FAST University Peshawar
Department of Computer Science

OS Project

Name: Tazmeen Afroz
Roll No: 22P-9252
Course: Operating Systems
Lecturer: Engr. M Usman Malik
Due Date: 25 November 2024

Task 1: CPU Scheduling Implementation

Task List Example

```
T1, 4, 20
T2, 2, 25
T3, 3, 25
T4, 3, 15
T5, 10, 10
```

Code Implementation

Main Structure:

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

#define tasks_n 10
#define time_quantum 5


typedef struct {
    char name[10];
    int priority;
    int cpuBurst;
    int remainingTime;
    int waitingTime;
    int turnaroundTime;
    int completed;
} Task;

// Function to print horizontal line
void printLine(int width) {
    for(int i = 0; i < width; i++) printf("-");
    printf("\n");
}

// Function to print formatted table
void printScheduleTable(Task tasks[], int n, const char* algorithm) {
    printf("\n%s Scheduling Results:\n", algorithm);
    printLine(75);
    printf("| %-8s | %-8s | %-12s | %-15s | %-15s |\n",
           "Task", "Priority", "Burst Time", "Waiting Time", "Turnaround Time");
    printLine(75);

    float avgWait = 0, avgTurnaround = 0;
```

```
35
36      for (int i = 0; i < n; i++) {
37          printf("| %-8s | %-8d | %-12d | %-15d | %-15d |\n",
38                 tasks[i].name,
39                 tasks[i].priority,
40                 tasks[i].cpuBurst,
41                 tasks[i].waitingTime,
42                 tasks[i].turnaroundTime);
43          avgWait += tasks[i].waitingTime;
44          avgTurnaround += tasks[i].turnaroundTime;
45      }
46
47      printLine(75);
48      printf("Average Waiting Time: %.2f\n", avgWait/n);
49      printf("Average Turnaround Time: %.2f\n", avgTurnaround/n);
50  }
51
52  // Function to load tasks from file
53  int loadTasks(Task tasks[], const char* filename) {
54      FILE* file = fopen(filename, "r");
55      if (file == NULL) {
56          printf("Error opening file!\n");
57          return 0;
58      }
59
60      int n = 0;
61      while (fscanf(file, "%[^,], %d, %d\n",
62             tasks[n].name, &tasks[n].priority, &tasks[n].cpuBurst) == 3) {
63          tasks[n].remainingTime = tasks[n].cpuBurst;
64          tasks[n].waitingTime = 0;
65          tasks[n].turnaroundTime = 0;
66          tasks[n].completed = 0;
67          n++;
68      }
69
70      fclose(file);
71      return n;
72  }
73
74  void printGanttChart(Task tasks[], int n, const char* algorithm) {
75      printf("\nGantt Chart for %s:\n", algorithm);
76
77
78      // Print the task execution bars
79      printf("|");
80      for(int i = 0; i < n; i++) {
81          for(int j = 0; j < tasks[i].cpuBurst; j++) {
82              printf("-");
83          }
84          printf("|");
85      }
86      printf("\n");
87
88      // Print the task names
89      printf("|");
90      for(int i = 0; i < n; i++) {
91          int spaces = tasks[i].cpuBurst/2;
92          for(int j = 0; j < spaces - (strlen(tasks[i].name)/2); j++) printf(" ");
93          printf("%s", tasks[i].name);
94          for(int j = 0; j < tasks[i].cpuBurst - spaces - (strlen(tasks[i].name) - strlen(
    tasks[i].name)/2); j++) printf(" ");
95          printf("|");
96      }
97      printf("\n");
98
99      // Print the time markers
100     printf("0");
101     int currentTime = 0;
```

```
    for(int i = 0; i < n; i++) {
        currentTime += tasks[i].cpuBurst;
        for(int j = 0; j < tasks[i].cpuBurst-1; j++) printf(" ");
        printf("%d", currentTime);
    }
    printf("\n");

