

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:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <limits.h>
5
6 #define tasks_n 10
7 #define time_quantum 5
8
9
10 typedef struct {
11     char name[10];
12     int priority;
13     int cpuBurst;
14     int remainingTime;
15     int waitingTime;
16     int turnaroundTime;
17     int completed;
18 } Task;
19
20 // Function to print horizontal line
21 void printLine(int width) {
22     for(int i = 0; i < width; i++) printf("-");
23     printf("\n");
24 }
25
26 // Function to print formatted table
27 void printScheduleTable(Task tasks[], int n, const char* algorithm) {
28     printf("\n%s Scheduling Results:\n", algorithm);
29     printLine(75);
30     printf("| %-8s | %-8s | %-12s | %-15s | %-15s |\n",
31           "Task", "Priority", "Burst Time", "Waiting Time", "Turnaround Time");
32     printLine(75);
33
34     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(
95             tasks[i].name)/2); j++) printf(" ");
96         printf("|");
97     }
98     printf("\n");
99
100     // Print the time markers
101     printf("0");
102     int currentTime = 0;

```

```

102     for(int i = 0; i < n; i++) {
103         currentTime += tasks[i].cpuBurst;
104         for(int j = 0; j < tasks[i].cpuBurst-1; j++) printf(" ");
105         printf("%d", currentTime);
106     }
107     printf("\n");
108 }
109
110
111
112
113 void fcfs(Task tasks[], int n) {
114     Task tempTasks[tasks_n];
115     memcpy(tempTasks, tasks, sizeof(Task) * n);
116
117     int currentTime = 0;
118     for (int i = 0; i < n; i++) {
119         tempTasks[i].waitingTime = currentTime;
120         tempTasks[i].turnaroundTime = currentTime + tempTasks[i].cpuBurst;
121         currentTime += tempTasks[i].cpuBurst;
122     }
123
124     printScheduleTable(tempTasks, n, "FCFS");
125     printGanttChart(tempTasks, n, "FCFS");
126 }
127
128 void sjf(Task tasks[], int n) {
129     Task tempTasks[tasks_n];
130     memcpy(tempTasks, tasks, sizeof(Task) * n);
131
132     // Sort by CPU burst time
133     for (int i = 0; i < n-1; i++) {
134         for (int j = 0; j < n-i-1; j++) {
135             if (tempTasks[j].cpuBurst > tempTasks[j+1].cpuBurst) {
136                 Task temp = tempTasks[j];
137                 tempTasks[j] = tempTasks[j+1];
138                 tempTasks[j+1] = temp;
139             }
140         }
141     }
142
143     int currentTime = 0;
144     for (int i = 0; i < n; i++) {
145         tempTasks[i].waitingTime = currentTime;
146         tempTasks[i].turnaroundTime = currentTime + tempTasks[i].cpuBurst;
147         currentTime += tempTasks[i].cpuBurst;
148     }
149
150     printScheduleTable(tempTasks, n, "SJF");
151     printGanttChart(tempTasks, n, "SJF");
152 }
153
154 void priorityScheduling(Task tasks[], int n) {
155     Task tempTasks[tasks_n];
156     memcpy(tempTasks, tasks, sizeof(Task) * n);
157
158     // Sort by priority
159     for (int i = 0; i < n-1; i++) {
160         for (int j = 0; j < n-i-1; j++) {
161             if (tempTasks[j].priority > tempTasks[j+1].priority) {
162                 Task temp = tempTasks[j];
163                 tempTasks[j] = tempTasks[j+1];
164                 tempTasks[j+1] = temp;
165             }
166         }
167     }
168
169     int currentTime = 0;

```

