Documentation On

LINUX MULTI-THREADED CLIENT-SERVER

USING SHARED MEMORY

Table Of Contents

1. Overview

This project involves developing a client-server application that uses threads (pthreads), shared memory for inter-process communication and semaphore for synchronization. The server handles multiple clients concurrently, logging their interactions to a shared memory segment. Clients can send messages to the server and receive responses from the server, with both the client and server able to view the current state of the shared memory. This setup demonstrates inter-process communication mechanisms, particularly shared memory and semaphores to manage access and synchronization between multiple processes.

2. Introduction

This project demonstrates the implementation of a client-server communication system using inter-process communication (IPC) mechanisms such as shared memory and semaphores in the C programming language. The core objective is to enable multiple clients to interact with a server simultaneously, with shared memory used to store and update the exchange of messages between them.

In this system, the server is designed to handle multiple clients concurrently by spawning a new thread for each client connection. Each client can send messages to the server, which processes the message, updates the shared memory, and responds back to the client. The shared memory segment is protected by a semaphore to ensure that only one process can access it at a time, preventing race conditions and ensuring data consistency.

A unique feature of the server is its logging capability: the server logs the contents of the shared memory to a file whenever an update occurs, providing a persistent record of all communications. This is accomplished using a dedicated logging thread that monitors the shared memory for changes.

On the client side, users can send messages to the server and receive responses in real-time. Clients also have the ability to view the current state of the shared memory, thus seeing the most recent communication updates.

This project highlights the practical application of these concepts in a concurrent programming environment.

3. Objectives

The scope of the project includes:

- Implementing a server application that listens for incoming connections and handles client messages.
- Developing a client application that connects to the server, sends messages, and displays responses.
- Utilizing shared memory for storing messages exchanged between the server and clients.
- Using semaphores for synchronizing access to shared memory.

4. Requirements

4.1. Functional Requirements:

1. Server Application:

- **Initialize Shared Memory:** Set up a shared memory segment for storing messages.
- **Initialize Semaphores:** Set up semaphores to synchronize access to shared memory.
- **Accept Connections:** Listen for and accept incoming client connections.
- **Handle Clients:** Create a new thread for each client to handle communication.
- **Log Messages:** Continuously log the contents of shared memory to a file.
- **Close Connections:** Cleanly shut down client connections and free resources.

2. Client Application:

- **Connect to Server**: Establish a connection with the server.
- **Send Messages:** Send user-input messages to the server.
- **Receive Responses**: Receive and display responses from the server.
- **Display Shared Memory**: Show the current contents of shared memory after each interaction.
- **Quit:** Allow the client to quit gracefully.

4.2. Non-Functional Requirements

1. Performance:

- The server should handle multiple clients concurrently without significant delays.
- Shared memory access should be synchronized efficiently to minimize wait times.

2. Reliability:

- The server should handle client disconnections gracefully.

- The logging mechanism should ensure that all messages are recorded accurately.

3. Usability:

- The client interface should be simple and user-friendly, allowing easy message input and display.

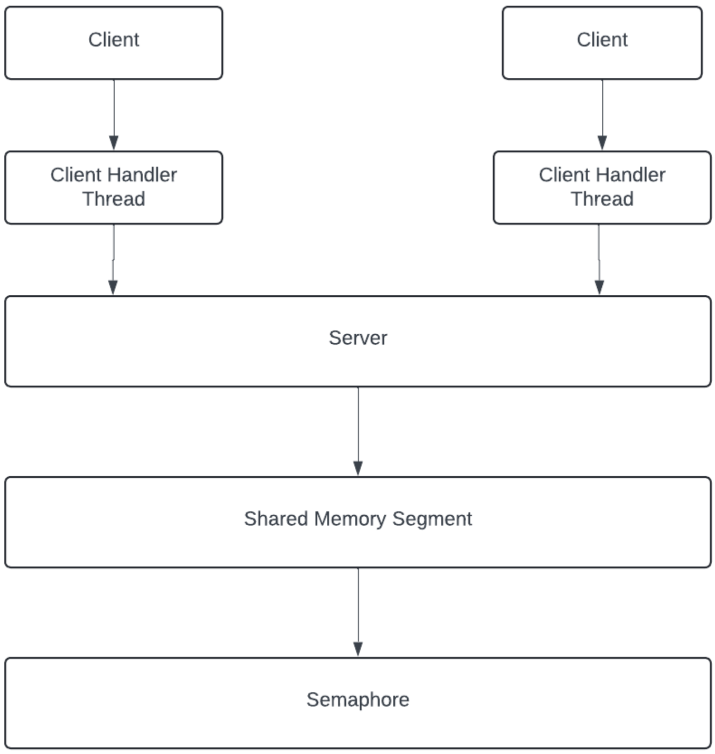
4. Scalability:

- The server design should accommodate an increase in the number of clients with minimal performance degradation.