}

// Modify each scheduling function to include the Gantt chart. Here's how to update them:

void fcfs(Task tasks[], int n) {
    Task tempTasks[tasks_n];
    memcpy(tempTasks, tasks, sizeof(Task) * n);

    int currentTime = 0;
    for (int i = 0; i < n; i++) {
        tempTasks[i].waitingTime = currentTime;
        tempTasks[i].turnaroundTime = currentTime + tempTasks[i].cpuBurst;
        currentTime += tempTasks[i].cpuBurst;
    }

    printScheduleTable(tempTasks, n, "FCFS");
    printGanttChart(tempTasks, n, "FCFS");
}

void sjf(Task tasks[], int n) {
    Task tempTasks[tasks_n];
    memcpy(tempTasks, tasks, sizeof(Task) * n);

    // Sort by CPU burst time
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (tempTasks[j].cpuBurst > tempTasks[j+1].cpuBurst) {
                Task temp = tempTasks[j];
                tempTasks[j] = tempTasks[j+1];
                tempTasks[j+1] = temp;
            }
        }
    }

    int currentTime = 0;
    for (int i = 0; i < n; i++) {
        tempTasks[i].waitingTime = currentTime;
        tempTasks[i].turnaroundTime = currentTime + tempTasks[i].cpuBurst;
        currentTime += tempTasks[i].cpuBurst;
    }

    printScheduleTable(tempTasks, n, "SJF");
    printGanttChart(tempTasks, n, "SJF");
}

void priorityScheduling(Task tasks[], int n) {
    Task tempTasks[tasks_n];
    memcpy(tempTasks, tasks, sizeof(Task) * n);

    // Sort by priority
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (tempTasks[j].priority > tempTasks[j+1].priority) {
                Task temp = tempTasks[j];
                tempTasks[j] = tempTasks[j+1];
                tempTasks[j+1] = temp;
            }
        }
    }

    int currentTime = 0;
```

```
    for (int i = 0; i < n; i++) {
        tempTasks[i].waitingTime = currentTime;
        tempTasks[i].turnaroundTime = currentTime + tempTasks[i].cpuBurst;
        currentTime += tempTasks[i].cpuBurst;
    }

    printScheduleTable(tempTasks, n, "Priority");
    printGanttChart(tempTasks, n, "Priority");
}

void roundRobin(Task tasks[], int n, int timeQuantum) {
    Task tempTasks[tasks_n];
    memcpy(tempTasks, tasks, sizeof(Task) * n);

    // Array to store execution order for Gantt chart
    Task executionOrder[100];  // Assuming max 100 time slices
    int executionCount = 0;

    int remainingTasks = n;
    int currentTime = 0;

    // Initialize remaining time
    for (int i = 0; i < n; i++) {
        tempTasks[i].remainingTime = tempTasks[i].cpuBurst;
    }

    while (remainingTasks > 0) {
        for (int i = 0; i < n; i++) {
            if (tempTasks[i].remainingTime > 0) {
                int executeTime = (tempTasks[i].remainingTime > timeQuantum) ?
                                  timeQuantum : tempTasks[i].remainingTime;

                // Store execution step for Gantt chart
                executionOrder[executionCount] = tempTasks[i];
                executionOrder[executionCount].cpuBurst = executeTime;
                executionCount++;

                tempTasks[i].remainingTime -= executeTime;
                currentTime += executeTime;

                if (tempTasks[i].remainingTime == 0) {
                    remainingTasks--;
                    tempTasks[i].turnaroundTime = currentTime;
                    tempTasks[i].waitingTime = tempTasks[i].turnaroundTime - tempTasks[i].
    cpuBurst;
                }
            }
        }
    }

    printScheduleTable(tempTasks, n, "Round Robin");
    printGanttChart(executionOrder, executionCount, "Round Robin");
}

void priorityRoundRobin(Task tasks[], int n, int timeQuantum) {
    Task tempTasks[tasks_n];
    memcpy(tempTasks, tasks, sizeof(Task) * n);

    // Array to store execution order for Gantt chart
    Task executionOrder[100];  // Assuming max 100 time slices
    int executionCount = 0;

    int remainingTasks = n;
    int currentTime = 0;