```

170     for (int i = 0; i < n; i++) {
171         tempTasks[i].waitingTime = currentTime;
172         tempTasks[i].turnaroundTime = currentTime + tempTasks[i].cpuBurst;
173         currentTime += tempTasks[i].cpuBurst;
174     }
175
176     printScheduleTable(tempTasks, n, "Priority");
177     printGanttChart(tempTasks, n, "Priority");
178 }
179
180 void roundRobin(Task tasks[], int n, int timeQuantum) {
181     Task tempTasks[task_n];
182     memcpy(tempTasks, tasks, sizeof(Task) * n);
183
184
185     Task executionOrder[100];
186     int executionCount = 0;
187
188     int remainingTasks = n;
189     int currentTime = 0;
190
191     // Initialize remaining time
192     for (int i = 0; i < n; i++) {
193         tempTasks[i].remainingTime = tempTasks[i].cpuBurst;
194     }
195
196     while (remainingTasks > 0) {
197         for (int i = 0; i < n; i++) {
198             if (tempTasks[i].remainingTime > 0) {
199                 int executeTime = (tempTasks[i].remainingTime > timeQuantum) ?
200                     timeQuantum : tempTasks[i].remainingTime;
201
202                 // Store execution step for Gantt chart
203                 executionOrder[executionCount] = tempTasks[i];
204                 executionOrder[executionCount].cpuBurst = executeTime;
205                 executionCount++;
206
207                 tempTasks[i].remainingTime -= executeTime;
208                 currentTime += executeTime;
209
210                 if (tempTasks[i].remainingTime == 0) {
211                     remainingTasks--;
212                     tempTasks[i].turnaroundTime = currentTime;
213                     tempTasks[i].waitingTime = tempTasks[i].turnaroundTime - tempTasks[i].
214                         cpuBurst;
215                 }
216             }
217         }
218
219         printScheduleTable(tempTasks, n, "Round Robin");
220         printGanttChart(executionOrder, executionCount, "Round Robin");
221     }
222
223 void priorityRoundRobin(Task tasks[], int n, int timeQuantum) {
224     Task tempTasks[task_n];
225     memcpy(tempTasks, tasks, sizeof(Task) * n);
226
227
228     Task executionOrder[100];
229     int executionCount = 0;
230
231     int remainingTasks = n;
232     int currentTime = 0;
233
234     // Initialize remaining time
235     for (int i = 0; i < n; i++) {
236         tempTasks[i].remainingTime = tempTasks[i].cpuBurst;

```

