

CPU Scheduling and Socket Programming Implementation

Yousaf Maaz
FAST National University, Peshawar Campus

November 26, 2024

Contents

1	Introduction	2
2	Task 1: CPU Scheduling Implementation	2
2.1	Code	2
2.2	Execution Screenshot	7
3	Task 2: Socket Programming Implementation	8
3.1	Part 1: Local System Implementation	8
3.1.1	Code	8
3.1.2	Execution Screenshot	13
3.2	Part 2: Distributed System Implementation	14
3.2.1	Code	14
3.2.2	Execution Screenshot	18
4	Conclusion	18

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

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 // Define a structure to represent a task
6 typedef struct
7 {
8     char name[10]; // Name of the task
9     int priority;  // Priority of the task (lower value = higher
10                    // priority)
11     int cpuBurst; // CPU Burst Time (execution time required)
12 } Task;
13
14 // Function prototypes for different scheduling algorithms
15 void fcfs(Task tasks[], int n); // First Come, First Serve
16 void sjf(Task tasks[], int n);  // Shortest Job First
```

```

16 void priorityScheduling(Task tasks[], int n);
    // Priority Scheduling
17 void roundRobin(Task tasks[], int n, int timeQuantum);
    // Round Robin
18 void priorityWithRoundRobin(Task tasks[], int n, int
    timeQuantum); // Priority + Round Robin
19
20 // Main function
21 int main()
22 {
23     // Open the input file containing task details
24     FILE *file = fopen("schedule.txt", "r");
25     if (!file) // Check if the file opened successfully
26     {
27         printf("Error: Could not open schedule.txt.\n");
28         return 1; // Exit the program if the file could not be
            opened
29     }
30
31     // Read tasks from the file
32     Task tasks[100]; // Array to store up to 100 tasks
33     int count = 0;    // Variable to count the number of tasks
34     while (fscanf(file, "%[^,], %d, %d\n", tasks[count].name,
        &tasks[count].priority, &tasks[count].cpuBurst) != EOF)
35     {
36         count++; // Increment task count after reading each line
37     }
38     fclose(file); // Close the file after reading
39
40     int choice, timeQuantum; // Variables for user input
41     // Display menu options for CPU scheduling algorithms
42     printf("CPU Scheduling Algorithms\n");
43     printf("1. FCFS\n2. SJF\n3. Priority Scheduling\n4. Round
        Robin\n5. Priority with Round Robin\n");
44     printf("Enter your choice (1-5): ");
45     scanf("%d", &choice); // Take user's choice as input
46
47     // Execute the chosen scheduling algorithm
48     switch (choice)
49     {
50     case 1: // First Come, First Serve
51         fcfs(tasks, count);
52         break;
53     case 2: // Shortest Job First
54         sjf(tasks, count);
55         break;
56     case 3: // Priority Scheduling
57         priorityScheduling(tasks, count);
58         break;
59     case 4: // Round Robin
60         printf("Enter time quantum for Round Robin: ");

```

```

61     scanf("%d", &timeQuantum);
62     roundRobin(tasks, count, timeQuantum);
63     break;
64 case 5: // Priority with Round Robin
65     printf("Enter time quantum for Priority with Round
66           Robin: ");
67     scanf("%d", &timeQuantum);
68     priorityWithRoundRobin(tasks, count, timeQuantum);
69     break;
70 default: // Invalid choice
71     printf("Invalid choice.\n");
72     break;
73 }
74
75 return 0; // End of program
76 }
77
78 // FCFS (First Come, First Serve) implementation
79 void fcfs(Task tasks[], int n)
80 {
81     printf("Task Execution Order (FCFS):\n");
82     // Tasks are executed in the order they appear
83     for (int i = 0; i < n; i++)
84     {
85         printf("Task: %s, Priority: %d, CPU Burst Time: %d\n",
86               tasks[i].name, tasks[i].priority,
87               tasks[i].cpuBurst);
88     }
89 }
90
91 // SJF (Shortest Job First) implementation
92 void sjf(Task tasks[], int n)
93 {
94     // Sort tasks by CPU Burst Time (ascending order)
95     for (int i = 0; i < n - 1; i++)
96     {
97         for (int j = 0; j < n - i - 1; j++)
98         {
99             if (tasks[j].cpuBurst > tasks[j + 1].cpuBurst)
100             {
101                 Task temp = tasks[j]; // Swap tasks
102                 tasks[j] = tasks[j + 1];
103                 tasks[j + 1] = temp;
104             }
105         }
106     }
107
108     // Print tasks after sorting
109     printf("Task Execution Order (SJF):\n");
110     for (int i = 0; i < n; i++)
111     {

```