    // Initialize remaining time
    for (int i = 0; i < n; i++) {
        tempTasks[i].remainingTime = tempTasks[i].cpuBurst;
```

```
    }

    while (remainingTasks > 0) {
        int highestPriority = INT_MAX;
        for (int i = 0; i < n; i++) {
            if (tempTasks[i].remainingTime > 0 && tempTasks[i].priority < highestPriority) {
                highestPriority = tempTasks[i].priority;
            }
        }

        int taskExecuted = 0;
        for (int i = 0; i < n; i++) {
            if (tempTasks[i].remainingTime > 0 && tempTasks[i].priority == highestPriority)
    {
                int executeTime = (tempTasks[i].remainingTime > timeQuantum) ?
                                  timeQuantum : tempTasks[i].remainingTime;

                // Store execution step for Gantt chart
                executionOrder[executionCount] = tempTasks[i];
                executionOrder[executionCount].cpuBurst = executeTime;
                executionCount++;

                tempTasks[i].remainingTime -= executeTime;
                currentTime += executeTime;
                taskExecuted = 1;

                if (tempTasks[i].remainingTime == 0) {
                    remainingTasks--;
                    tempTasks[i].turnaroundTime = currentTime;
                    tempTasks[i].waitingTime = tempTasks[i].turnaroundTime - tempTasks[i].
    cpuBurst;
                }
            }
        }

        if (!taskExecuted) break;
    }

    printScheduleTable(tempTasks, n, "Priority Round Robin");
    printGanttChart(executionOrder, executionCount, "Priority Round Robin");
}

int main() {
    Task tasks[tasks_n];
    int n = loadTasks(tasks, "schedule.txt");

    if (n == 0) {
        printf("No tasks loaded!\n");
        return 1;
    }

    printf("CPU Scheduling Simulator\n");
    printLine(50);
    printf("Loaded %d tasks.\n\n", n);

    // Execute all scheduling algorithms
    fcfs(tasks, n);
    sjf(tasks, n);
    priorityScheduling(tasks, n);
    roundRobin(tasks, n, time_quantum);
    priorityRoundRobin(tasks, n, time_quantum);

    return 0;
}
```

Listing 1: Scheduling Code

```
FCFS Scheduling Results:
------------------------------------------------------------------------
| Task     | Priority | Burst Time   | Waiting Time    | Turnaround Time |
------------------------------------------------------------------------
| T1       | 4        | 20           | 0               | 20              |
| T2       | 2        | 25           | 20              | 45              |
| T3       | 3        | 25           | 45              | 70              |
| T4       | 3        | 15           | 70              | 85              |
| T5       | 10       | 10           | 85              | 95              |
------------------------------------------------------------------------
Average Waiting Time: 44.00
Average Turnaround Time: 63.00

Gantt Chart for FCFS:
|--------------------|------------------------|------------------------|--------------|----------|
|         T1         |           T2           |           T3           |      T4      |    T5    |
0                   20                       45                       70             85         95
```

Figure 1: FCFS

```
SJF Scheduling Results:
------------------------------------------------------------------------
| Task     | Priority | Burst Time   | Waiting Time    | Turnaround Time |
------------------------------------------------------------------------
| T5       | 10       | 10           | 0               | 10              |
| T4       | 3        | 15           | 10              | 25              |
| T1       | 4        | 20           | 25              | 45              |
| T2       | 2        | 25           | 45              | 70              |
| T3       | 3        | 25           | 70              | 95              |
------------------------------------------------------------------------
Average Waiting Time: 30.00
Average Turnaround Time: 49.00

Gantt Chart for SJF:
|----------|--------------|--------------------|------------------------|------------------------|
|    T5    |      T4      |         T1         |           T2           |           T3           |
0         10             25                   45                       70                       95
```

Figure 2: SJF

