MULTI-THREADED SERVER-CLIENT USING SHARED MEMORY PROJECT

**Submitted By: Ankur Kumar Gupta**

Table of Contents

[Project Overview 3](#_Toc168343784)

[Objectives 3](#_Toc168343785)

[Project Scope 3](#_Toc168343786)

[Prerequisites 3](#_Toc168343787)

[Project Requirements 4](#_Toc168343788)

[Software 4](#_Toc168343789)

[Libraries 4](#_Toc168343790)

[Code Functionality 4](#_Toc168343791)

[Server (`server.c`) 4](#_Toc168343792)

[Client (`client.c`) 5](#_Toc168343793)

[Implementation 5](#_Toc168343794)

[Server Implementation 5](#_Toc168343795)

[Client Implementation 6](#_Toc168343796)

[Mechanisms Used 7](#_Toc168343797)

[Sockets 7](#_Toc168343798)

[Shared Memory 7](#_Toc168343799)

[Semaphores 7](#_Toc168343800)

[Multi-Threading 7](#_Toc168343801)

[Troubleshooting 7](#_Toc168343802)

# Project Overview

This project demonstrates a multi-threaded client-server architecture where the server generates random numbers and shares them with clients using shared memory. The project utilizes inter-process communication (IPC) mechanisms like shared memory and semaphores to synchronize access to the shared data.

# Objectives

* Develop a multi-threaded server that can handle multiple client connections.
* Use shared memory for efficient data sharing between server and clients.
* Synchronize access to shared memory using semaphores.
* Implement a client that connects to the server and reads data from shared memory.

# Project Scope

* Creation and management of a multi-threaded server.
* Generation and sharing of random numbers using shared memory.
* Synchronization using semaphores to avoid race conditions.
* Implementation of a client that connects to the server and accesses the shared data.

# Prerequisites

* C programming language.
* Socket programming.
* Inter-process communication (IPC) mechanisms.
* Multi-threading in C.
* Linux system programming.

# Project Requirements

## Software

* GCC (GNU Compiler Collection)
* POSIX-compliant operating system (e.g., Linux)
* Basic C development tools (e.g., `make`, `gcc`)

## Libraries

* `pthread.h` for multi-threading
* `sys/socket.h`, `netinet/in.h`, `arpa/inet.h` for socket programming
* `sys/ipc.h`, `sys/shm.h` for shared memory
* `semaphore.h` for semaphore handling

# Code Functionality

## Server (`server.c`)

1. Server Setup: Initializes a socket, binds it to a port, and listens for incoming client connections.
2. Random Number Generation: A separate thread generates random numbers and stores them in shared memory.
3. Client handling: A multi-threaded approach to accept and manage multiple clients, creating new threads when the number of clients exceeds a predefined threshold.
4. Synchronization: Uses semaphores to synchronize access to shared memory, ensuring data consistency and preventing race conditions.

## Client (`client.c`)

1. Client Setup: Creates a socket and connects to the server.
2. Shared Memory Access: Opens the shared memory segment and maps it to the client's address space.
3. Data Retrieval: Reads random numbers from the shared memory synchronized using semaphores.

# Implementation

## Server Implementation

1. Socket Initialization:

* Creates a socket using `socket()`.
* Sets socket options using `setsockopt()`.
* Binds the socket to an IP address and port using `bind()`.
* Listens for incoming connections using `listen()`.

1. Thread Creation:

* Creates a thread for generating random numbers (`generate\_random\_numbers` function).
* Creates a thread for handling client connections (`handle\_clients` function).

1. Random Number Generation:

* Uses `rand()` to generate random numbers.
* Stores the generated numbers in shared memory.
* Synchronizes access to shared memory using semaphores (`sem\_wait()` and `sem\_post()`).

1. Client Handling:

* Accepts new client connections using `accept()`.
* Stores client sockets and creates new threads when the number of clients exceeds a threshold.

## Client Implementation

1. Socket Initialization:

* Creates a socket using `socket()`.
* Connects to the server using `connect()`.

1. Shared Memory Access:

* Opens the shared memory segment using `shm\_open()`.
* Maps the shared memory to the client's address space using `mmap()`.

1. Data Retrieval:

* Reads random numbers from shared memory.
* Uses semaphores for synchronized access (`sem\_wait()` and `sem\_post()`).

# Mechanisms Used

## Sockets

* Server: Listens for and accepts client connections.
* Client: Connects to the server.

## Shared Memory

* Used to store and share random numbers between server and clients.
* Created using `shmget()` and accessed using `shmat()`.

## Semaphores

* Used to synchronize access to shared memory.
* Created using `sem\_open()` and accessed using `sem\_wait()` and `sem\_post()`.

## Multi-Threading

* Server: Uses threads to handle random number generation and client connections.
* Client: Reads data from shared memory in a loop.

# Troubleshooting

* Socket Errors: Ensure the port is not already in use and the server is running.
* Shared Memory Errors: Verify that the shared memory segment is created and accessible.
* Semaphore Errors: Ensure semaphores are properly initialized and accessible.

BY:   
ANKUR KUMAR GUPTA