```

110         printf("Task: %s, Priority: %d, CPU Burst Time: %d\n",
111                tasks[i].name, tasks[i].priority,
112                tasks[i].cpuBurst);
113     }
114 }
115 // Priority Scheduling implementation
116 void priorityScheduling(Task tasks[], int n)
117 {
118     // Sort tasks by Priority (ascending order, lower value =
119     // higher priority)
120     for (int i = 0; i < n - 1; i++)
121     {
122         for (int j = 0; j < n - i - 1; j++)
123         {
124             if (tasks[j].priority > tasks[j + 1].priority)
125             {
126                 Task temp = tasks[j]; // Swap tasks
127                 tasks[j] = tasks[j + 1];
128                 tasks[j + 1] = temp;
129             }
130         }
131     }
132     // Print tasks after sorting
133     printf("Task Execution Order (Priority Scheduling):\n");
134     for (int i = 0; i < n; i++)
135     {
136         printf("Task: %s, Priority: %d, CPU Burst Time: %d\n",
137                tasks[i].name, tasks[i].priority,
138                tasks[i].cpuBurst);
139     }
140 }
141 // Round Robin implementation
142 void roundRobin(Task tasks[], int n, int timeQuantum)
143 {
144     int remainingBurst[n]; // Array to track remaining burst
145     // time for each task
146     for (int i = 0; i < n; i++)
147     {
148         remainingBurst[i] = tasks[i].cpuBurst; // Initialize
149         // remaining burst times
150     }
151     int time = 0; // Keep track of total execution time
152     printf("Task Execution Order (Round Robin):\n");
153     while (1) // Continue until all tasks are completed
154     {
155         int done = 1; // Flag to check if all tasks are done
156         for (int i = 0; i < n; i++)

```

```

156     {
157         if (remainingBurst[i] > 0) // If task is not yet
158             completed
159         {
160             done = 0; // Mark as not done
161             if (remainingBurst[i] > timeQuantum)
162             {
163                 time += timeQuantum;
164                 remainingBurst[i] -= timeQuantum;
165                 printf("Task: %s executed for %d units.\n",
166                     tasks[i].name, timeQuantum);
167             }
168             else
169             {
170                 time += remainingBurst[i];
171                 printf("Task: %s executed for %d units.\n",
172                     tasks[i].name, remainingBurst[i]);
173                 remainingBurst[i] = 0; // Mark task as
174                     completed
175             }
176         }
177     }
178     if (done) // Exit loop if all tasks are completed
179         break;
180 }
181 printf("Total Time: %d\n", time); // Print total execution
182     time
183 }
184
185 // Priority with Round Robin implementation
186 void priorityWithRoundRobin(Task tasks[], int n, int timeQuantum)
187 {
188     // Sort tasks by priority first
189     for (int i = 0; i < n - 1; i++)
190     {
191         for (int j = 0; j < n - i - 1; j++)
192         {
193             if (tasks[j].priority > tasks[j + 1].priority)
194             {
195                 Task temp = tasks[j]; // Swap tasks
196                 tasks[j] = tasks[j + 1];
197                 tasks[j + 1] = temp;
198             }
199         }
200     }
201
202     // Apply Round Robin on sorted tasks
203     int remainingBurst[n]; // Track remaining burst times
204     for (int i = 0; i < n; i++)
205     {
206         remainingBurst[i] = tasks[i].cpuBurst;
207     }

```