```
Priority Scheduling Results:
------------------------------------------------------------------------
| Task     | Priority | Burst Time   | Waiting Time    | Turnaround Time |
------------------------------------------------------------------------
| T2       | 2        | 25           | 0               | 25              |
| T3       | 3        | 25           | 25              | 50              |
| T4       | 3        | 15           | 50              | 65              |
| T1       | 4        | 20           | 65              | 85              |
| T5       | 10       | 10           | 85              | 95              |
------------------------------------------------------------------------
Average Waiting Time: 45.00
Average Turnaround Time: 64.00

Gantt Chart for Priority:
|------------------------|------------------------|--------------|--------------------|----------|
|           T2           |           T3           |      T4      |         T1         |    T5    |
0                       25                       50             65                   85         95
```

Figure 3: Priority

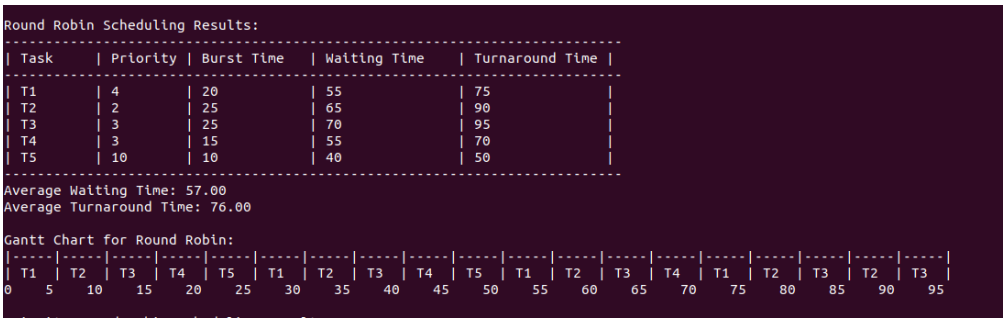

Figure 4: Round Robin

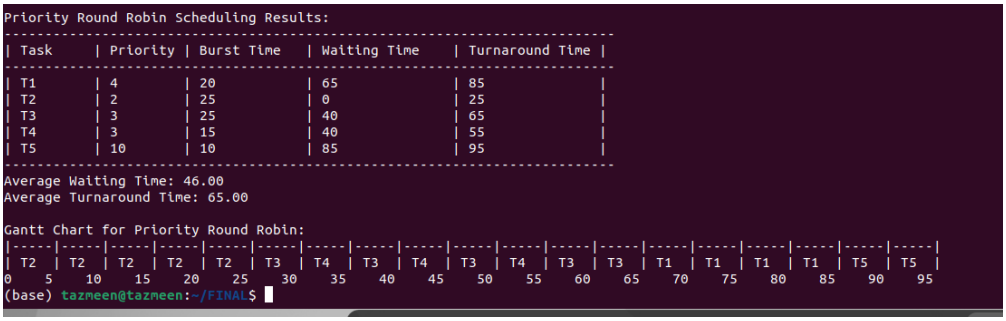

Figure 5: Priority Round Robin

Task 2: Socket Programming Implementation

Part 1: Local System Socket Programming

Code Implementation

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>
#include <semaphore.h>
#include <errno.h>
#include <signal.h>
#include <time.h>

#define PORT 8080
#define MAX_CLIENTS 4
#define SERVER_PREFIX "SERVER: "
#define LOG_FILE "server_log.txt"
#define MAX_MESSAGE_LENGTH 1024

typedef struct {
    int socket;
    int id;
    char messages[100][1024];
    int msg_count;
    int active;
} Client;
```

```
26
27 // Global variables
28 Client clients[MAX_CLIENTS];
29 int client_count = 0;
30 sem_t server_semaphore;
31 int server_running = 1;
32
33 void log_error(const char *message) {
34     FILE *log_file = fopen(LOG_FILE, "a");
35     if (log_file == NULL) {
36         perror("Failed to open log file");
37         return;
38     }
39     fprintf(log_file, "ERROR: %s: %s\n", message, strerror(errno));
40     fclose(log_file);
41 }
42
43 void broadcast_message(const char *message) {
44     char broadcast_message[1050];
45     snprintf(broadcast_message, sizeof(broadcast_message), "%s%s", SERVER_PREFIX, message);
46
47     sem_wait(&server_semaphore);  // Lock with semaphore
48     for (int i = 0; i < client_count; i++) {
49         if (clients[i].active) {
50             if (send(clients[i].socket, broadcast_message, strlen(broadcast_message), 0) <
    0) {
51                 perror("Send failed");
52                 log_error("Send failed");
53             }
54         }
55     }
56     sem_post(&server_semaphore);  // Unlock with semaphore
57 }
58
59 void store_client_message(int client_id, const char *message) {
60     sem_wait(&server_semaphore);
61     int msg_idx = clients[client_id].msg_count % 100;
62     strncpy(clients[client_id].messages[msg_idx], message, 1024);
63     clients[client_id].msg_count++;
64     sem_post(&server_semaphore);
65 }
66
67 void show_menu() {
68     printf("\n=== Server Menu ===\n");
69     printf("1. Send Broadcast Message\n");
70     printf("2. View Individual Client Messages\n");
71     printf("3. Exit\n");
72     printf("Enter choice : ");
73 }
74
75 void *handle_client(void *client_data) {
76     Client *client = (Client*)client_data;
77     int read_size;
78     char client_message[1024];
79     char display_message[1100];
80
81     printf("\nClient %d connected\n", client->id);
82
83     while (server_running && (read_size = recv(client->socket, client_message, 1024, 0)) >
    0) {
84         client_message[read_size] = '\0';
85
86         snprintf(display_message, sizeof(display_message), "Client %d: %s", client->id,
    client_message);
87         store_client_message(client->id, display_message);
88
89         printf("\n%s\n", display_message);
90     }
```

```
91
92      sem_wait(&server_semaphore);
93      client->active = 0;
94      printf("\nClient %d disconnected\n", client->id);
95      sem_post(&server_semaphore);
96
97      close(client->socket);
98      return 0;
99  }
100
101 void view_individual_messages() {
102     int client_id;
103     printf("\nActive clients:\n");
104
105     sem_wait(&server_semaphore);
106     for (int i = 0; i < client_count; i++) {
