Babble

ABDUL RAHMAN Zulhusni FERREIRA Christopher

November 2016

1 Introduction

This assignment's objective is to create a multi-threaded application from a given sequential program. In this case a client-server application called Babble which allows clients to post comments, follow other clients and see what others have posted. Learning to solve concurrency issues related to multi-threaded programming is the key aspect in finding a functional program.

2 Stage 0

Digging the code:

1. The part of code which opens the socket for listening is the server_connection_init in the main function of the babble_server c file.
2. The part starting from the server_connection_accept until the loop on client messages handles new connections and login of the clients.
3. When a message is received, the server parses the command, process it and then answers the command. It then loops again waiting for new messages.
4. The LOGIN command is managed differently as we need to register the socket associated with the new client before processing the command.
5. The registration table is used to store the data associated to each clients.
6. The keys are used to search for the corresponding client in the registration table using the registration_lookup function.
7. To each command is associated a field *answer* which is basically a list of message lines (strings) to send back to the user. The answer is then done in two steps. *process_command* fills the *answer* field depending on the command and then *answer_command* is a generic function which send those strings over the network.
8. The FOLLOW request first requires to verify that the client to be followed is logged in and not already followed. Then the number of followed count of the requesting client is incremented.
9. To ensure that only new messages are shown in the TIMELINE, we first read the last_timeline date from the client_data. Then, we recover the publication from each followed client that have a date superior to the last_timeline.
10. The high-level function provided is write_to_client which returns 0 on success and -1 on failure. We also have network_recv which return the size of the received data and network_send which returns the size of the sent data.

3 Stage 1

3.1 One connection - one thread

In this first part an implementation of two different kinds of threads are demanded. The first kind called *Communication Thread* is responsible for receiving messages from the client and parsing this messages as executable commands and the other kind called *Executor Thread* executes the aforementioned commands. A *Communication Thread* will be created for each client that logs in while only one *Executor Thread* will be responsible for managing all of the commands.

3.2 Command buffer

As this scenario involves the *Communication Threads* creating multiple commands and the *Executor Thread* taking the commands, this problems resembles the producer-consumer problem with *Communication Threads* being the producers and *Executor Thread* being the consumer.

With this information we created a buffer structure which will store the commands to be passed to the *Executor Thread*. As this buffer is considered a shared memory we implemented a combination of lock and semaphore to ensure that no conflicts will appear when reading or writing to the buffer.

```
1  void buffer_add_item(buffer_t* buf, void* item){
2    sem_wait(&buf->not_full);
3    pthread_mutex_lock(&buf->mutex);
4
5      /* Write to buffer */
6
7    pthread_mutex_unlock(&buf->mutex);
8    sem_post(&buf->not_empty);
9  }
10
11 void* buffer_get_item(buffer_t* buf){
12   sem_wait(&buf->not_empty);
13   pthread_mutex_lock(&buf->mutex);
14
15   /* Read from buffer */
16
17   pthread_mutex_unlock(&buf->mutex);
18   sem_post(&buf->not_full);
19
20   return item;
21 }
```

Figure 1: Implementation of the buffer queue

3.3 Reader-Writer Lock for Registration Table

Another shared memory issue to consider for this application is the registration table which can be read or written by multiple threads depending on the command given to the threads. To be more precise, the registration table is modified when the user logs in (inside one of the *Communication Threads*), read by all commands (inside the *Executor Thread*) and modified again when the user logs out. Because modification of the registration table only occur at the start and at the end of the user session, we decided to use the reader-writer scheme to avoid unnecessary contentions for the commands.

The logout also poses another problem because it occurs in the *Communication Thread* and free the data relative to the user. This can happen while the *Executor Thread* has yet to handle one command of this user which may have to access these freed data. To avoid this problem we decided to implement the logout as a command which has to be executed by the *Executor Thread*. This solve the problem as long as there is only one executor which handle the commands in the order they arrive.

