CPU Scheduling and Socket Programming Implementation

Yousaf Maaz
FAST National University, Peshawar Campus

November 26, 2024

Contents

1 Introduction

This document outlines the implementation of CPU scheduling algorithms and socket programming concepts in both local and distributed settings. The tasks are divided as follows:

- **Task 1:** CPU Scheduling Implementation in C.
- **Task 2:** Socket Programming Implementation, including both local and distributed systems.

2 Task 1: CPU Scheduling Implementation

This section provides the implementation of the following CPU scheduling algorithms:

1. First-Come-First-Served (FCFS)
2. Shortest Job First (SJF)
3. Priority Scheduling
4. Round Robin (RR)
5. Priority with Round Robin

The task list is read from a file named `schedule.txt`, and the program processes each task accordingly.

2.1 Code

Add the CPU scheduling algorithms' code here.

Listing 1: CPU Scheduling Algorithms Implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Define a structure to represent a task
typedef struct
{
    char name[10]; // Name of the task
    int priority;  // Priority of the task (lower value = higher priority)
    int cpuBurst;  // CPU Burst Time (execution time required)
} Task;

// Function prototypes for different scheduling algorithms
void fcfs(Task tasks[], int n);                          // First Come, First Serve
void sjf(Task tasks[], int n);                           // Shortest Job First
```

```
void priorityScheduling(Task tasks[], int n);                          // Priority Scheduling
void roundRobin(Task tasks[], int n, int timeQuantum);                 // Round Robin
void priorityWithRoundRobin(Task tasks[], int n, int timeQuantum); // Priority + Round Robin

// Main function
int main()
{
    // Open the input file containing task details
    FILE *file = fopen("schedule.txt", "r");
    if (!file) // Check if the file opened successfully
    {
        printf("Error: Could not open schedule.txt.\n");
        return 1; // Exit the program if the file could not be opened
    }

    // Read tasks from the file
    Task tasks[100]; // Array to store up to 100 tasks
    int count = 0;   // Variable to count the number of tasks
    while (fscanf(file, "%[^,], %d, %d\n", tasks[count].name, &tasks[count].priority, &tasks[count].cpuBurst) != EOF)
    {
        count++; // Increment task count after reading each line
    }
    fclose(file); // Close the file after reading

    int choice, timeQuantum; // Variables for user input
    // Display menu options for CPU scheduling algorithms
    printf("CPU Scheduling Algorithms\n");
    printf("1. FCFS\n2. SJF\n3. Priority Scheduling\n4. Round Robin\n5. Priority with Round Robin\n");
    printf("Enter your choice (1-5): ");
    scanf("%d", &choice); // Take user's choice as input

    // Execute the chosen scheduling algorithm
    switch (choice)
    {
    case 1: // First Come, First Serve
        fcfs(tasks, count);
        break;
    case 2: // Shortest Job First
        sjf(tasks, count);
        break;
    case 3: // Priority Scheduling
        priorityScheduling(tasks, count);
        break;
    case 4: // Round Robin
        printf("Enter time quantum for Round Robin: ");
```

```c
            scanf("%d", &timeQuantum);
            roundRobin(tasks, count, timeQuantum);
            break;
        case 5: // Priority with Round Robin
            printf("Enter time quantum for Priority with Round
               Robin: ");
            scanf("%d", &timeQuantum);
            priorityWithRoundRobin(tasks, count, timeQuantum);
            break;
        default: // Invalid choice
            printf("Invalid choice.\n");
            break;
    }

    return 0; // End of program
}

// FCFS (First Come, First Serve) implementation
void fcfs(Task tasks[], int n)
{
    printf("Task Execution Order (FCFS):\n");
    // Tasks are executed in the order they appear
    for (int i = 0; i < n; i++)
    {
        printf("Task: %s, Priority: %d, CPU Burst Time: %d\n",
               tasks[i].name, tasks[i].priority,
                  tasks[i].cpuBurst);
    }
}

// SJF (Shortest Job First) implementation
void sjf(Task tasks[], int n)
{
    // Sort tasks by CPU Burst Time (ascending order)
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (tasks[j].cpuBurst > tasks[j + 1].cpuBurst)
            {
                Task temp = tasks[j]; // Swap tasks
                tasks[j] = tasks[j + 1];
                tasks[j + 1] = temp;
            }
        }
    }