5. Security:

- Access to shared memory should be controlled to prevent unauthorized access or data corruption.

5. System Design



6. Code comments and Explanation:

6.1. Server code:(server.c)

```
 #include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <semaphore.h>

#define PORT 9090
#define MAX_CLIENTS 100
#define SHM_KEY 4
#define SHM_SIZE 1024

sem_t *semaphore;

void *client_handler(void *socket_desc) {
    int client_sock = *(int*)socket_desc;
    free(socket_desc);

    struct sockaddr_in client;
    socklen_t client_len = sizeof(client);
    getpeername(client_sock, (struct sockaddr*)&client, &client_len);
    char client_ip[INET_ADDRSTRLEN];
    inet_ntop(AF_INET, &(client.sin_addr), client_ip, INET_ADDRSTRLEN);
    printf("Client connected: \n");

    // Access shared memory
    int shm_id = shmget(SHM_KEY, SHM_SIZE, 0666);
    if (shm_id == -1) {
        perror("Failed to open shared memory");
        return NULL;
    }

    char *shared_mem = (char*)shmat(shm_id, NULL, 0);
    if (shared_mem == (char*)-1) {
        perror("Failed to attach shared memory");
        return NULL;
    }

    // Handle client communication
    char buffer[256];
    while (1) {
        memset(buffer, 0, 256);
        int read_size = recv(client_sock, buffer, 256, 0);
        if (read_size <= 0) {
            break;
        }printf("Thread is created\n");

        printf("Received message from client : %s\n", buffer);
```

```
        sem_wait(semaphore);
        strcpy(shared_mem, buffer); // Write to shared memory
        sem_post(semaphore);

        // Echo back to client
        send(client_sock, buffer, strlen(buffer), 0);
    }

    printf("Client disconnected: \n");

    close(client_sock);
    return NULL;
}

int main() {
    int server_fd, client_sock, *new_sock;
    struct sockaddr_in server, client;
    socklen_t client_len = sizeof(client);
    pthread_t client_threads[MAX_CLIENTS];
    int client_count = 0;

    // Initialize semaphore
    semaphore = sem_open("/sem_example", O_CREAT, 0666, 1);
    if (semaphore == SEM_FAILED) {
        perror("Failed to initialize semaphore");
        return 1;
    }

    // Create socket
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == -1) {
        perror("Could not create socket");
        return 1;
    }
    printf("Socket created\n");

    // Set SO_REUSEADDR option
    int opt = 1;
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt))) {
        perror("setsockopt failed");
        close(server_fd);
        return 1;
    }

    // Prepare the sockaddr_in structure
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(PORT);

    // Bind
    if (bind(server_fd, (struct sockaddr *)&server, sizeof(server)) < 0) {
        perror("Bind failed");
        close(server_fd);
        return 1;
    }
    printf("Bind successful\n");

    // Listen
    if (listen(server_fd, 3) < 0) {
```

```
        perror("Listen failed");
        close(server_fd);
        return 1;
    }
    printf("Server listening on port %d...\n", PORT);

    // Create shared memory
    int shm_id = shmget(SHM_KEY, SHM_SIZE, IPC_CREAT | 0666);
    if (shm_id == -1) {
        perror("Failed to create shared memory");
        return 1;
    }

   // printf("Shared Memory Key: %#x\n", SHM_KEY); // Print shared memory key in hex
    printf("Shared Memory ID: %d\n", shm_id); // Print shared memory ID

    char *shared_mem = (char*)shmat(shm_id, NULL, 0);
    if (shared_mem == (char*)-1) {
        perror("Failed to attach shared memory");
        return 1;
    }

    // Accept incoming connections and handle them in separate threads
    while ((client_sock = accept(server_fd, (struct sockaddr *)&client, &client_len))) {
        printf("Connection accepted\n");

        // Create a new thread for this client
        new_sock = malloc(sizeof(int));
        *new_sock = client_sock;

        if (pthread_create(&client_threads[client_count], NULL, client_handler, (void*) new_sock)
< 0) {
            perror("Could not create thread");
            return 1;
        }

        client_count++;
        if (client_count >= MAX_CLIENTS) {
            printf("Maximum clients reached. No longer accepting connections.\n");
            break;
        }
    }

    if (client_sock < 0) {
        perror("Accept failed");
        return 1;
    }

    // Join all client threads before exiting
    for (int i = 0; i < client_count; i++) {
        pthread_join(client_threads[i], NULL);
    }

    // Cleanup
    shmctl(shm_id, IPC_RMID, NULL);
    sem_close(semaphore);
    sem_unlink("/sem_example");

    return 0;
}
```

