Operating System Lab 7 MoSIG
Babble- A Multi Threaded Server

Shakeel Ahmad Sheikh, Vaynee Sungeelee

Stages	FollowTest (with and without Streaming)	StressTest (with and without Streaming)
Stage 1	Passed	Passed
Stage 2	Passed	Passed
Stage 3	Passed	Passed
Stage 4	Not Passed	Test Not Passed

Stage 0 First Contact with Babble

1. ***Which part of the code opens the socket where the server is going to listen for new connections?***
server_connection_init(int port) in the file "**babble_server_implem.c**", is the part of the code which will open the socket and server will listen for new connections.

2. ***Which part of the code manages new connections on the server side?***
server_connection_accept(int sock) in "**babble_server_implem.c**" is code piece which returns a new file descriptor for a new connection.

3. ***What are the major steps that are run on the server when it receives a message from the client?***

- Parsing the message received from the client
- Creating the command after parsing
- Processing the command
- Answering back to the client

4. ***Why are LOGIN messages managed differently from other commands?***
When a client tries to LOGIN, a new key is generated in the form of hash value on the client name and is being updated in the registration table. A socket is closed if LOGIN fails.

5. ***What is the purpose of the "registration_table"?***
The keys corresponding to a client is stored in registration_table. This registration table is used to store keys which are looked up upon FOLLOW, PUBLISH, TIMELINE, etc operations.

6. ***How are keys used on the server?***
Keys are generated by hashing the client id which is stored in the registration_table. In order to see whether the client is valid or not, registration_table is being searched for its hash values (keys).

7. ***How answering to requests is implemented on the server?***

When a client makes a request to the server, this request is being parsed and processed at the server side and the answer is being returned accordingly.

8. ***What happens on the execution of a FOLLOW request?***
On execution of a FOLLOW request, the server looks for the key corresponding to the id that the client wanted to follow. If the key is not found, then the server will generate an error, else it will add the client to the list of followed and increment the follow count of the user, who is being followed.

Stage 1

Problems Faced:

Producer-Consumer And Reader Writer:

Because we have a multithreaded application, multiple communicator threads and the executor thread can access the shared command buffer simultaneously and create a **producer consumer** problem (Fig 1 & 2). The executor thread takes the commands from the buffer, processes them and gives feedback/ answer to the clients. In our case we have communicator threads as producers and executor threads as consumers. The second concurrency problem is the **reader-writer** problem. The registration table in the registration.c file can be accessed concurrently by multiple communicator/executor threads for each insert/lookup/deletion.