    // Print tasks after sorting
    printf("Task Execution Order (SJF):\n");
    for (int i = 0; i < n; i++)
    {
```

```c
        printf("Task: %s, Priority: %d, CPU Burst Time: %d\n",
                tasks[i].name, tasks[i].priority, tasks[i].cpuBurst);
    }
}

// Priority Scheduling implementation
void priorityScheduling(Task tasks[], int n)
{
    // Sort tasks by Priority (ascending order, lower value = higher priority)
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (tasks[j].priority > tasks[j + 1].priority)
            {
                Task temp = tasks[j]; // Swap tasks
                tasks[j] = tasks[j + 1];
                tasks[j + 1] = temp;
            }
        }
    }

    // Print tasks after sorting
    printf("Task Execution Order (Priority Scheduling):\n");
    for (int i = 0; i < n; i++)
    {
        printf("Task: %s, Priority: %d, CPU Burst Time: %d\n",
                tasks[i].name, tasks[i].priority, tasks[i].cpuBurst);
    }
}

// Round Robin implementation
void roundRobin(Task tasks[], int n, int timeQuantum)
{
    int remainingBurst[n]; // Array to track remaining burst time for each task
    for (int i = 0; i < n; i++)
    {
        remainingBurst[i] = tasks[i].cpuBurst; // Initialize remaining burst times
    }

    int time = 0; // Keep track of total execution time
    printf("Task Execution Order (Round Robin):\n");
    while (1) // Continue until all tasks are completed
    {
        int done = 1; // Flag to check if all tasks are done
        for (int i = 0; i < n; i++)
```

```
        {
            if (remainingBurst[i] > 0) // If task is not yet completed
            {
                done = 0; // Mark as not done
                if (remainingBurst[i] > timeQuantum)
                {
                    time += timeQuantum;
                    remainingBurst[i] -= timeQuantum;
                    printf("Task: %s executed for %d units.\n", tasks[i].name, timeQuantum);
                }
                else
                {
                    time += remainingBurst[i];
                    printf("Task: %s executed for %d units.\n", tasks[i].name, remainingBurst[i]);
                    remainingBurst[i] = 0; // Mark task as completed
                }
            }
        }
        if (done) // Exit loop if all tasks are completed
            break;
    }
    printf("Total Time: %d\n", time); // Print total execution time
}

// Priority with Round Robin implementation
void priorityWithRoundRobin(Task tasks[], int n, int timeQuantum)
{
    // Sort tasks by priority first
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (tasks[j].priority > tasks[j + 1].priority)
            {
                Task temp = tasks[j]; // Swap tasks
                tasks[j] = tasks[j + 1];
                tasks[j + 1] = temp;
            }
        }
    }

    // Apply Round Robin on sorted tasks
    int remainingBurst[n]; // Track remaining burst times
    for (int i = 0; i < n; i++)
    {
        remainingBurst[i] = tasks[i].cpuBurst;
```