6.2. Client Code:(client.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define SERVER_IP "127.0.0.1" // Change this to the server IP address you want to connect to
#define PORT 9090

int main() {
    int sock;
    struct sockaddr_in server;
    char message[256];
    // Create socket
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("Socket creation failed");
        return 1;
    }
    // Configure server address
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = inet_addr(SERVER_IP);
    server.sin_port = htons(PORT);
    // Connect to server
    if (connect(sock, (struct sockaddr *)&server, sizeof(server)) < 0) {
        perror("Connection failed");
        return 1;
    }
     printf("Connected to server\n");

    // Send messages to server
    while (1) {
        printf("Enter message (Q to quit): ");
        fgets(message, sizeof(message), stdin);

        // Send message to server
        if (send(sock, message, strlen(message), 0) < 0) {
            perror("Send failed");
```

```
            return 1;
        }
        // Quit on 'Q' or 'q'
        if (message[0] == 'Q' || message[0] == 'q') {
            break;
        }
        // Clear the buffer
        memset(message, 0, sizeof(message));
        // Receive server response
        if (recv(sock, message, sizeof(message), 0) < 0) {
            perror("Receive failed");
            return 1;
        }
        printf("Server response: %s\n", message);
    }
    // Close socket
    close(sock);
    printf("Disconnected from server\n");

    return 0;
}
```

Server Code Explanation:

Headers and Macros:

- Includes standard libraries for input/output, memory management, socket programming, threading, and synchronization.
- Defines constants for the max clients, shared memory key, and shared memory size.

Client Handling Thread (handle_client):

- Handles communication with a connected client.
- Updates the shared memory with messages from the client.
- Prompts the server user for a response and sends it to the client.
- Uses semaphores to ensure exclusive access to shared memory and to update the client count.

Main Function:

- Initializes shared memory and semaphores.
- Creates a socket, binds it to the specified port, and listens for incoming connections.
- Starts the logging thread.
- Accepts incoming client connections and creates a new thread to handle each client.

Client Code Explanation

Headers and Macros:

- Includes standard libraries for input/output, memory management, socket programming, and synchronization.
- Defines constants for the server port, shared memory key, and shared memory size.

Main Function:

- Initializes shared memory and semaphore.
- Creates a socket and connects it to the server.
- Enters a loop to send messages to the server and receive responses.
- After each response from the server, prints the current content of the shared memory.
- Closes the socket and terminates when the user types "exit".

User manual

This user manual provides instructions for compiling, running, and testing the server and client applications. These applications demonstrate the use of shared memory, semaphores, and threading in a networked client-server setup.

Prerequisites

- A Linux-based operating system (or any OS with shared memory and semaphore support).
- GCC compiler.
- Basic knowledge of using the terminal/command prompt.

File Description

- **server.c:** Source code for the server application.
- **client.c:** Source code for the client application.

Compiling Application:

- Open a terminal window.
- Navigate to the directory containing the source code files.
- Compile the server application:
    - gcc server.c -o server

This command compiles server.c and creates an executable named server.

- Compile the client application:
    - gcc client.c -o client

This command compiles client.c and creates an executable named client.

Running Application:

1. Running the Server

➢ Start the server application:

- ./server

The server will start and listen for incoming client connections on port 9090. It will also start a logging thread that writes shared memory updates to shared_memory_log.txt.

2. Running the Client

➢ Open another terminal window.

➢ Start the client application:

- ./client

The client will attempt to connect to the server running on 127.0.0.1 (localhost) on port 9090. Once connected, the client can send messages to the server.

Testing the Applications

Single Client Test

1. Start the server:

   - ./server

2. Run a single client instance:

   - ./client

3. Send a message from the client to the server.
4. Verify that the server receives the message and sends a response.
5. Check the shared memory contents from the client.

Multiple Client Test

1. Start the server:

   - ./server

2. Use the provided script to run multiple client instances:

   - ./test.sh

