CAB403 Assignment Report

Group 007 - Semester 2, 2019

Name	**Student Number**
Lachlan Kuhr	n9767151
Andrew Mather	n9713671
Jeremy Sheather	n9734783

Statement of Completeness

All parts of the assigned task were completed.

Statement of Contribution

Name	**Contribution**
Lachlan Kuhr	33%
Andrew Mather	33%
Jeremy Sheather	33%

Table of Contents

Data Structures

In completing this assignment, some data structures were used. These were typically stored on the server. There were three main data structures required on the server:

1. Clients
2. Messages
3. Channels

The structs used for these are available in data.h.

Clients

The client struct, denoted **struct client** and aliased to **client_t,** contained four main variables:

1. The client ID.
2. The client's connection socket.
3. An array of "client channels", used to store a client's subscription status to channels.
4. An array of pointers to the last read message in each channel, so that the server could keep track of where each client was up to without duplicating messages.

Messages

The message struct, denoted **struct msg** and aliased to **msg_t**, containing three variables:

1. The message string.
2. The ID of the user who sent the message.
3. The time at which the message was sent, as available in time.h.

Channels

Channels are the way clients can interact with each other via messages. A client sends a message to a channel, which other clients may then receive. Each channel is implemented as a **linked list**, with a pointer to the head of the linked list stored in an array of size 256. Each index in the array refers to a different channel. The main struct for channels, denoted **struct msg_node** and aliased to **msgnode_t**, is a node in this linked list, and stores the variables:

1. A pointer to the message struct referred to by the node.
2. A pointer to the next node in the linked list.

The head, or the pointer at each index in the array, will always point to the last message sent in the channel. Meanwhile, the server tracks which message each client has read using the array in the client struct. When the client wants to read the next message, the server will iterate the pointer from the head to the last message read, and display the message before the last message is read. This will then be set as the last message read.

Task 2

Forking

Each time the server receives a new connection, it forks a new process to handle this connection. As such, each client connects to a different ‘version’ of the server, all of which are children to the main parent connection.

To implement this, the **fork** command was used. When the server accepts the new connection, **fork** is called and its output stored and checked. If the check returns something greater than 0, it is known that this process is the parent, and the parent will continue to wait to accept new connections from clients. If the return value is 0, then the child begins processing for the new client.

The data structures described above need to be accessible between processes. To do this, shared memory has been implemented using **shm_open** and **mmap**. One of the first processing steps the server performs is to set up shared memory. It predefines five shared memory objects:

1. Locations of message nodes, to store 256 msgnode_t pointers.
2. Message nodes, to store 1000 msgnode_t structs.
3. Messages, to store 1000 msg_t structs.
4. Counts of messages sent, to store 257 integers.
5. Locks, to store two shared reader-writer locks.

To do this, first **shm_open** is used to create shared memory objects, and then **ftruncate** ensures they are the correct size. Then, **mmap** maps these shared memory objects to variables. **mmap** returns a pointer to the start of the shared memory segments. Finally, these are initialised to be NULL or 0 depending on the structure in use.

These may then be used as normal array variables, with normal pointers associated. The variables within will be mapped across all processes, and any pointers returned will point to an address in the shared memory segment.

Critical Section Problem

To handle the critical section problem, pthread reader-writer locks (pthread_rwlock_t) were used. Two locks were made available – one to control access to structs in shared memory containing actual messages. The second was used to control the arrays in shared memory containing message counts.

The library in use contains three main usage functions, along with functions to initialise and destroy the locks where appropriate during setup and shutdown. Initialisation occurred immediately after shared memory segments were setup, as part of the server's startup procedures, while destroying them occurs as part of the SIGINT shutdown procedures.

