CAB403 Assignment Report

Eugene Martens

N10318313

Tasks Completed

Able to implement all three 3 tasks

Notable bugs

- Server Exit is caught by reader_thread although it is unable to cancel getchar() loop on clients.
- Although not able to replicate with certainty at this point, there are times where data read in livefeed_thread is incremented but not posted to clients.

Group

Worked Alone

Data structures

Shared Memory Block: shared.c / shared.h
Area of memory that all processes/clients/threads write to/from created at the start of a new server initialization and cleared during server close.

Shared memory consists of a memory struct with 255 channels, integer num_clients (tracks the number of clients) as well as integer status flag (currently unused).

Channel is a struct that contains a mutex lock (Threads and Processes used to notify Read/Write status) and an array of 255 posts and an integer post_index that functions as a pointer index. Post is a struct that contains a read flag (currently unused) and a 1024 sized array of bytes where messages are stored.

Subscription linked-list: subscrption.c / subscription.h
Subscription linked list is used to keep track of client data, each node within the linked list is a struct *subbed channel* that holds a channel ID integer (channel number), read index integer (tracks user read position) and a next subbed channel pointer.

The list is initialized at the start of each forked process, where a client struct is allocated in memory. Client struct contains a subbed channel node pointer (head) and client ID integer, with the head used to point to the start of the list.

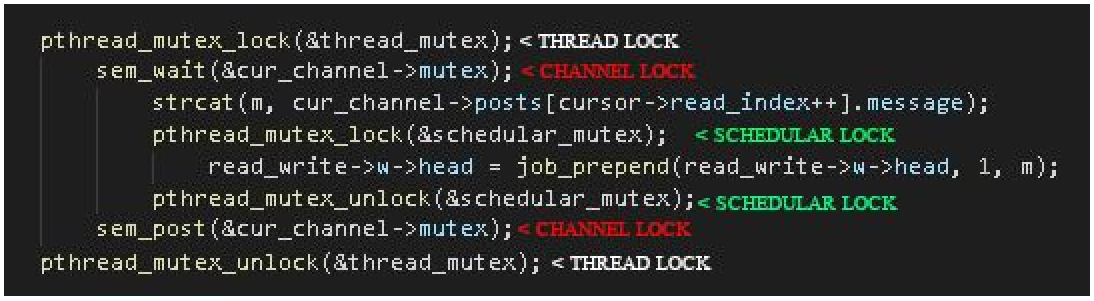
Worker linked-list: worker.c / worker.h
Worker linked list is used for output queuing from the server, each node within the linked list is a struct *job* that holds a job ID integer (currently unused), char* data (holds relevant data) and a next *job* pointer.

The list is initialized at the start of each forked process, where a *worker* struct is allocated in memory. Worker struct contains a *job* node pointer head used to point to the start of the list.

Forking

Upen receiving a connection from a client, server will invoke a fork process, that includes a server chat handler that will communicate with the client and accept commands and post responses via poster thread. Poster thread reads from the worker linked list and posts any commands or outputs currently in queue and pops off the node when done.

Critical Section Resolution, Task 2 and Task 3 : pthreads.h and semaphores.h



```
pthread_mutex_lock(&thread_mutex); < THREAD LOCK
    sem_wait(&cur_channel->mutex); < CHANNEL LOCK
        strcat(m, cur_channel->posts[cursor->read_index++].message);
        pthread_mutex_lock(&schedular_mutex);  < SCHEDULAR LOCK
            read_write->w->head = job_prepend(read_write->w->head, 1, m);
        pthread_mutex_unlock(&schedular_mutex);< SCHEDULAR LOCK
    sem_post(&cur_channel->mutex);< CHANNEL LOCK
pthread_mutex_unlock(&thread_mutex); < THREAD LOCK
```

Scheduler mutual exclusion: pthreads.h
All processes and threads that send output will do so by appending to the *worker* linked-list where the poster thread will read the data and remove it from the queue. To ensure that the correct node is added and removed in order, a pthread_mutex lock before each append to list and when removing from the front of the list.

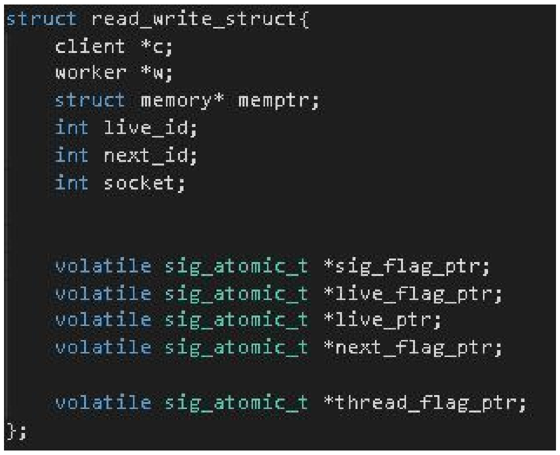
Thread mutual exclusion: pthreads.h
Specifically for threads adding/removing to the worker linked-list queue, to ensure correct order among threads a second thread-orientated pthread_mutex lock is used embedded without the scheduler lock. The lock guarantees a add/remove order coupled with the overall lock for the entire program.

Channel mutual exclusion: semaphores.h
Used by all processes (including forked) and threads as well as general processes such as SEND where a semaphore mutex lock within the channel struct is used to ensure a read write order for everyone using the shared memory.

Threading : pthreads.h

Server Threads

All server threads are created during a fork when a client connects to the server, and joined again when the client or server closes. All Threads use a read_write struct that is passed as an argument that contains various integer flags and pointers to client and worker lists. All threads sit in loops to be kept active in the background using the volatile integers sig_flag and thread_flag, where thread_flag will be set to 0 when a client closes the session and killing all threads currently running for that session.



```
struct read_write_struct{
    client *c;
    worker *w;
    struct memory* memptr;
    int live_id;
    int next_id;
    int socket;


    volatile sig_atomic_t *sig_flag_ptr;
    volatile sig_atomic_t *live_flag_ptr;
    volatile sig_atomic_t *live_ptr;
    volatile sig_atomic_t *next_flag_ptr;

    volatile sig_atomic_t *thread_flag_ptr;
};
```

Live Thread

- Using volatile integers live_flag and live, accessed through the read_write ptr's is set to 1 on receiving LIVEFEED and only if appropriate (wrong values or not being subscribed to the channel). The live thread will continuously check for new messages, either from all channels (if live_id == -1) or from specified channel, only closing when STOP, or CTRL-C flag is received from the client or UNSUB causes the user to have no channels or from the specified live feeds channel.

Next Thread

- Using volatile integer next_flag, the thread will either look through all channels (if next_id = = -1) or from specified channel. Live and next threads use separate channel id flags in order to be able to access one channel while still live feeding another

Poster Thread

- All output is handled here, always running until thread_flag or sig_flag are set to 0. The thread will either post responses to a client, or a stream of "/0" terminators in order to keep the stream open, popping off nodes from the worker link list upon each iteration.

Client Threads

Works similar to server, initialized when connected to the server using thread_flag and sig_flag to keep a threaded while loop open in the background.

Reader Thread

- Reads all input responses from the server and prints them to the client terminal. If input from the server are not "\0" termination characters, the response is printed to the screen. There are also response flags set up such as server connection checking where it will set the read_buffer to a SIG and close the client chat.

Instructions

A make file is included in the project file that you can call within the terminal via “make”
Both server and client as well as any dependencies are compiled and ready to use.