```c
    }

    int time = 0; // Track total execution time
    printf("Task Execution Order (Priority with Round Robin):\n");
    while (1) // Continue until all tasks are completed
    {
        int done = 1; // Flag to check if all tasks are done
        for (int i = 0; i < n; i++)
        {
            if (remainingBurst[i] > 0) // If task is not yet completed
            {
                done = 0; // Mark as not done
                if (remainingBurst[i] > timeQuantum)
                {
                    time += timeQuantum;
                    remainingBurst[i] -= timeQuantum;
                    printf("Task: %s executed for %d units (Priority: %d).\n",
                           tasks[i].name, timeQuantum, tasks[i].priority);
                }
                else
                {
                    time += remainingBurst[i];
                    printf("Task: %s executed for %d units (Priority: %d).\n",
                           tasks[i].name, remainingBurst[i], tasks[i].priority);
                    remainingBurst[i] = 0; // Mark task as completed
                }
            }
        }
        if (done) // Exit loop if all tasks are completed
            break;
    }
    printf("Total Time: %d\n", time); // Print total execution time
}
```

2.2 Execution Screenshot

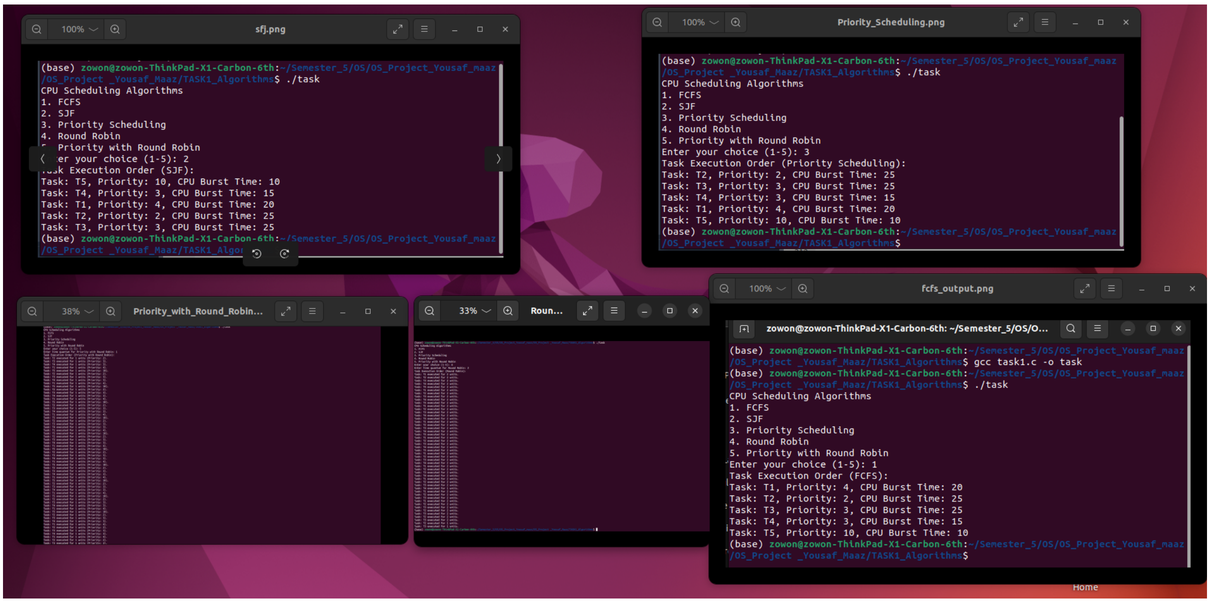

Figure 1: Execution of CPU Scheduling Program

3 Task 2: Socket Programming Implementation

Socket programming is implemented in two parts: local system and distributed system.

3.1 Part 1: Local System Implementation

In this part, a server communicates with multiple clients on the same system. The server broadcasts messages to all connected clients, and clients send messages to the server.

3.1.1 Code

Add the local system's server and client code here.

Listing 2: Local System Server Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <pthread.h>

// Define the server port number
#define PORT 8080

// Define the maximum number of clients that can connect
#define MAX_CLIENTS 10

// Array to store client sockets
int client_sockets[MAX_CLIENTS];
int client_count = 0;
```

```c

// Mutex to synchronize access to shared resources
pthread_mutex_t lock;

// Function to handle communication with a client
void *handle_client(void *arg)
{
    int client_socket = *(int *)arg; // Retrieve the client socket passed as an argument
    char buffer[1024];               // Buffer to store messages from the client

    while (1)
    {
        // Clear the buffer before receiving a new message
        memset(buffer, 0, sizeof(buffer));

        // Receive a message from the client
        int bytes_received = recv(client_socket, buffer, sizeof(buffer), 0);
        if (bytes_received <= 0)
        {
            // If no bytes are received, the client has disconnected
            printf("Client disconnected.\n");
            close(client_socket); // Close the client socket
            pthread_exit(NULL);   // Exit the thread
        }

        // Print the received message to the server console
        printf("Received from client: %s\n", buffer);

        // Broadcast the received message to all other connected clients
        pthread_mutex_lock(&lock); // Lock the mutex before accessing shared resources
        for (int i = 0; i < client_count; i++)
        {
            if (client_sockets[i] != client_socket)
            {
                // Send the message to all clients except the sender
                send(client_sockets[i], buffer, strlen(buffer), 0);
            }
        }
        pthread_mutex_unlock(&lock); // Unlock the mutex after broadcasting
    }
}