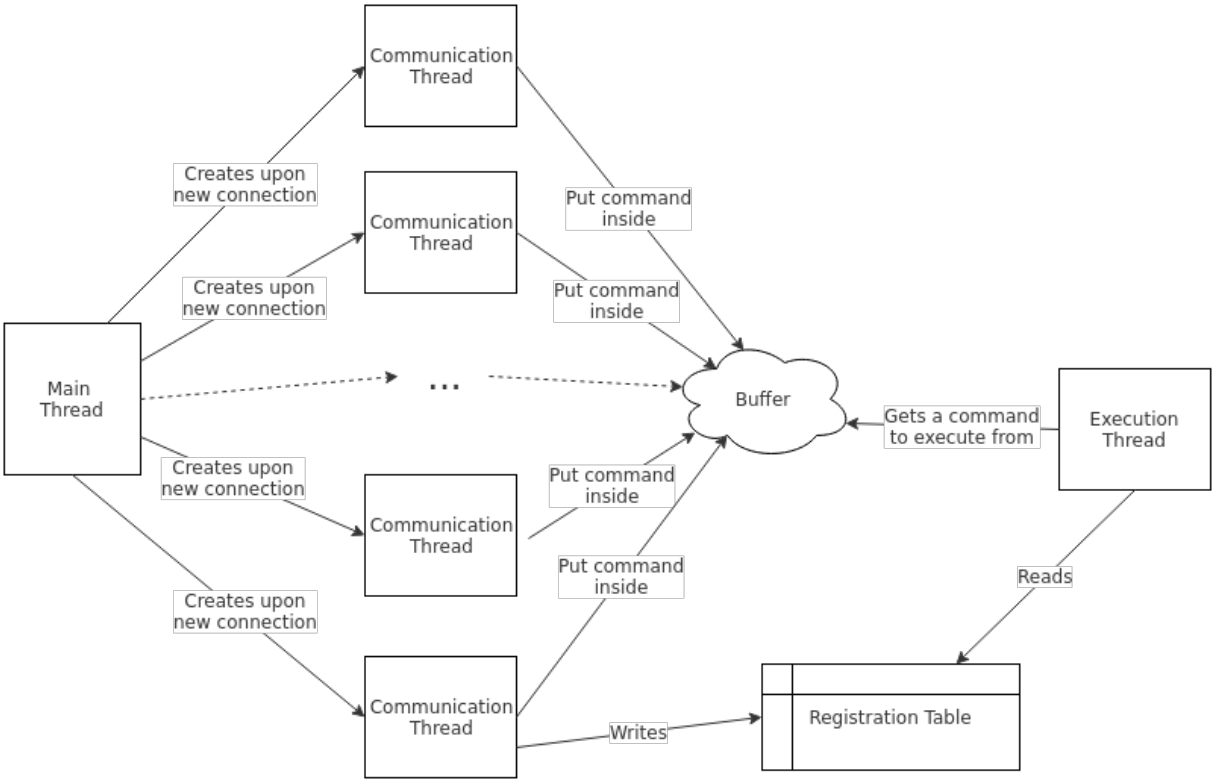

Figure 2: Threads-Producer-Consumer

4 Stage 2

4.1 Avoiding DoS attacks

With the current program, the application is very susceptible to DoS attacks as the the server allows the creation of limitless clients. This can result to an attacker filling the server with connection requests and the server would spend all of its resources on handling these requests rather than processing messages from clients. We modify our server to create a fix number of *Communication Threads* in the beginning which will avoid the problem stated before. There are two possible solutions that we find to implement this modification.

The first solution is similar to the first stage where we will create a buffer to store all the connections accepted by the main thread and the *Communication Threads* will read the connections and handles further requests from the clients. The second solution is each *Communication Threads* have their own accept function. This approach is correct as the accept function is *thread-safe* and it simplifies the implementation as it does not involve more concurrency problems to the program.

In the beginning we went with the second solution but later changed to the first solution as it will be more coherent with future modification which will be discussed on Stage 4.

5 Stage 3

5.1 Multiple Executor Threads

To increase the efficiency of the server, we now want to create multiple *Executor Threads* which can handle the commands in parallel. This modification introduces new concurrency issues related to the commands when dealing with client data; the publication of a client, the follower count and the followed count.

We implemented read-write locks on the publication sets and the followed count as both of these data are accessible by other clients which means there can be multiple readers at a time. For the follower

count we used a simple atomic fetch_and_add.

One more thing to consider is the problem when a client uses the *RDV* command. When using multiple *Executor Threads* the *RDV* command have to know the state of the other commands before it from the same client; whether they are finished or not. To solve this we add two counters on the client data; a received counter which is incremented each time a command from this client is created and a finished counter which is incremented after a command from this client is handled. When the *RDV* command is executed, it waits for a signal from a condition variable which is sent when both the values for received counter and finished counter is equal.

```
1 void ensure_order(client_data_t *client, command_t *cmd) {
2     pthread_mutex_lock(&client->commands_finished_mutex);
3     while (client->commands_finished < cmd->order) {
4         pthread_cond_wait(&client->commands_finished_cond, &client->
      commands_finished_mutex);
5     }
6     pthread_mutex_unlock(&client->commands_finished_mutex);
7 }
```

Figure 3: Implementation of Order Enforcement

This solution is also used with the *UNREGISTER* command which requires that all previous commands related to this client are finished before removing the client from the registration table.

6 Stage 4

We want a solution which would allow communication threads to work as executor threads when no new connection has to be handled. This means the communication threads has to be to wait if both the new connection buffer and the command buffer are empty. Our previous solution for the bounded-buffers used semaphore which proved to be tricky to use for this new problem. We decided instead to switch to an implementation using condition variables. The idea is that communication threads will wait on one condition variable and executor threads will wait on another one. Then when a new command is inserted inside the command buffers both condition variables are signaled.

```c
void buffer_add_command(void* item){
  sem_wait(&cmd_buf->not_full);
  pthread_mutex_lock(&mutex);

    /* Write to command buffer */

  pthread_cond_signal(&cmd_buf->not_empty);
  pthread_cond_signal(&client_buf->not_empty);
  pthread_mutex_unlock(&mutex);
}

void* buffer_get_client_or_command() {
  pthread_mutex_lock(&mutex);
  while (client_buffer.count == 0 && cmd_buffer.count == 0) {
    pthread_cond_wait(&client_buffer.not_empty, &mutex);
  }

  void* item;
  if (client_buffer.count > 0) {
      item = /* Taken from client buffer */
  }
  else {
      item = /* Taken from command buffer */
  }

  pthread_mutex_unlock(&mutex);
  sem_post(&buf->not_full);

  return item;
}
```

Figure 4: Clients and commands buffers synchronization for Stage 4

Note that for this solution to work, it is necessary to use the same mutex for both buffers which is not ideal. Note also that this solution does not ensure that command will be given to available executor threads in priority over communication threads.

We decided on a solution which keeps a fixed number of threads dedicated to the execution of commands because without this constraint, some situations might lead to deadlock. For example if all N threads are hybrid threads, and N clients connect, it is possible that all hybrid threads becomes communication threads and starts filling the commands buffer. If this buffer gets full, all threads will be put to wait, waiting for an unexisting thread to empty the buffer.

7 Conclusion

This Operating System Project teaches us to think and code with multiple threads which introduces a whole new class of problems related to concurrency. We learnt to identify the critical sections and leverage the different synchronization mechanisms provided by the operating system to ensure a coherent working environment between these multiple threads.