```

202 }
203
204 int time = 0; // Track total execution time
205 printf("Task Execution Order (Priority with Round
206 Robin):\n");
207 while (1) // Continue until all tasks are completed
208 {
209     int done = 1; // Flag to check if all tasks are done
210     for (int i = 0; i < n; i++)
211     {
212         if (remainingBurst[i] > 0) // If task is not yet
213             completed
214         {
215             done = 0; // Mark as not done
216             if (remainingBurst[i] > timeQuantum)
217             {
218                 time += timeQuantum;
219                 remainingBurst[i] -= timeQuantum;
220                 printf("Task: %s executed for %d units
221 (Priority: %d).\n",
222                     tasks[i].name, timeQuantum,
223                     tasks[i].priority);
224             }
225             else
226             {
227                 time += remainingBurst[i];
228                 printf("Task: %s executed for %d units
229 (Priority: %d).\n",
230                     tasks[i].name, remainingBurst[i],
231                     tasks[i].priority);
232                 remainingBurst[i] = 0; // Mark task as
233                 completed
234             }
235         }
236     }
237     if (done) // Exit loop if all tasks are completed
238         break;
239 }
240 printf("Total Time: %d\n", time); // Print total execution
241 time

```

2.2 Execution Screenshot

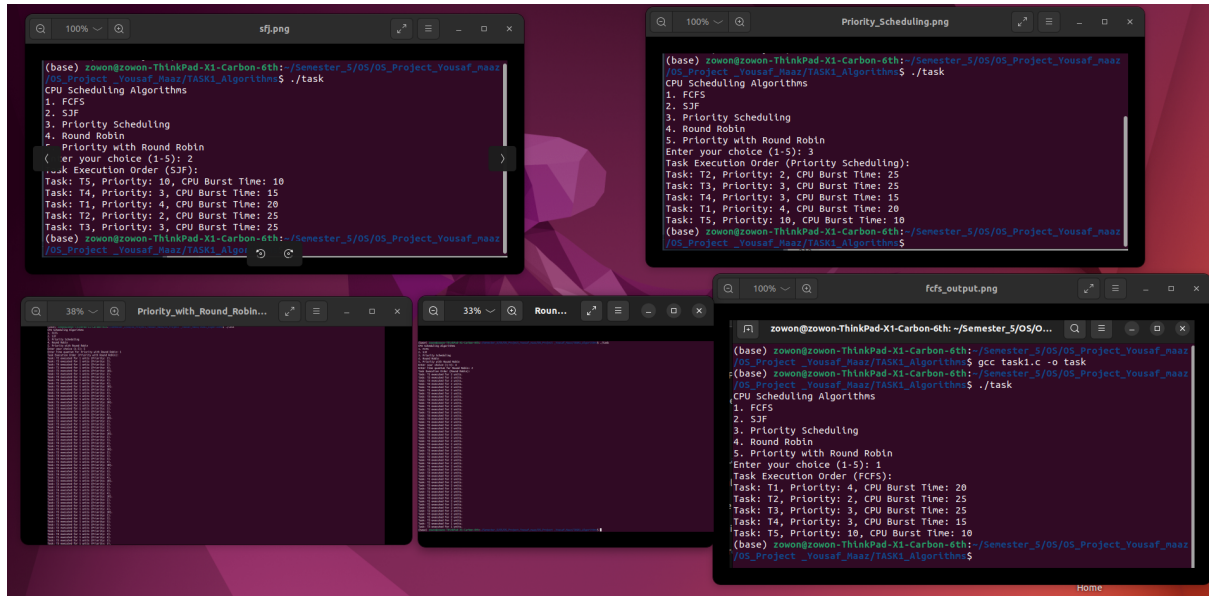


Figure 1: Execution of CPU Scheduling Program

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

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/socket.h>
5 #include <arpa/inet.h>
6 #include <unistd.h>
7 #include <pthread.h>
8
9 // Define the server port number
10 #define PORT 8080
11
12 // Define the maximum number of clients that can connect
13 #define MAX_CLIENTS 10
14
15 // Array to store client sockets
16 int client_sockets[MAX_CLIENTS];
17 int client_count = 0;