```

```c
int main()
{
    int server_fd, new_socket;      // Server socket and client socket
    struct sockaddr_in address;     // Structure to hold server address details
    int addrlen = sizeof(address); // Size of the address structure

    // Initialize the mutex for synchronizing shared resources
    pthread_mutex_init(&lock, NULL);

    // Create the server socket
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("Socket creation failed"); // Print error if socket creation fails
        exit(EXIT_FAILURE);
    }

    // Define server address properties
    address.sin_family = AF_INET;          // Use IPv4
    address.sin_addr.s_addr = INADDR_ANY; // Accept connections from any IP address
    address.sin_port = htons(PORT);        // Convert port number to network byte order

    // Bind the socket to the specified IP and port
    if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) < 0)
    {
        perror("Bind failed"); // Print error if binding fails
        exit(EXIT_FAILURE);
    }

    // Start listening for incoming connections
    if (listen(server_fd, MAX_CLIENTS) < 0)
    {
        perror("Listen failed"); // Print error if listening fails
        exit(EXIT_FAILURE);
    }

    printf("Server is listening on port %d\n", PORT);

    while (1)
    {
        // Accept a new client connection
        if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (socklen_t *)&addrlen)) < 0)
        {
```

```c
            perror("Accept failed"); // Print error if accepting a connection fails
            continue;                // Skip to the next iteration to handle other clients
        }

        printf("New client connected.\n");

        // Add the new client socket to the client sockets array
        pthread_mutex_lock(&lock); // Lock the mutex before modifying the array
        client_sockets[client_count++] = new_socket;
        pthread_mutex_unlock(&lock); // Unlock the mutex after modification

        // Create a new thread to handle communication with this client
        pthread_t thread_id;
        if (pthread_create(&thread_id, NULL, handle_client, (void *)&new_socket) != 0)
        {
            perror("Thread creation failed"); // Print error if thread creation fails
        }
    }

    // Clean up resources
    pthread_mutex_destroy(&lock); // Destroy the mutex
    close(server_fd);             // Close the server socket

    return 0;
}
```

Listing 3: Local System Client Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <pthread.h>

#define PORT 8080 // Define the port number for the connection

// Function to handle receiving messages from the server
void *receive_messages(void *arg)
{
    int socket_fd = *(int *)arg; // Cast the argument to an integer (socket descriptor)
    char buffer[1024];           // Buffer to store received messages
```

```
    while (1)
    {
        memset(buffer, 0, sizeof(buffer));
                                                                // Clear the buffer
        int bytes_received = recv(socket_fd, buffer,
            sizeof(buffer), 0); // Receive message from the server
        if (bytes_received <= 0)
        { // Check if the connection is lost
            printf("Disconnected from server.\n");
            pthread_exit(NULL); // Exit the thread if
                disconnected
        }
        printf("Server: %s\n", buffer); // Print the message
            from the server
    }
}

int main()
{
    int socket_fd;                          // Socket file descriptor
    struct sockaddr_in server_address; // Structure to hold
        server address details
    char message[1024];                     // Buffer to hold
        messages to send

    // Create a socket
    if ((socket_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("Socket failed"); // Print error if socket
            creation fails
        exit(EXIT_FAILURE);      // Exit the program
    }

    // Configure the server address
    server_address.sin_family = AF_INET;   // Use IPv4
    server_address.sin_port = htons(PORT); // Set the port
        number in network byte order

    // Convert IP address from text to binary form
    if (inet_pton(AF_INET, "127.0.0.1",
        &server_address.sin_addr) <= 0)
    {
        perror("Invalid address"); // Print error if address
            conversion fails
        exit(EXIT_FAILURE);        // Exit the program
    }

    // Connect to the server
    if (connect(socket_fd, (struct sockaddr *)&server_address,
        sizeof(server_address)) < 0)
```

```
    {
        perror("Connection failed"); // Print error if connection fails
        exit(EXIT_FAILURE);          // Exit the program
    }

    printf("Connected to the server. Type 'exit' to disconnect.\n");

