| Doc Id | Language | Version | Author | Date | Page No. |
|---|---|---|---|---|---|
| ReqDOC/1 | English | 1.0 | Akshatha D | June 4 | 1 |

# LINUX MULTI-THREADED CLIENT-SERVER USING SHARED MEMORY

## Table Contents

## 1. Abstract

This project implements a client-server system where the server generates random numbers and shares them with connected clients using shared memory. The server supports multiple clients simultaneously and uses semaphores to synchronize access to shared memory, ensuring data integrity.

## 2. Introduction

The primary goal of this project is to create a networked system where a server generates random numbers and shares them with multiple clients. The server stores the random numbers in a shared memory segment, and clients read these numbers. Synchronization is handled using semaphores to avoid race conditions. This project showcases basic concepts of socket programming, inter-process communication (IPC) using shared memory, and synchronization mechanisms in a multi-threaded environment.

## Project Scope

The scope includes developing:

- A server application that:
  - Listens for incoming client connections.
  - Handles multiple client connections concurrently using threads.
  - Uses shared memory to store messages from clients.
  - Uses semaphores to synchronize access to shared memory.
- A client application that:
  - Connects to the server.
  - Sends messages to the server and receives responses.
  - Allows the user to quit the connection gracefully.

## 3. Requirements

### Functional Requirements

1. Server Requirements:
   - Create and bind a socket to listen for incoming client connections.
   - Generate random numbers at regular intervals.
   - Store generated random numbers in a shared memory segment.
   - Use a semaphore to synchronize access to shared memory.
   - Handle multiple client connections.
2. Client Requirements:
   - Connect to the server using a socket.
   - Access the shared memory segment created by the server.
   - Use a semaphore to synchronize read access to shared memory.
   - Display the random number read from shared memory.

### Non-Functional Requirements

- **Performance:** Efficiently handle multiple client connections and synchronize access to shared memory.
- **Reliability:** Ensure robust communication between clients and server, handle disconnections gracefully.
- **Usability:** Provide a simple interface for clients to send messages and receive responses.
- **Scalability:** Support up to a specified maximum number of concurrent clients (e.g., 50).
- **Security:** Ensure safe and synchronized access to shared memory to prevent data corruption.

## 4. System Design

### Server Application

1. Socket Creation and Binding:
    - Create a socket and bind it to a specified port to listen for client connections.
2. Shared Memory and Semaphore:
    - Create and configure shared memory to store the random number.
    - Initialize a semaphore for synchronizing access to shared memory.
3. Thread Management:
    - One thread generates random numbers and stores them in shared memory.
    - Another thread handles incoming client connections, managing them in a loop and creating new threads for additional clients if needed.

### Client Application

1. Socket Connection:
    - Create a socket and connect to the server.
2. Shared Memory and Semaphore:
    - Access the shared memory segment created by the server.
    - Use the semaphore for synchronized read access.
3. Display Data:
    - Continuously read and display the random number from shared memory.

## 5. Code comments and Explanations

**Server Code:**

```
// Server


#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>

#include <string.h>

#include <fcntl.h>

#include <sys/mman.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <semaphore.h>


#define PORT 9090

#define MAX_CLIENTS 1

#define SHM_NAME "/shm_random_numbers"

#define SHM_SIZE sizeof(int)


sem_t *semaphore;

int server_fd;

int client_count = 0;
```

```
// Function to generate random numbers and store them in shared memory

void *generate_random_numbers(void *arg) {

    int shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);

    if (shm_fd == -1) {

        perror("shm_open");

        exit(EXIT_FAILURE);

    }

    ftruncate(shm_fd, SHM_SIZE);

    int *shared_memory = mmap(0, SHM_SIZE, PROT_WRITE, MAP_SHARED, shm_fd,
0);

    if (shared_memory == MAP_FAILED) {

        perror("mmap");

        exit(EXIT_FAILURE);

    }



    printf("Shared memory ID: %d\n", shm_fd);



    while (1) {

        int random_number = rand() % 90 + 10;

        sem_wait(semaphore);  // Wait for semaphore to ensure exclusive access

        *shared_memory = random_number;  // Write random number to shared memory

        sem_post(semaphore);  // Release semaphore

        sleep(1);  // Generate a new number every second

    }

    return NULL;

}
```

```
// Function to handle multiple clients
void *handle_clients(void *arg) {
    struct sockaddr_in address;
    int addrlen = sizeof(address);
    int clients[MAX_CLIENTS];
    int num_clients = 0;


    // Continuously accept and handle clients
    while (1) {
        if (num_clients < MAX_CLIENTS) {
            // Accept new client connections
            int new_socket = accept(server_fd, (struct sockaddr *)&address,
(socklen_t*)&addrlen);
            if (new_socket < 0) {
                perror("accept");
                exit(EXIT_FAILURE);
            }
            // Store client socket in the array
            clients[num_clients++] = new_socket;
            client_count++;
            printf("\nAdding client on %d\n", new_socket);
        } else {
            // If maximum clients reached, create a new thread to handle additional clients
            pthread_t tid_client;
            pthread_create(&tid_client, NULL, handle_clients, NULL);
            num_clients = 0; // Reset the count for the new thread

            printf("\nCreating new thread group\n");

        }



        for (int i = 0; i < num_clients; i++) {

            // Handle client connection

            sleep(1); // Just to simulate some work with the client

        }

    }

    return NULL;

}



int main() {

    struct sockaddr_in address;

    int opt = 1;



    // Create socket file descriptor

    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {

        perror("socket failed");

        exit(EXIT_FAILURE);

    }



    // Attach socket to the port

    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt,
sizeof(opt))) {

        perror("setsockopt");
    exit(EXIT_FAILURE);

}

address.sin_family = AF_INET;

address.sin_addr.s_addr = INADDR_ANY;

address.sin_port = htons(PORT);


// Bind the socket

if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0) {

    perror("bind failed");

    exit(EXIT_FAILURE);

}


// Listen for incoming connections

if (listen(server_fd, 3) < 0) {

    perror("listen");

    exit(EXIT_FAILURE);

}


printf("\nServer is listening to port: %d\n", PORT);


// Initialize semaphore

semaphore = sem_open("/semaphore", O_CREAT, 0644, 1);

if (semaphore == SEM_FAILED) {

    perror("sem_open");

    exit(EXIT_FAILURE);
```

```
  }



    // Create threads for generating random numbers and accepting clients

    pthread_t tid_random, tid_client;

    pthread_create(&tid_random, NULL, generate_random_numbers, NULL);

    pthread_create(&tid_client, NULL, handle_clients, NULL);



    pthread_join(tid_random, NULL);

    pthread_join(tid_client, NULL);



    // Cleanup

    sem_close(semaphore);

    sem_unlink("/semaphore");

    shm_unlink(SHM_NAME);



    return 0;

}
```