The other three functions available are a read lock (**pthreads_rwlock_rdlock**), a write lock (**pthreads_rwlock_wrlock**) and an unlock (**pthreads_rwlock_unlock**). These are used around critical sections of code. When a lock is locked for writing, no other critical sections that use that lock for anything (reading or writing) may obtain the lock until it is unlocked, which occurs after the critical section is completed. Meanwhile, when a lock is locked for reading, other critical sections that obtain the lock for reading are permitted, but no writers are allowed to obtain the lock until all read locks have been unlocked.

Locks are used in numerous places throughout the code, with read locks more common than write locks for efficiency.

The message struct lock is used in the following places:

- When the client subscribes to a channel, they need to obtain a pointer to the last message sent to the channel. This uses a **read** lock.
- When the client wants to obtain the next message (without specifying the channel ID), the message structs are locked for **reading** while the server reads from the channels, and determines which message was sent first (by comparing the time values).
- The server locks the messages for **writing** to update the client's position in the channel linked list after the client has read a message.
- When the client wants to obtain the next message with a channel ID, **read** lock.
- When the client sends a message to a channel, the server obtains a **write** lock to create the new message and add it to the channel's linked list.

The message count lock is used in the following places:

- When the channels command is called, a **read** lock is obtained to read the message counts from the message count array.
- When the client sends a message, a **write** lock is obtained to increment the number of sent messages to each channel, and the total number of messages sent to the server.

Task 3 Threading

Threading

Threading was implemented on the client side of the application in order to allow commands to be entered while NEXT and LIVEFEED commands are running. A new command, STOP, was also implemented, with the purpose of stopping these functions if they are running.

A global **pthread_t** array was implemented to contain information about running threads.

Threading was implemented using the pthreads library. New threads are only created for functions that require them, which are NEXT and LIVEFEED. When the client starts and connects to the server, it will sit and wait for input from the user. When it receives this input, it extracts the command part of the input, and checks this against known functions.

In the event of the command LIVEFEED or NEXT, **pthread_create** is called and an array tracking which threads are active in the pthread_t array is updated, and if a channel ID was provided it is parsed to an integer.

When, **pthread_create** is called. If the function called is NEXT, **pthread_create** is passed the following variables:

1. The next available thread in the pthread_t array, indicated by first inactive index.
2. NULL as the attribute, as there are no attributes that need to be specified.
3. The function pointer to the function **nextThreadFunc,** which is responsible for sending the request to the server and receiving and printing the output.
4. Arguments required for **nextThreadFunc,** are the channel ID and the index of the thread created. These are sent as a void pointer struct, and cast back in the thread function. The index of the thread allows the thread to update its completion easily.

nextThreadFunc then proceeds to execute normally. It will send the command as NEXT or NEXT <channel ID> to the server (depending on the presence of a channel ID), before sending a request and receiving a message back from the server. At the end of its execution, it updates the thread as completeand calls **pthread_exit** to close the thread.

Calling LIVEFEED performs similar actions. When **pthread_create** is called, it is passed the same variables as NEXT described above, but with the difference of the function pointer being to **livefeedThreadFunc**.

The function of **livefeedThreadFunc** is similar to **nextThreadFunc**, with the exception that the send and receive commands are looped. If an error occurs, the loop breaks and the thread closes itself appropriately. When STOP, BYE, or CTRL+C is used, the system cleanly closes any open threads. All of these will call the function **closeThreads** as part of their execution. **closeThreads** loops through the thread array, calling **pthread_cancel** on those that are active (according to a second status array).

How to run the program

Compilation

A Makefile is included at the root of the project to make compilation straightforward. To compile both the client and server sides, simply call **make** at the command line from the root of the project.

If only one of client or server needs to be compiled, call **make client** or **make server**. To remove the client or server, call **make clean**.

Running

To run the program, change into the *build* directory from the root of the project. The server needs run before any clients can be run. Call **./server <port>** on to run the server, where **port** is the port number that clients will use to connect (port defaults to 12345 if not chosen).

To run client programs, run **./client <ip address> <port>. Ip address** is the IP of the machine the server is running on (if this is the same as the client's machine, IP will be 127.0.0.1), while **port** is the port specific when the server was run.