    // Create a thread to handle receiving messages from the server
    pthread_t thread_id;
    if (pthread_create(&thread_id, NULL, receive_messages, (void *)&socket_fd) != 0)
    {
        perror("Thread creation failed"); // Print error if thread creation fails
        return -1;
    }

    // Main loop to send messages to the server
    while (1)
    {
        memset(message, 0, sizeof(message)); // Clear the message buffer
        printf("You: ");
        fgets(message, sizeof(message), stdin); // Read input from the user
        message[strcspn(message, "\n")] = 0;    // Remove the newline character

        // Check if the user wants to disconnect
        if (strcmp(message, "exit") == 0)
        {
            printf("Disconnecting...\n");
            break; // Exit the loop
        }

        // Send the message to the server
        send(socket_fd, message, strlen(message), 0);
    }

    // Close the socket
    close(socket_fd);
    return 0;
}
```

3.1.2 Execution Screenshot

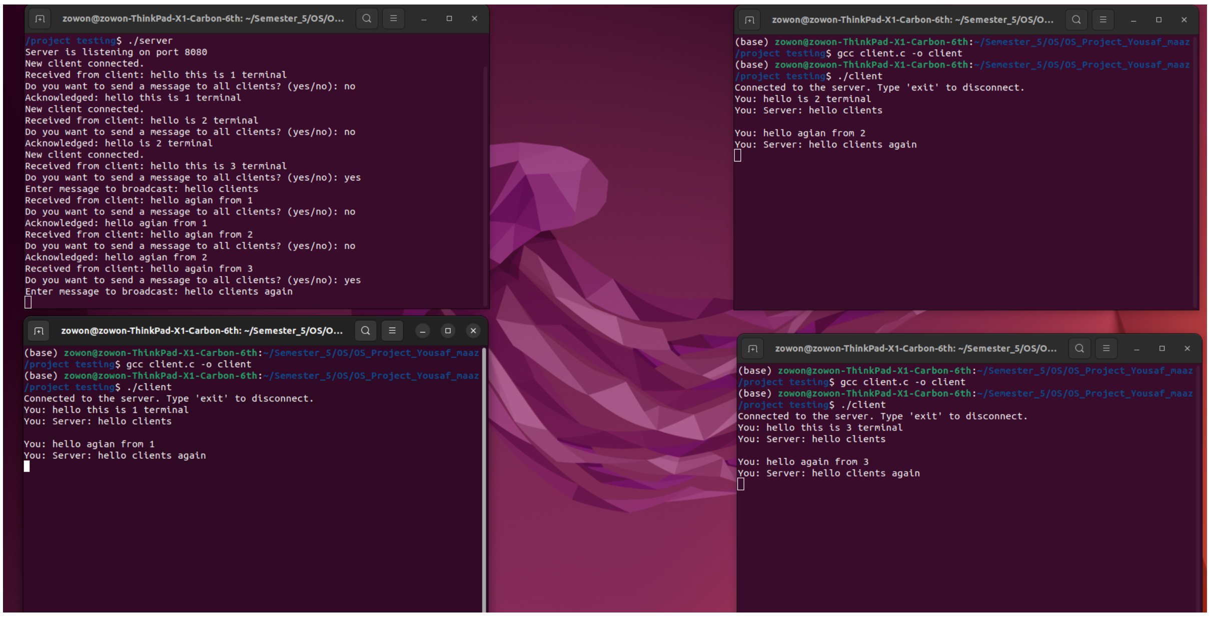

Figure 2: Server and Client Communication on Local System

3.2 Part 2: Distributed System Implementation

In this part, a server and a client communicate across two separate laptops using socket programming.

3.2.1 Code

Add the distributed system's server and client code here.

Listing 4: Distributed System Server Code

```c
#include <stdio.h>       // Standard input/output functions
#include <stdlib.h>      // Standard library functions
#include <string.h>      // String manipulation functions
#include <sys/socket.h>  // Socket programming functions
#include <arpa/inet.h>   // Internet address manipulation functions
#include <unistd.h>      // POSIX API (e.g., close)