**CLIENT CODE:**

```
// Client

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <semaphore.h>

#define PORT 9090
#define SHM_NAME "/shm_random_numbers"
#define SHM_SIZE sizeof(int)
```

```
int main() {
    struct sockaddr_in address;
    int sock = 0;
    struct sockaddr_in serv_addr;

    // Create socket
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("\n Socket creation error \n");
        return -1;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Convert IPv4 and IPv6 addresses from text to binary form
    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
        printf("\nInvalid address/ Address not supported \n");
        return -1;
    }

    // Connect to server
    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {
        printf("\nConnection Failed \n");
        return -1;
    }

    // Open shared memory
    int shm_fd = shm_open(SHM_NAME, O_RDONLY, 0666);
    if (shm_fd == -1) {
        perror("shm_open");
        return -1;
    }

    // Map shared memory
    int *shared_memory = mmap(0, SHM_SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
    if (shared_memory == MAP_FAILED) {
        perror("mmap");
        return -1;
    }

    // Open semaphore
    sem_t *semaphore = sem_open("/semaphore", 0);
    if (semaphore == SEM_FAILED) {
        perror("sem_open");
        return -1;
    }

    while (1) {
        sem_wait(semaphore);  // Wait for semaphore to ensure exclusive access
        int random_number = *shared_memory;  // Read random number from shared memory
sem_post(semaphore);  // Release semaphore
        printf("Received Data: %d\r", random_number);  // Display the number
        sleep(1);  // Read the next number after one second
    }

    // Cleanup
    sem_close(semaphore);
munmap(shared_memory, SHM_SIZE);
    close(shm_fd);
    close(sock);

    return 0;
}
```

## test.sh :

```
#!/bin/bash

# Define the number of clients to run
num_clients=15

# Loop to run clients
for ((i=1; i<=$num_clients; i++))
do
    # Command to run your client, replace this with your actual client command
    # Example: ./client_program &
    ./client&

    # Optional: Add sleep to stagger the start of each client
    # sleep 1
done

# Optional: Wait for all clients to finish
wait
```

## Server Code

1.Socket Creation and Binding:

- Create a socket and bind it to a specified port to listen for client connections.

2.Shared Memory and Semaphore:

- Create and configure shared memory to store the random number.
- Initialize a semaphore for synchronizing access to shared memory.

3.Thread Management:

- One thread generates random numbers and stores them in shared memory.
- Another thread handles incoming client connections, managing them in a loop and creating new threads for additional clients if needed.

### Client Code

1.Socket Connection:
- Create a socket and connect to the server.

2.Shared Memory and Semaphore:
- Access the shared memory segment created by the server.
- Use the semaphore for synchronized read access.

3.Display Data:
- Continuously read and display the random number from shared memory.

## 7.User Manual

### Prerequisites

- Linux-based OS with POSIX support
- GCC compiler

### Compiling and Running

1. Server Setup:
    - Compile and run the server code.
    - Ensure that the server is running and listening on the specified port.
2. Client Setup:
    - Compile and run the client code.
    - Multiple clients can be launched using the provided test script.
    - The client will connect to the server and display the random numbers received from the server.
3. Termination:
    - Terminate the server and clients using appropriate termination commands

## 8.The Synchronization Mechanism

- **Semaphores:** Used for synchronizing access to shared memory between multiple client handler threads.
- **Thread Synchronization:** Ensures concurrent handling of client connections without data corruption.

## 9.Conclusion

This project demonstrates the implementation of a client-server system for generating and sharing random numbers using shared memory and sockets. By employing semaphores for synchronization, it ensures thread safety and data consistency in a multi-client environment. The provided test script facilitates testing the system with multiple clients simultaneously, validating its functionality .