```



```

18
19 // Mutex to synchronize access to shared resources
20 pthread_mutex_t lock;
21
22 // Function to handle communication with a client
23 void *handle_client(void *arg)
24 {
25     int client_socket = *(int *)arg; // Retrieve the client
26     // socket passed as an argument
27     char buffer[1024];                // Buffer to store messages
28     // from the client
29
30     while (1)
31     {
32         // Clear the buffer before receiving a new message
33         memset(buffer, 0, sizeof(buffer));
34
35         // Receive a message from the client
36         int bytes_received = recv(client_socket, buffer,
37         sizeof(buffer), 0);
38         if (bytes_received <= 0)
39         {
40             // If no bytes are received, the client has
41             // disconnected
42             printf("Client disconnected.\n");
43             close(client_socket); // Close the client socket
44             pthread_exit(NULL);   // Exit the thread
45         }
46
47         // Print the received message to the server console
48         printf("Received from client: %s\n", buffer);
49
50         // Broadcast the received message to all other connected
51         // clients
52         pthread_mutex_lock(&lock); // Lock the mutex before
53         // accessing shared resources
54         for (int i = 0; i < client_count; i++)
55         {
56             if (client_sockets[i] != client_socket)
57             {
58                 // Send the message to all clients except the
59                 // sender
60                 send(client_sockets[i], buffer, strlen(buffer),
61                 0);
62             }
63         }
64         pthread_mutex_unlock(&lock); // Unlock the mutex after
65         // broadcasting
66     }
67 }

```

```

60 int main()
61 {
62     int server_fd, new_socket;    // Server socket and client
        socket
63     struct sockaddr_in address;    // Structure to hold server
        address details
64     int addrlen = sizeof(address); // Size of the address
        structure
65
66     // Initialize the mutex for synchronizing shared resources
67     pthread_mutex_init(&lock, NULL);
68
69     // Create the server socket
70     if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
71     {
72         perror("Socket creation failed"); // Print error if
            socket creation fails
73         exit(EXIT_FAILURE);
74     }
75
76     // Define server address properties
77     address.sin_family = AF_INET;    // Use IPv4
78     address.sin_addr.s_addr = INADDR_ANY; // Accept connections
        from any IP address
79     address.sin_port = htons(PORT);    // Convert port number
        to network byte order
80
81     // Bind the socket to the specified IP and port
82     if (bind(server_fd, (struct sockaddr *)&address,
        sizeof(address)) < 0)
83     {
84         perror("Bind failed"); // Print error if binding fails
85         exit(EXIT_FAILURE);
86     }
87
88     // Start listening for incoming connections
89     if (listen(server_fd, MAX_CLIENTS) < 0)
90     {
91         perror("Listen failed"); // Print error if listening
            fails
92         exit(EXIT_FAILURE);
93     }
94
95     printf("Server is listening on port %d\n", PORT);
96
97     while (1)
98     {
99         // Accept a new client connection
100         if ((new_socket = accept(server_fd, (struct sockaddr
            *)&address, (socklen_t *)&addrlen)) < 0)
101         {

```

```

102         perror("Accept failed"); // Print error if accepting
           a connection fails
103         continue;                // Skip to the next
           iteration to handle other clients
104     }
105
106     printf("New client connected.\n");
107
108     // Add the new client socket to the client sockets array
109     pthread_mutex_lock(&lock); // Lock the mutex before
           modifying the array
110     client_sockets[client_count++] = new_socket;
111     pthread_mutex_unlock(&lock); // Unlock the mutex after
           modification
112
113     // Create a new thread to handle communication with this
           client
114     pthread_t thread_id;
115     if (pthread_create(&thread_id, NULL, handle_client,
           (void *)&new_socket) != 0)
116     {
117         perror("Thread creation failed"); // Print error if
           thread creation fails
118     }
119 }
120
121 // Clean up resources
122 pthread_mutex_destroy(&lock); // Destroy the mutex
123 close(server_fd);             // Close the server socket
124
125 return 0;
126 }