107         if (clients[i].active) {
108             printf("Client %d\n", i);
109         }
110     }
111     sem_post(&server_semaphore);
112
113     printf("Enter client ID to view messages: ");
114     scanf("%d", &client_id);
115     getchar(); // Clear newline
116
117     if (client_id >= 0 && client_id < client_count) {
118         printf("\nMessages from Client %d:\n", client_id);
119         sem_wait(&server_semaphore);
120         int start = (clients[client_id].msg_count > 100) ?
121                     clients[client_id].msg_count - 100 : 0;
122         for (int i = start; i < clients[client_id].msg_count; i++) {
123             printf("%s\n", clients[client_id].messages[i % 100]);
124         }
125         sem_post(&server_semaphore);
126     } else {
127         printf("Invalid client ID\n");
128     }
129 }
130
131 void *server_menu(void *arg) {
132     int choice;
133     char input[1024];
134     char broadcast_msg[1024];
135
136     show_menu();
137
138     while (server_running) {
139         fgets(input, sizeof(input), stdin);
140         choice = atoi(input);
141
142         switch (choice) {
143             case 1:
144                 printf("Enter message to broadcast: ");
145                 fgets(broadcast_msg, sizeof(broadcast_msg), stdin);
146                 broadcast_msg[strcspn(broadcast_msg, "\n")] = '\0';
147                 broadcast_message(broadcast_msg);
148                 printf("Message broadcasted\n");
149                 break;
150
151             case 2:
152                 view_individual_messages();
153                 break;
154
155             case 3:
156                 printf("\nShutting down server...\n");
157                 server_running = 0;
158                 sem_wait(&server_semaphore);
```

```
                for (int i = 0; i < client_count; i++) {
                    if (clients[i].active) {
                        send(clients[i].socket, "SERVER_SHUTDOWN", 14, 0);
                        close(clients[i].socket);
                    }
                }
                sem_post(&server_semaphore);
                exit(0);
                break;

            default:
                printf("Invalid choice!\n");
                show_menu();
        }
    }
    return 0;
}

int main() {
    int server_fd, new_socket, c;
    struct sockaddr_in server, client;
    pthread_t thread_id, menu_thread;

    // Initialize semaphore
    if (sem_init(&server_semaphore, 0, 1) < 0) {
        perror("Semaphore initialization failed");
        return 1;
    }

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd == -1) {
        perror("Could not create socket");
        log_error("Could not create socket");
        return 1;
    }

    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(PORT);

    if (bind(server_fd, (struct sockaddr *)&server, sizeof(server)) < 0) {
        perror("bind failed");
        log_error("bind failed");
        return 1;
    }

    listen(server_fd, 3);
    printf("Server started on port %d\n", PORT);
    printf("Showing all client activity - Use menu for additional options\n");

    if (pthread_create(&menu_thread, NULL, server_menu, NULL) < 0) {
        perror("could not create menu thread");
        log_error("could not create menu thread");
        return 1;
    }

    c = sizeof(struct sockaddr_in);
    while (server_running && (new_socket = accept(server_fd, (struct sockaddr *)&client, (
    socklen_t*)&c))) {
        if (new_socket < 0) {
            perror("accept failed");
            log_error("accept failed");
            continue;
        }

        sem_wait(&server_semaphore);
        if (client_count >= MAX_CLIENTS) {
            send(new_socket, "Server is full", 13, 0);
```

```
            close(new_socket);
            sem_post(&server_semaphore);
            continue;
        }

        clients[client_count].socket = new_socket;
        clients[client_count].id = client_count;
        clients[client_count].active = 1;
        clients[client_count].msg_count = 0;

        if (pthread_create(&thread_id, NULL, handle_client, (void*)&clients[client_count]) <
 0) {
            perror("could not create thread");
            log_error("could not create thread");
            sem_post(&server_semaphore);
            return 1;
        }

        client_count++;
        sem_post(&server_semaphore);
    }

    sem_destroy(&server_semaphore);
    return 0;
}
```