#define PORT 8080          // Port number for the server
#define MAX_CONNECTIONS 5  // Maximum number of pending connections

int main()
{
    int server_fd, client_socket;  // Server and client socket file descriptors
    struct sockaddr_in server_address, client_address;  // Server and client address structures
    int addr_len = sizeof(client_address);  // Length of the client address structure
    char buffer[1024] = {0};  // Buffer to store messages
```

```c
    char *welcome_message = "Server: Connection established.\n"; // Welcome message for the client

    // Step 1: Create a socket
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("Socket failed"); // Print error if socket creation fails
        exit(EXIT_FAILURE);      // Exit with failure
    }

    // Step 2: Configure server address structure
    server_address.sin_family = AF_INET;         // Use IPv4
    server_address.sin_addr.s_addr = INADDR_ANY; // Accept connections from any IP address
    server_address.sin_port = htons(PORT);       // Convert port number to network byte order

    // Step 3: Bind the socket to the specified IP and port
    if (bind(server_fd, (struct sockaddr *)&server_address, sizeof(server_address)) < 0)
    {
        perror("Bind failed"); // Print error if binding fails
        exit(EXIT_FAILURE);    // Exit with failure
    }

    // Step 4: Listen for incoming connections
    if (listen(server_fd, MAX_CONNECTIONS) < 0)
    {
        perror("Listen failed"); // Print error if listening fails
        exit(EXIT_FAILURE);      // Exit with failure
    }

    printf("Server is listening on port %d...\n", PORT);

    // Step 5: Accept a client connection
    if ((client_socket = accept(server_fd, (struct sockaddr *)&client_address, (socklen_t *)&addr_len)) < 0)
    {
        perror("Accept failed"); // Print error if accepting a connection fails
        exit(EXIT_FAILURE);      // Exit with failure
    }

    // Print details of the connected client
    printf("Client connected from IP: %s, Port: %d\n",
           inet_ntoa(client_address.sin_addr), ntohs(client_address.sin_port));

    // Step 6: Send a welcome message to the client
```

```
    send(client_socket, welcome_message, strlen(welcome_message), 0);

    // Step 7: Communicate with the client
    while (1)
    {
        // Receive a message from the client
        int valread = read(client_socket, buffer, sizeof(buffer));
        if (valread > 0)
        {
            buffer[valread] = '\0';        // Null-terminate the received message
            printf("Client: %s", buffer); // Display the client's message
        }
        else
        {
            // If no bytes are read, the client has disconnected
            printf("Client disconnected.\n");
            break;
        }

        // Prompt the server to enter a response
        printf("Enter message for client: ");
        fgets(buffer, sizeof(buffer), stdin);              // Read input from the server user
        send(client_socket, buffer, strlen(buffer), 0); // Send the message to the client
    }

    // Step 8: Close the connections
    close(client_socket); // Close the client socket
    close(server_fd);     // Close the server socket

    return 0; // Exit successfully
}
```

Listing 5: Distributed System Client Code

```
#include <stdio.h>       // Standard input/output functions
#include <stdlib.h>      // Standard library functions
#include <string.h>      // String manipulation functions
#include <sys/socket.h> // Socket programming functions
#include <arpa/inet.h>  // Internet address manipulation functions
#include <unistd.h>      // POSIX API (e.g., close)

#define PORT 8080 // The port number to connect to the server

int main()
{
```

```
    int client_socket;                        // File descriptor for the client socket
    struct sockaddr_in server_address; // Structure to store the server's address
    char buffer[1024] = {0};               // Buffer to store messages

    // Step 1: Create a socket
    if ((client_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("Socket creation failed"); // Print error if socket creation fails
        exit(EXIT_FAILURE);               // Exit with failure
    }

    // Step 2: Configure server address structure
    server_address.sin_family = AF_INET;   // Use IPv4
    server_address.sin_port = htons(PORT); // Convert port number to network byte order

    // Step 3: Convert server IP address to binary form
    if (inet_pton(AF_INET, "192.168.1.55", &server_address.sin_addr) <= 0)
    {
        perror("Invalid address or address not supported"); // Error if the IP is invalid
        exit(EXIT_FAILURE);                                  // Exit with failure
    }