```

Listing 3: Local System Client Code

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/socket.h>
5 #include <arpa/inet.h>
6 #include <unistd.h>
7 #include <pthread.h>
8
9 #define PORT 8080 // Define the port number for the connection
10
11 // Function to handle receiving messages from the server
12 void *receive_messages(void *arg)
13 {
14     int socket_fd = *(int *)arg; // Cast the argument to an
           integer (socket descriptor)
15     char buffer[1024];            // Buffer to store received
           messages

```

```

16
17 while (1)
18 {
19     memset(buffer, 0, sizeof(buffer));
20                                     // Clear the buffer
21     int bytes_received = recv(socket_fd, buffer,
22                             sizeof(buffer), 0); // Receive message from the server
23     if (bytes_received <= 0)
24     { // Check if the connection is lost
25         printf("Disconnected from server.\n");
26         pthread_exit(NULL); // Exit the thread if
27                             disconnected
28     }
29     printf("Server: %s\n", buffer); // Print the message
30                                     from the server
31 }
32
33 int main()
34 {
35     int socket_fd; // Socket file descriptor
36     struct sockaddr_in server_address; // Structure to hold
37                                     server address details
38     char message[1024]; // Buffer to hold
39                                     messages to send
40
41     // Create a socket
42     if ((socket_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
43     {
44         perror("Socket failed"); // Print error if socket
45                                     creation fails
46         exit(EXIT_FAILURE); // Exit the program
47     }
48
49     // Configure the server address
50     server_address.sin_family = AF_INET; // Use IPv4
51     server_address.sin_port = htons(PORT); // Set the port
52                                     number in network byte order
53
54     // Convert IP address from text to binary form
55     if (inet_pton(AF_INET, "127.0.0.1",
56                 &server_address.sin_addr) <= 0)
57     {
58         perror("Invalid address"); // Print error if address
59                                     conversion fails
60         exit(EXIT_FAILURE); // Exit the program
61     }
62
63     // Connect to the server
64     if (connect(socket_fd, (struct sockaddr *)&server_address,
65                 sizeof(server_address)) < 0)

```