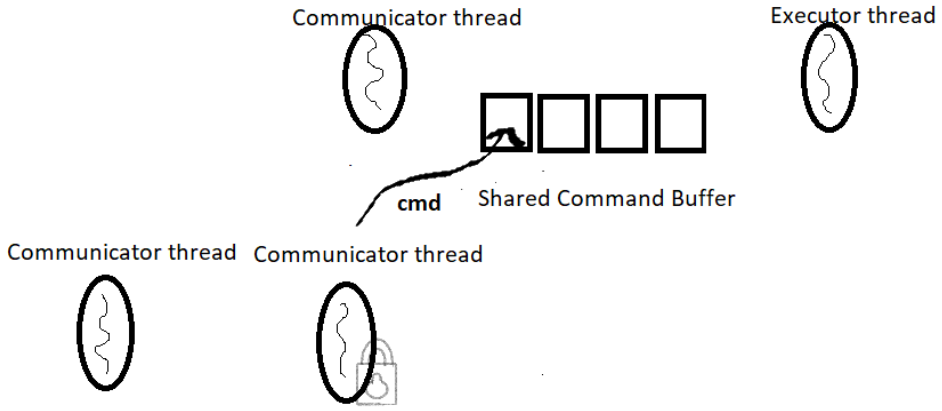

Figure 1 Communicator thread adding command

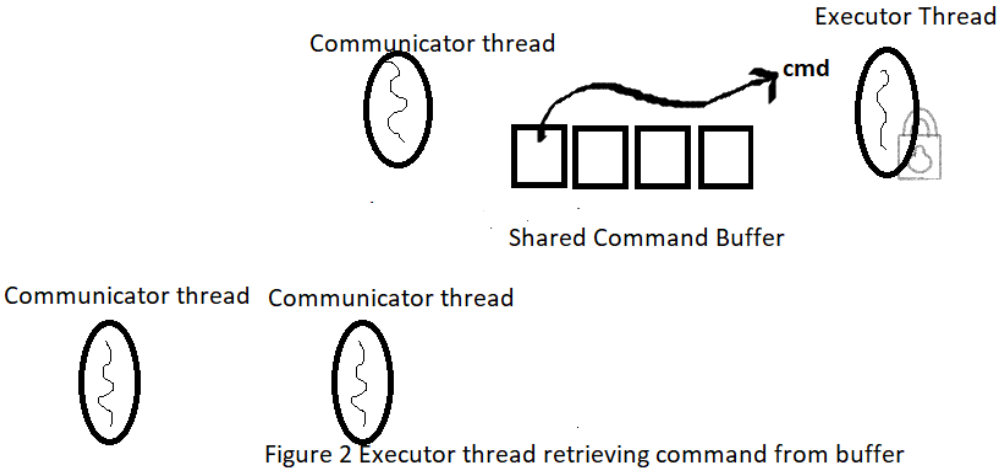

Figure 2 Executor thread retrieving command from buffer

Solution

We created multiple communication threads, one for each client. They will forward the commands from the clients to a command buffer (of size BABBLE_PRODCONS_SIZE). We have 1 executor thread whose job is to manage the processing of commands and sending answers. We created two functions separately **communicator_thread_func** (multiple clients can use it) for producer and **executor_thread_func** for consumer. So in order to solve the producer consumer problem, we used 3 semaphores with the initial values of *1* for the lock (mutual exclusion), *N* (size of command buffer) for empty and *0* for full. The locking semaphore ensures mutual exclusion for the shared variables in the critical section, the **full** semaphore blocks the producer when the buffer is full while the **empty** semaphore blocks the consumer when the buffer is empty. Semaphores were ideal in this case because the value of the semaphore ***(N)*** acts as a counter by itself and eliminates the need for another variable to keep track of the current buffer size.

Read-Write Problem:

In order to solve this problem, we used a readlock (pthread_rwlock_rdlock) in the function **registration_lookup** to prevent a writer () from accessing the shared variables like the registration table and **nb_registered_clients** while readers are reading these variables. We used a writelock (pthread_rwlock_wrlock) in the code for **registration_insert** since it should only be accessed by one thread a time to get the correct value for the variable **nb_registered_clients** after it is incremented .In the same way, the writelock is placed in the **registration_remove** code because only 1 communicator thread should unregister the client at a time while decrementing the shared variable **nb_registered_clients.**

Stage 2

Problem Faced:

In addition to the producer- consumer and read write problem faced in stage 1, we faced another producer consumer problem where multiple answer threads are accessing the answer_buffer concurrently.

Solution:

In order to solve this problem, we used 3 semaphores as we did in stage 1. We created a shared answer buffer of size **ANS_BUFFER_SIZE** set to 10. The executor thread processes the commands retrieved from the command buffer and stores the answers in the answer buffer. The function **answer_thread_func** looks for answers generated by the executor thread from the answer buffer and sends them to the clients. Since multiple answer threads can access the answer buffer concurrently, the critical section is the part of the code that accesses the answer buffer. We do a sem_wait so that the executor thread checks and acquires the lock and puts the answer returned from process_command into the answer buffer. Then we do a sem_post to wake up consumers (answer threads). Now the answer threads can send the answers to the client by accessing the critical section one at a time and retrieve an answer from the answer buffer.

Stage 3

Problem Faced:

In stage 3, we have identified the concurrency issues which arise when we introduce more than 1 executor thread. In particular, we have race conditions on the variables which are shared in the different commands PUBLISH , FOLLOW , TIMELINE, FOLLOW_COUNT , RDV. We have a reader-writer problem.

Solution:

The solution we came up with is that we use read-write locks on shared variables in the following functions used to process a command by the executor threads concurrently: **run_publish_command**, **run_follow_command**,**run_fcount_command**,**run_rdv_command**, run_timeline_command, in the **babble_server_implem.c** file.

Stage 4

In the stage 4, we used another shared buffer (publish_buffer shared between communicator and executor threads) to store PUBLISH commands only. We want to process publish commands before any other command. The communicator thread puts the publish commands in this buffer and executor thread consumes the command from the buffer. The idea we had is illustrated below:

```
//In  communicator thread

communicator_thread() {
If (command == PUBLISH)  {
Publish_buffer = command
  count ++
}
else {
Command_buffer = command
}
//In  executor thread
executor_thread() {
If ( count is 0)
        Sleep:

elseif( count>0)
        Take from publish buffer
else
Take from normal buffer
}
```

References:

1. Lecture 11, 12 on Thread Synchronization by Thomas Ropar (M1 MoSIG Course on operating system)
2. http://pubs.opengroup.org/onlinepubs/009604499/functions/pthread_rwlock_rdlock.html
3. https://cs.stackexchange.com/questions/83286/21-mutual-exclusion-with-semaphores?noredirect=1&lq=1