    // Step 4: Connect to the server
    if (connect(client_socket, (struct sockaddr *)&server_address, sizeof(server_address)) < 0)
    {
        perror("Connection failed"); // Print error if connection fails
        exit(EXIT_FAILURE);          // Exit with failure
    }

    printf("Connected to the server.\n");

    // Step 5: Receive the welcome message from the server
    int valread = read(client_socket, buffer, sizeof(buffer)); // Read the welcome message
    if (valread > 0)
    {
        buffer[valread] = '\0'; // Null-terminate the received message
        printf("%s", buffer);   // Print the welcome message
    }

```

```
    // Step 6: Communicate with the server
    while (1)
    {
        // Send a message to the server
        printf("Enter message for server: ");
        fgets(buffer, sizeof(buffer), stdin);          // Read input from the user
        send(client_socket, buffer, strlen(buffer), 0); // Send the message to the server

        // Receive a response from the server
        valread = read(client_socket, buffer, sizeof(buffer)); // Read the server's response
        if (valread > 0)
        {
            buffer[valread] = '\0';       // Null-terminate the received message
            printf("Server: %s", buffer); // Print the server's response
        }
        else
        {
            // If no bytes are read, the server has disconnected
            printf("Server disconnected.\n");
            break;
        }
    }

    // Step 7: Close the connection
    close(client_socket); // Close the client socket

    return 0; // Exit successfully
}
```

3.2.2 Execution Screenshot

4 Conclusion

The tasks successfully demonstrate the implementation of CPU scheduling algorithms and socket programming concepts in both local and distributed systems. The execution results validate the functionality of the programs.

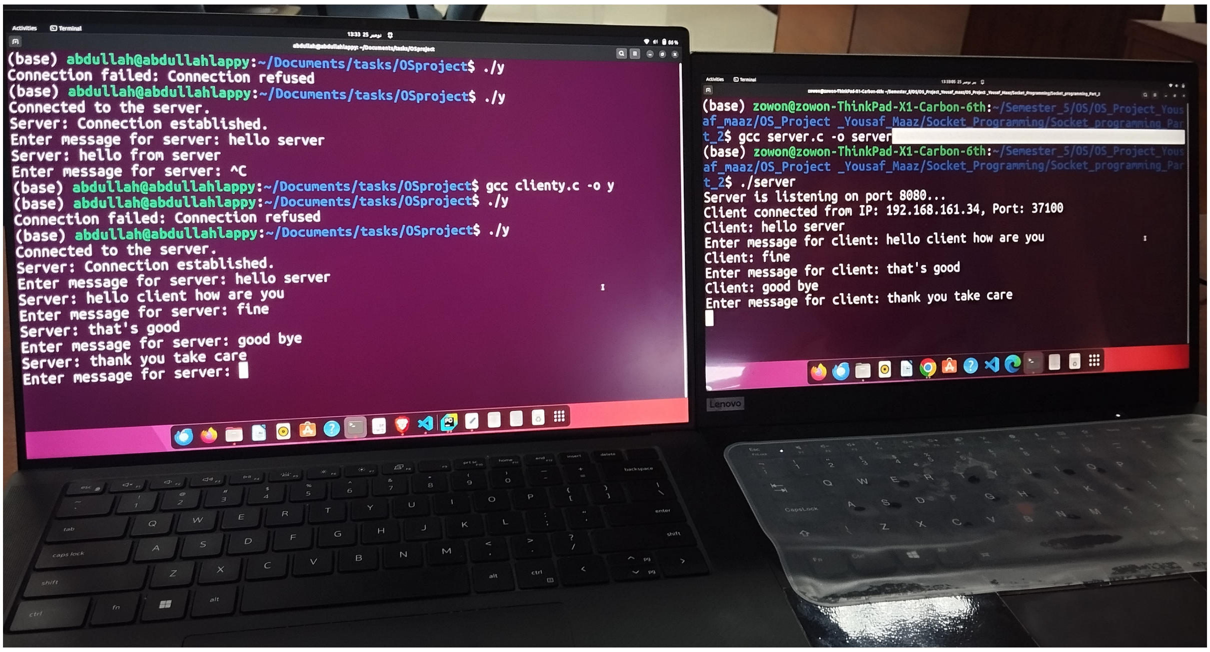

Figure 3: Server and Client Communication in Distributed System