```

56 {
57     perror("Connection failed"); // Print error if
        connection fails
58     exit(EXIT_FAILURE);          // Exit the program
59 }
60
61 printf("Connected to the server. Type 'exit' to
        disconnect.\n");
62
63 // Create a thread to handle receiving messages from the
        server
64 pthread_t thread_id;
65 if (pthread_create(&thread_id, NULL, receive_messages, (void
        *)&socket_fd) != 0)
66 {
67     perror("Thread creation failed"); // Print error if
        thread creation fails
68     return -1;
69 }
70
71 // Main loop to send messages to the server
72 while (1)
73 {
74     memset(message, 0, sizeof(message)); // Clear the
        message buffer
75     printf("You: ");
76     fgets(message, sizeof(message), stdin); // Read input
        from the user
77     message[strcspn(message, "\n")] = 0;    // Remove the
        newline character
78
79     // Check if the user wants to disconnect
80     if (strcmp(message, "exit") == 0)
81     {
82         printf("Disconnecting...\n");
83         break; // Exit the loop
84     }
85
86     // Send the message to the server
87     send(socket_fd, message, strlen(message), 0);
88 }
89
90 // Close the socket
91 close(socket_fd);
92 return 0;
93 }

```

3.1.2 Execution Screenshot

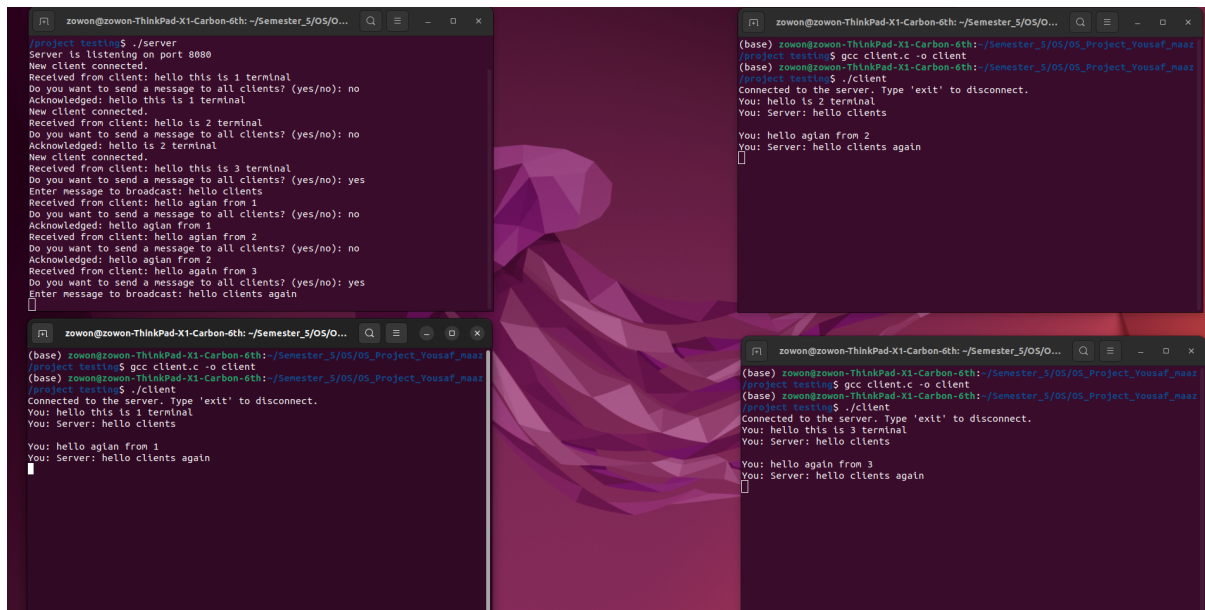


Figure 2: Server and Client Communication on Local System

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

```

1 #include <stdio.h>           // Standard input/output functions
2 #include <stdlib.h>          // Standard library functions
3 #include <string.h>           // String manipulation functions
4 #include <sys/socket.h>       // Socket programming functions
5 #include <arpa/inet.h>        // Internet address manipulation
6                               // functions
7
8 #define PORT 8080             // Port number for the server
9 #define MAX_CONNECTIONS 5    // Maximum number of pending
10                               // connections
11
12 int main()
13 {
14     int server_fd, client_socket;
15     // Server and client socket file descriptors
16     struct sockaddr_in server_address, client_address;
17     // Server and client address structures
18     int addr_len = sizeof(client_address);
19     // Length of the client address structure
20     char buffer[1024] = {0};
21     // Buffer to store messages

```