```
#!/bin/bash

# Define the number of clients to run
num_clients=50

# 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
```

Verify that the server handles all client connections concurrently.

3. Check the shared memory contents from each client.

7.The Synchronization Mechanism

The server and client applications make use of several synchronization mechanisms to ensure correct and safe access to shared resources, particularly shared memory. The primary synchronization mechanisms used are semaphores and mutexes. Below is a detailed description of each mechanism and its role in the application.

Semaphores

Semaphores are used extensively in the server and client applications to manage concurrent access to shared resources. There are two main semaphores used:

- Shared Memory Semaphore (semaphore)
- Client Count Semaphore (client_semaphore)

Shared Memory Semaphore (semaphore)

- Purpose: To synchronize access to the shared memory segment.
- Initialization:

```
semaphore = sem_open("/my_semaphore", O_CREAT, 0666, 1);
```

The semaphore is initialized with a value of 1, indicating that updates to the client count are allowed.

- Usage:

Acquiring the Semaphore (Wait/Lock):

```
sem_wait(&client_semaphore);
```

When the server needs to increment the “client_count”, it calls “sem_wait(&client_semaphore)”. This ensures that only one thread can update the count at a time.

Releasing the Semaphore (Post/Unlock):

```
sem_post(&client_semaphore);
```

After updating the “client_count”, the thread calls “sem_post(&client_semaphore)”, allowing other threads to access the client count.

Scenarios of Usage:

Server: When a new client connects, the server increments the “client_coun”t.

Thread Synchronization

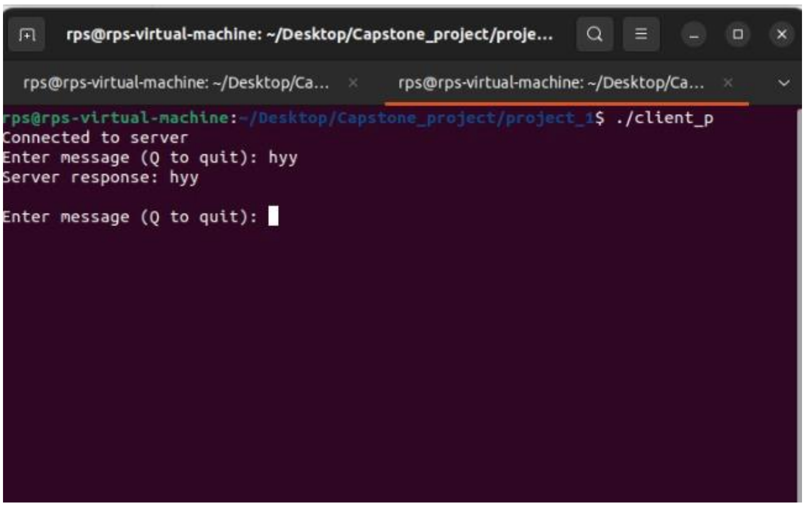
In addition to semaphores, the application relies on thread synchronization techniques to manage multiple threads:

