, locking the wall only is more complicated code that takes longer to run.

Challenges

1.

In the error handling Cathal wanted to use the approach of having the server have it's own nok: error codes, with the client script reading what the exit code for request had been from the pipe.

You could read the exit code with $? In the server where it printed the right code but in client it would be a different code completely. Cathal tried to find any help on the web but couldn't anything that worked.

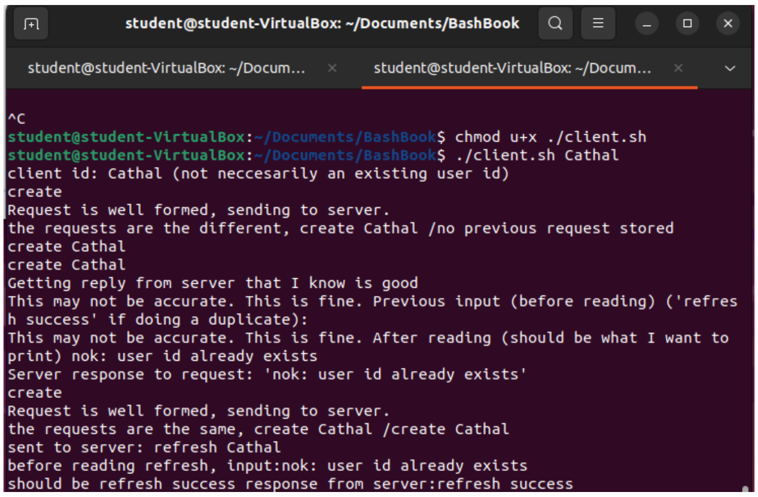
Cathal spent a lot of time trying to read the exit message from the pipe, or sending the code down the pipe separately. Looking back it was pretty silly as it was not required in the spec, and reading the error statements from the nok: echos worked anyhow.

2.

Noted issue: when the user enters the same command as before (and no other user has entered a command), the user updates the server.pipe (as its supposed to) but this update didn't change what was in the pipe (because the last command was the exact same).

To get around this we made a new function called refresh. This clears out the server.pipe and then passes the users command in again.
Where on the servers side it saw the pipe change from the previous command to an empty pipe and then a new command again to act on.

In the image to the right you can see how the server refreshes the pipe and then passes in create again.


student@student-VirtualBox: ~/Documents/BashBook
student@student-VirtualBox: ~/Docum...
student@student-VirtualBox: ~/Docum...

```
^C
student@student-VirtualBox:~/Documents/BashBook$ chmod u+x ./client.sh
student@student-VirtualBox:~/Documents/BashBook$ ./client.sh Cathal
client id: Cathal (not neccesarily an existing user id)
create
Request is well formed, sending to server.
the requests are the different, create Cathal /no previous request stored
create Cathal
create Cathal
Getting reply from server that I know is good
This may not be accurate. This is fine. Previous input (before reading) ('refres
h success' if doing a duplicate):
This may not be accurate. This is fine. After reading (should be what I want to
print) nok: user id already exists
Server response to request: 'nok: user id already exists'
create
Request is well formed, sending to server.
the requests are the same, create Cathal /create Cathal
sent to server: refresh Cathal
before reading refresh, input:nok: user id already exists
should be refresh success response from server:refresh success
```

Conclusion

We have made a working rudimentary version of Facebook with synchronisation and pipelining for each unique user.

With our program you can:
1. Create a user, with a friend list and a wall.
2. Add other users to your friend list. (Asymmetric – Adding a friend puts them on your friend list, they can now post on your wall).
3. Post a message to a user's wall with the name of the user who posted the message.
4. Display a user's wall.
5. Do any of the above operations by typing the request name in the client.sh terminal that you started with your id.