Listing 2: server code - Socket Programming Code

Client code

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

#define PORT 8080
#define LOG_FILE "client_log.txt"

int client_running = 1;
sem_t client_semaphore;

void log_error(const char *message) {
    FILE *log_file = fopen(LOG_FILE, "a");
    if (log_file == NULL) {
        perror("Failed to open log file");
        return;
    }
    fprintf(log_file, "ERROR: %s: %s\n", message, strerror(errno));
    fclose(log_file);
}

void show_menu() {
    printf("\n=== Client Menu ===\n");
    printf("1. Send Message\n");
    printf("2. Exit\n");
    printf("Enter choice: ");
}

void *receive_messages(void *socket_desc) {
    int sock = *(int*)socket_desc;
    char server_message[1024];
    int read_size;
```

```
38
39      while (client_running && (read_size = recv(sock, server_message, 1024, 0)) > 0) {
40          server_message[read_size] = '\0';
41
42          sem_wait(&client_semaphore);  // Protect shared resource access
43          if (strcmp(server_message, "SERVER_SHUTDOWN") == 0) {
44              printf("\nServer is shutting down. Press Enter to exit...\n");
45              client_running = 0;
46              sem_post(&client_semaphore);
47              break;
48          }
49
50          if (strncmp(server_message, "SERVER:", 7) == 0) {
51              printf("\n%s\n", server_message);
52          }
53
54          if (client_running) {
55              fflush(stdout);
56          }
57          sem_post(&client_semaphore);  // Release semaphore
58      }
59
60      sem_wait(&client_semaphore);
61      if (read_size == 0 && client_running) {
62          printf("\nServer disconnected\n");
63          client_running = 0;
64      } else if (read_size == -1 && client_running) {
65          perror("recv failed");
66          log_error("recv failed");
67          client_running = 0;
68      }
69      sem_post(&client_semaphore);
70
71      return 0;
72  }
73
74  int main() {
75      int sock;
76      struct sockaddr_in server;
77      char message[1024];
78      pthread_t thread_id;
79      int choice;
80      char input[10];
81
82      // Initialize semaphore
83      if (sem_init(&client_semaphore, 0, 1) < 0) {
84          perror("Semaphore initialization failed");
85          return 1;
86      }
87
88      sock = socket(AF_INET, SOCK_STREAM, 0);
89      if (sock == -1) {
90          perror("Could not create socket");
91          log_error("Could not create socket");
92          return 1;
93      }
94
95      server.sin_addr.s_addr = inet_addr("127.0.0.1");
96      server.sin_family = AF_INET;
97      server.sin_port = htons(PORT);
98
99      if (connect(sock, (struct sockaddr *)&server, sizeof(server)) < 0) {
100         perror("connect failed");
101         log_error("connect failed");
102         return 1;
103     }
104     printf("Connected to server\n");
105
```