```

17 char *welcome_message = "Server: Connection established.\n";
    // Welcome message for the client
18
19 // Step 1: Create a socket
20 if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
21 {
22     perror("Socket failed"); // Print error if socket
        creation fails
23     exit(EXIT_FAILURE);      // Exit with failure
24 }
25
26 // Step 2: Configure server address structure
27 server_address.sin_family = AF_INET;          // Use IPv4
28 server_address.sin_addr.s_addr = INADDR_ANY; // Accept
        connections from any IP address
29 server_address.sin_port = htons(PORT);        // Convert port
        number to network byte order
30
31 // Step 3: Bind the socket to the specified IP and port
32 if (bind(server_fd, (struct sockaddr *)&server_address,
    sizeof(server_address)) < 0)
33 {
34     perror("Bind failed"); // Print error if binding fails
35     exit(EXIT_FAILURE);    // Exit with failure
36 }
37
38 // Step 4: Listen for incoming connections
39 if (listen(server_fd, MAX_CONNECTIONS) < 0)
40 {
41     perror("Listen failed"); // Print error if listening
        fails
42     exit(EXIT_FAILURE);      // Exit with failure
43 }
44
45 printf("Server is listening on port %d...\n", PORT);
46
47 // Step 5: Accept a client connection
48 if ((client_socket = accept(server_fd, (struct sockaddr
    *)&client_address, (socklen_t *)&addr_len)) < 0)
49 {
50     perror("Accept failed"); // Print error if accepting a
        connection fails
51     exit(EXIT_FAILURE);      // Exit with failure
52 }
53
54 // Print details of the connected client
55 printf("Client connected from IP: %s, Port: %d\n",
    inet_ntoa(client_address.sin_addr),
    ntohs(client_address.sin_port));
56
57 // Step 6: Send a welcome message to the client
58

```

```

59     send(client_socket, welcome_message,
60           strlen(welcome_message), 0);
61
62     // Step 7: Communicate with the client
63     while (1)
64     {
65         // Receive a message from the client
66         int valread = read(client_socket, buffer,
67                             sizeof(buffer));
68         if (valread > 0)
69         {
70             buffer[valread] = '\0';          // Null-terminate the
71                                             received message
72             printf("Client: %s", buffer); // Display the
73                                             client's message
74         }
75         else
76         {
77             // If no bytes are read, the client has disconnected
78             printf("Client disconnected.\n");
79             break;
80         }
81
82         // Prompt the server to enter a response
83         printf("Enter message for client: ");
84         fgets(buffer, sizeof(buffer), stdin);          // Read
85                                                         input from the server user
86         send(client_socket, buffer, strlen(buffer), 0); // Send
87                                                         the message to the client
88     }
89
90     // Step 8: Close the connections
91     close(client_socket); // Close the client socket
92     close(server_fd);     // Close the server socket
93
94     return 0; // Exit successfully
95 }

```

Listing 5: Distributed System Client Code

```

1  #include <stdio.h>          // Standard input/output functions
2  #include <stdlib.h>         // Standard library functions
3  #include <string.h>         // String manipulation functions
4  #include <sys/socket.h>     // Socket programming functions
5  #include <arpa/inet.h>      // Internet address manipulation
6                               functions
7  #include <unistd.h>         // POSIX API (e.g., close)
8
9  #define PORT 8080 // The port number to connect to the server
10
11 int main()
12 {

```



```

12  int client_socket;                // File descriptor for
    the client socket
13  struct sockaddr_in server_address; // Structure to store the
    server's address
14  char buffer[1024] = {0};         // Buffer to store
    messages
15
16  // Step 1: Create a socket
17  if ((client_socket = socket(AF_INET, SOCK_STREAM, 0)) < 0)
18  {
19      perror("Socket creation failed"); // Print error if
    socket creation fails
20      exit(EXIT_FAILURE);              // Exit with failure
21  }
22
23  // Step 2: Configure server address structure
24  server_address.sin_family = AF_INET; // Use IPv4
25  server_address.sin_port = htons(PORT); // Convert port
    number to network byte order
26
27  // Step 3: Convert server IP address to binary form
28  if (inet_pton(AF_INET, "192.168.1.55",
    &server_address.sin_addr) <= 0)
29  {
30      perror("Invalid address or address not supported"); //
    Error if the IP is invalid
31      exit(EXIT_FAILURE); //
    Exit with failure
32  }
33
34  // Step 4: Connect to the server
35  if (connect(client_socket, (struct sockaddr
    *)&server_address, sizeof(server_address)) < 0)
36  {
37      perror("Connection failed"); // Print error if
    connection fails
38      exit(EXIT_FAILURE); // Exit with failure
39  }
40
41  printf("Connected to the server.\n");
42
43  // Step 5: Receive the welcome message from the server
44  int valread = read(client_socket, buffer, sizeof(buffer));
    // Read the welcome message
45  if (valread > 0)
46  {
47      buffer[valread] = '\0'; // Null-terminate the received
    message
48      printf("%s", buffer); // Print the welcome message
49  }
50

```