Client Handling Threads: Each client connection is handled by a separate thread created using “pthread_create”. This allows the server to handle multiple clients concurrently.

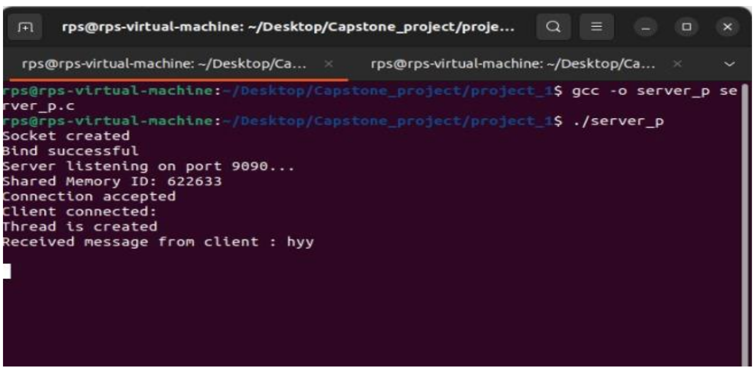
8. Test Cases and Results

Test Case 1: Single Client Communication

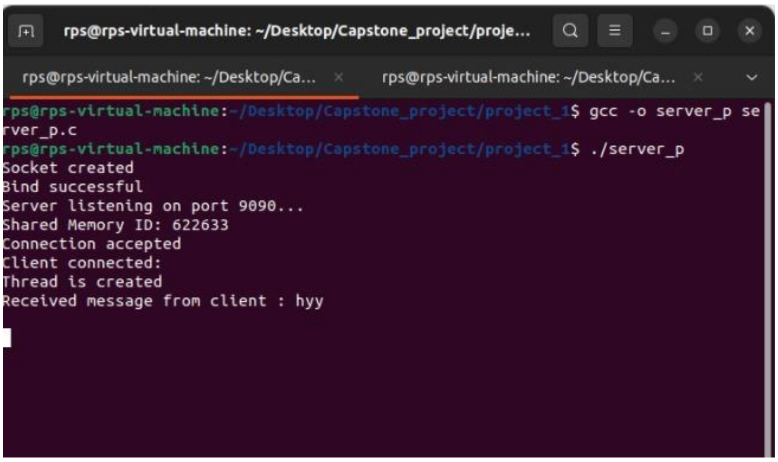
- **Test Steps**:
  1. Start the server.
  2. Run a single instance of the client application.



  3. Send a message from the client to the server.
  4. Verify that the server receives the message and sends a response.
  5. Check the shared memory contents from the client.

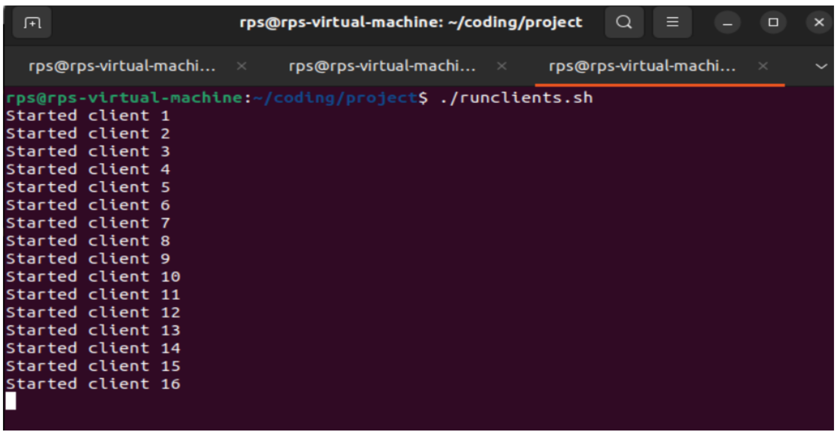


- **Expected Result**:
  - The server successfully receives the message and sends a response.
  - The shared memory contains the exchanged messages.



Test Case 2: Multiple Client Communication

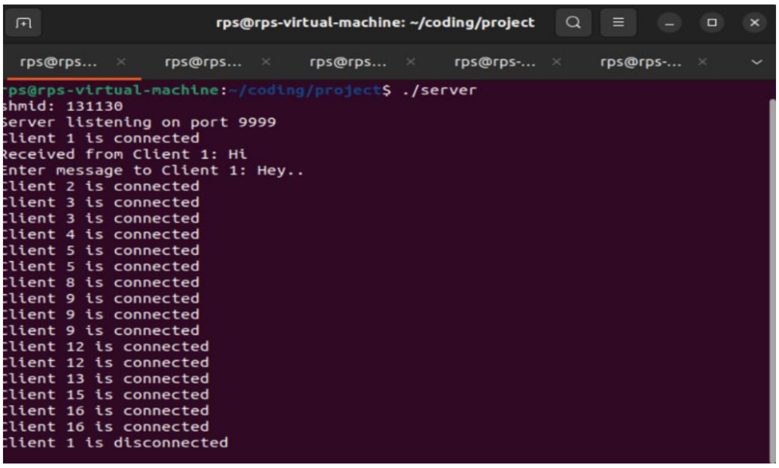
- **Test Steps**:
  1. Start the server.
  2. Run multiple instances of the client application concurrently using the provided script.



  3. Each client sends messages to the server.
  4. Verify that the server handles all client connections concurrently.

  5. Check the shared memory contents from each client.

- **Expected Result**:
  - The server handles multiple client connections concurrently without errors.



9. Conclusion

In conclusion, this project demonstrates the implementation of a client-server system using shared memory and semaphores for inter-process communication and synchronization. The server efficiently handles multiple client connections and logs communication activities. The client interacts with the server, sends messages, and displays responses along with shared memory contents. The use of shared memory and semaphores ensures reliable and synchronized communication between processes