```
    if (pthread_create(&thread_id, NULL, receive_messages, (void*)&sock) < 0) {
        perror("could not create thread");
        log_error("could not create thread");
        return 1;
    }

    while (client_running) {
        show_menu();
        fgets(input, sizeof(input), stdin);
        choice = atoi(input);

        sem_wait(&client_semaphore);  // Protect shared resource access
        switch (choice) {
            case 1:
                printf("Enter message: ");
                fgets(message, 1024, stdin);
                message[strcspn(message, "\n")] = '\0';

                if (send(sock, message, strlen(message), 0) < 0) {
                    perror("Send failed");
                    log_error("Send failed");
                    client_running = 0;
                } else {
                    printf("Message sent to server\n");
                }
                break;

            case 2:
                printf("Exiting...\n");
                client_running = 0;
                break;

            default:
                printf("Invalid choice!\n");
        }
        sem_post(&client_semaphore);  // Release semaphore
    }

    sem_destroy(&client_semaphore);
    close(sock);
    return 0;
}
```

Listing 3: client code - Socket Programming Code

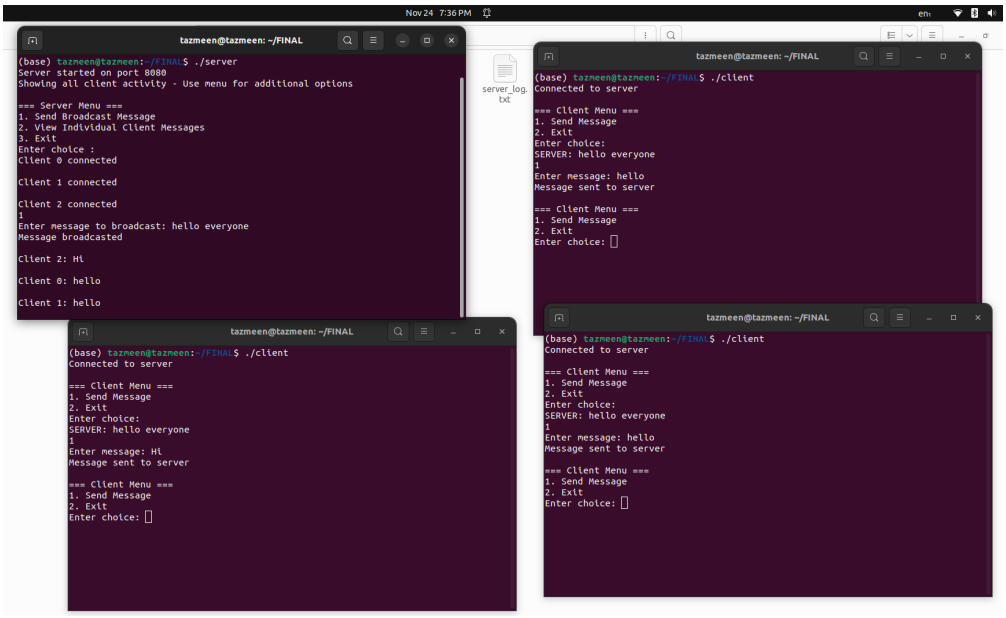

Figure 6: local socket programming

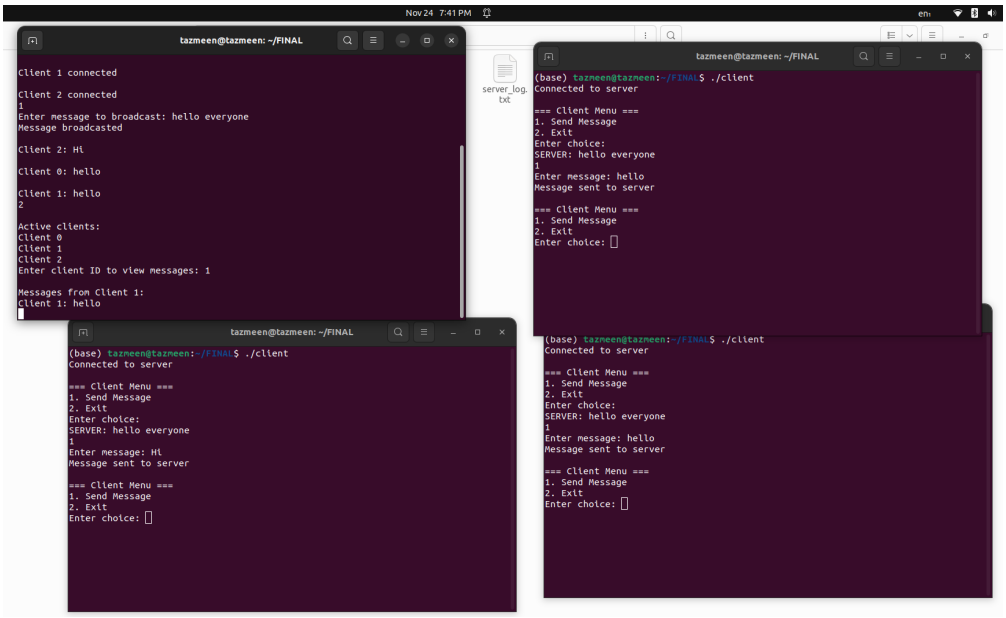

Figure 7: local socket programming

Part 2: Distributed System Socket Programming

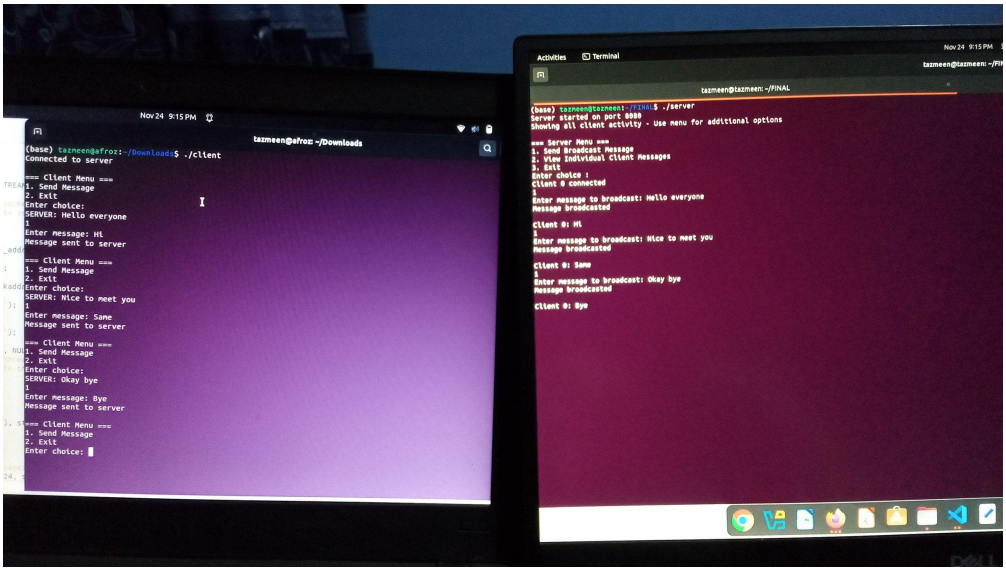

Figure 8: distributed system socket programming