```

51 // Step 6: Communicate with the server
52 while (1)
53 {
54     // Send a message to the server
55     printf("Enter message for server: ");
56     fgets(buffer, sizeof(buffer), stdin);           // Read
57     // input from the user
58     send(client_socket, buffer, strlen(buffer), 0); // Send
59     // the message to the server
60
61     // Receive a response from the server
62     valread = read(client_socket, buffer, sizeof(buffer));
63     // Read the server's response
64     if (valread > 0)
65     {
66         buffer[valread] = '\0';           // Null-terminate the
67         // received message
68         printf("Server: %s", buffer); // Print the server's
69         // response
70     }
71     else
72     {
73         // If no bytes are read, the server has disconnected
74         printf("Server disconnected.\n");
75         break;
76     }
77 }
78
79 // Step 7: Close the connection
80 close(client_socket); // Close the client socket
81
82 return 0; // Exit successfully
83 }

```

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.

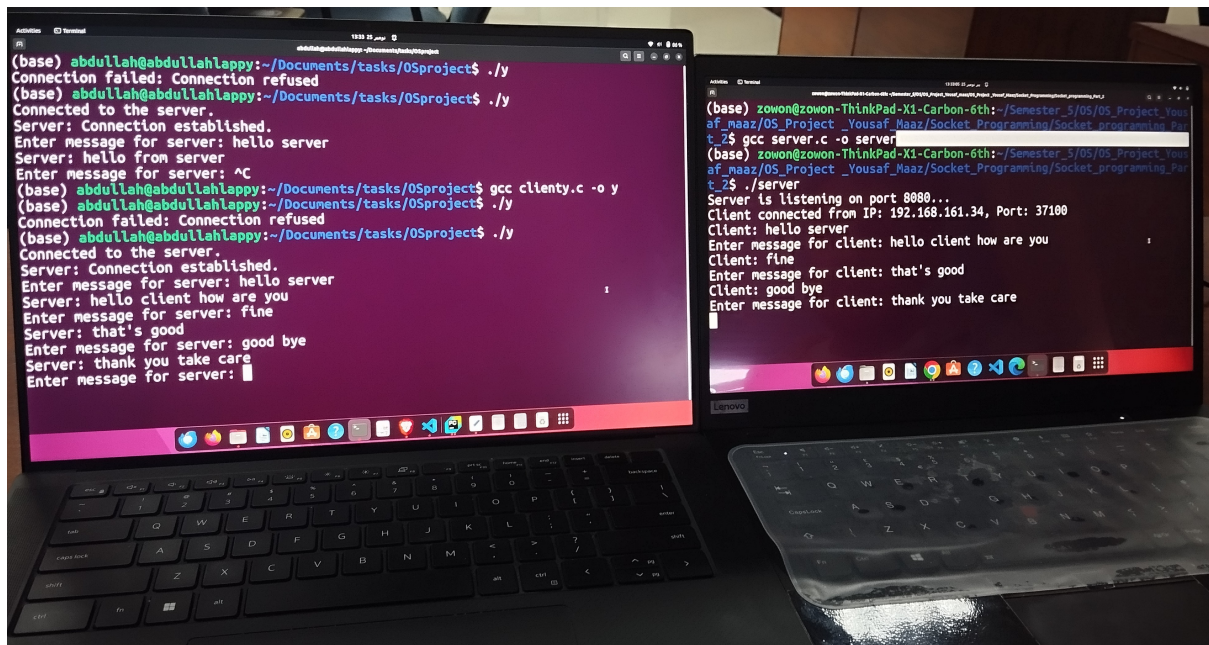


Figure 3: Server and Client Communication in Distributed System