Ticketing Server Implementation

2020310830 Sunghwan Cho

December 18, 2024

1 Introduction

In this project, we implement a ticketing server that handles up to 1024 clients concurrently. The server manages seat reservations for a set of 256 seats, ensuring that only logged-in users can reserve, check, or cancel their reservations. Each client interacts with the server through a query structure that defines various actions such as login, seat reservation, checking reservations, canceling reservations, and logging out. This project aims to enhance understanding of multi-threaded server design, resource synchronization, and inter-process communication in a Linux environment.

2 Implementation

The server follows a client-server model where the server listens for incoming client connections and handles multiple client queries concurrently. The server supports five main actions:

1. Log in: Validates user login, registers new users, and checks for concurrent logins.
2. Reserve: Allows a user to reserve a seat if they are logged in and the seat is available.
3. Check reservation: Returns the reserved seat of the logged-in user.
4. Cancel reservation: Cancels the current reservation of the logged-in user.
5. Log out: Logs the user out and frees up their login resources.

The server ensures synchronization using mutexes to prevent race conditions, especially in managing seat reservations and user logins. Each seat is protected by a dedicated mutex, and a global mutex is used for managing user login states.

The termination condition is met when a client sends a termination query ”0 0 0”, which causes the server to shut down and return the seat reservation status to the client.

Additionally, pthread_detach is used when creating threads, ensuring each thread automatically releases its resources upon termination.

3 Code Explanation

The server code utilizes several important libraries and constructs:

- `<stdio.h>` and `<stdlib.h>`: Standard libraries for input/output and memory management.
- `<string.h>`: For string operations, primarily used for network communication.
- `<unistd.h>`, `<pthread.h>`, and `<signal.h>`: Used for thread management, signal handling, and low-level system operations.
- `<netinet/in.h>` and `<arpa/inet.h>`: For socket programming and handling internet addresses.

The key structures and functions in the implementation are:

- `pthread_mutex_t`: Mutexes are used to synchronize access to critical resources like the seat array and user login states.
- `query`: A structure that represents the query sent from the client, containing the user ID, action, and data for the action.
- `initialize_server()`: Initializes all arrays and mutexes used in the server.
- `client_handler()`: This function runs in a separate thread for each client and processes their queries based on the action ID.
- `signal_handler()`: Handles the `SIGINT` signal (Ctrl+C) to properly clean up server resources and close sockets.

4 Experiment

To test the concurrency and thread safety of the ticketing server, we created a client program capable of simulating 1024 clients connecting to the server. Each client attempts to log in, reserve a seat, and log out. The test was conducted using two configurations for client thread creation and execution to observe the behavior of the reservation system. Specific code is here.

4.1 Experiment 1: Reverse Order Client Creation

In the first test, the client threads were created in reverse order, starting from the highest user ID (1023) down to the lowest (0). Once all threads were created, they were detached, allowing each thread to execute independently. The client was terminated by sending the termination query `0 0 0`, and the reserved seat array was printed. The output was as follows:

```
768 769 770 771 772 773 774
...
1019 1020 1021 1022 1023
```

This result demonstrates that the server processed client threads in a thread-safe manner, ensuring that each client was able to reserve a seat according to its request. Even with concurrent execution, no seats were overbooked or skipped, verifying the correctness of the synchronization mechanisms.

4.2 Experiment 2: Sequential Client Creation

In the second test, the client threads were created in sequential order, starting from the lowest user ID (0) up to the highest (1023). Each thread was joined to ensure all threads completed execution before the program terminated. The reserved seat array was as follows:

```
0 1 2 3 4 5 6 7 8 9
...
252 253 254 255
```

This result confirms that the server handled sequential client execution correctly, with each client reserving a seat corresponding to their user ID modulo 256. The thread-safe design ensured that no race conditions occurred even with a high volume of concurrent threads.

4.3 Results

The tests demonstrate that the ticketing server is thread-safe and maintains correct concurrency behavior. The use of mutexes for synchronization in seat reservation and user login management successfully prevents race conditions. The server consistently handles 1024 concurrent clients and ensures that all reservations are processed without conflicts, verifying the integrity of the implemented algorithm.

5 Conclusion

The ticketing server successfully manages multiple users, ensuring proper synchronization for seat reservations, cancellations, and logins. It effectively handles concurrent client requests through the use of threads and mutexes. This project highlights the importance of resource management and synchronization in multi-threaded server applications.