```

237 }
238
239 while (remainingTasks > 0) {
240     int highestPriority = INT_MAX;
241     for (int i = 0; i < n; i++) {
242         if (tempTasks[i].remainingTime > 0 && tempTasks[i].priority < highestPriority) {
243             highestPriority = tempTasks[i].priority;
244         }
245     }
246
247     int taskExecuted = 0;
248     for (int i = 0; i < n; i++) {
249         if (tempTasks[i].remainingTime > 0 && tempTasks[i].priority == highestPriority)
250     {
251         int executeTime = (tempTasks[i].remainingTime > timeQuantum) ?
252             timeQuantum : tempTasks[i].remainingTime;
253
254         executionOrder[executionCount] = tempTasks[i];
255         executionOrder[executionCount].cpuBurst = executeTime;
256         executionCount++;
257
258         tempTasks[i].remainingTime -= executeTime;
259         currentTime += executeTime;
260         taskExecuted = 1;
261
262         if (tempTasks[i].remainingTime == 0) {
263             remainingTasks--;
264             tempTasks[i].turnaroundTime = currentTime;
265             tempTasks[i].waitingTime = tempTasks[i].turnaroundTime - tempTasks[i].
266             cpuBurst;
267         }
268     }
269
270     if (!taskExecuted) break;
271 }
272
273 printScheduleTable(tempTasks, n, "Priority Round Robin");
274 printGanttChart(executionOrder, executionCount, "Priority Round Robin");
275 }
276
277 int main() {
278     Task tasks[tasks_n];
279     int n = loadTasks(tasks, "schedule.txt");
280
281     if (n == 0) {
282         printf("No tasks loaded!\n");
283         return 1;
284     }
285
286     printf("CPU Scheduling Simulator\n");
287     printLine(50);
288     printf("Loaded %d tasks.\n\n", n);
289
290     // Execute all scheduling algorithms
291     fcfs(tasks, n);
292     sjf(tasks, n);
293     priorityScheduling(tasks, n);
294     roundRobin(tasks, n, time_quantum);
295     priorityRoundRobin(tasks, n, time_quantum);
296
297     return 0;
298 }

```

Listing 1: Scheduling Code

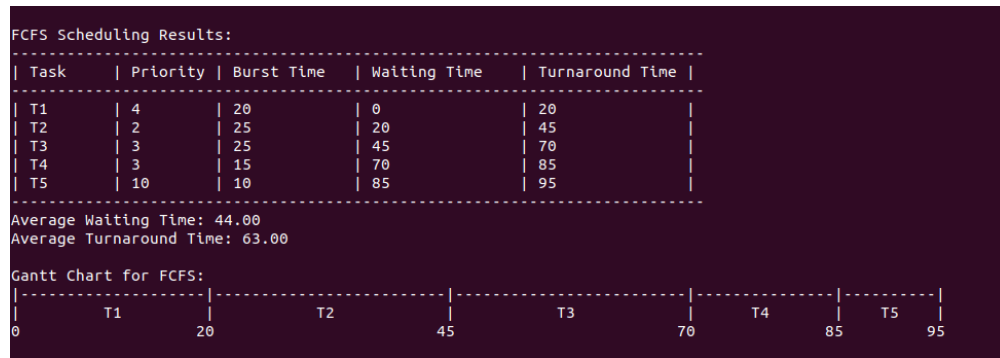


Figure 1: FCFS

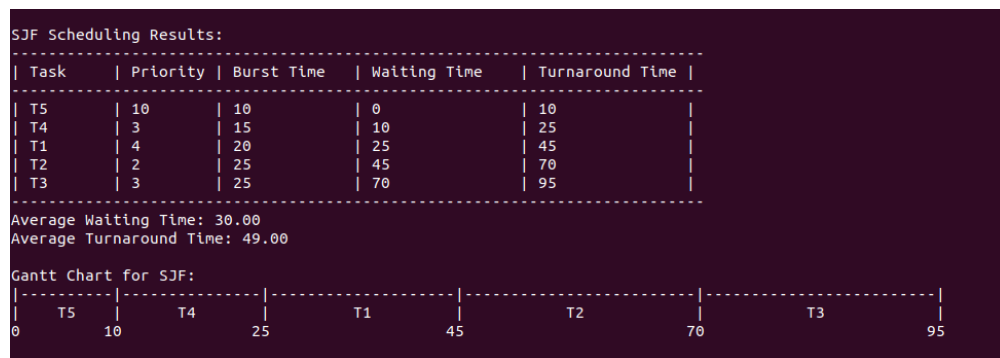


Figure 2: SJF

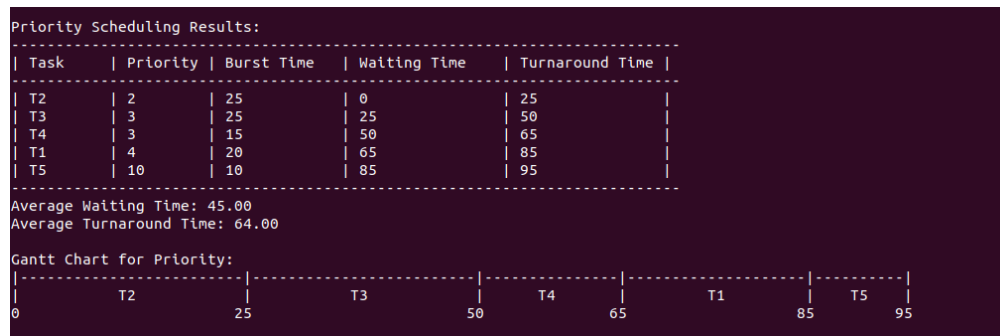


Figure 3: Priority

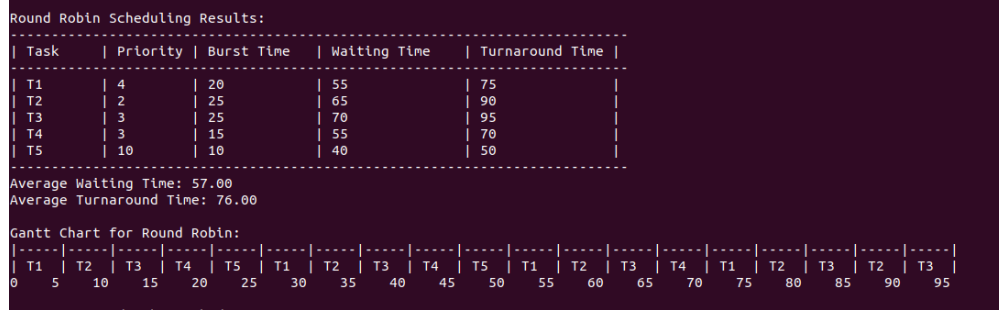


Figure 4: Round Robin

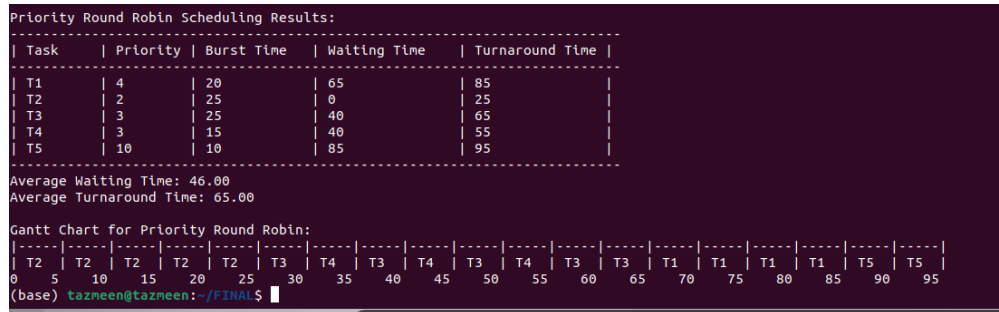


Figure 5: Priority Round Robin

Task 2: Socket Programming Implementation

Part 1: Local System Socket Programming

Code Implementation

Server code

```

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  #include <semaphore.h>
9  #include <errno.h>
10 #include <signal.h>
11 #include <time.h>
12
13 #define PORT 8080
14 #define MAX_CLIENTS 4
15 #define SERVER_PREFIX "SERVER: "
16 #define LOG_FILE "server_log.txt"
17 #define MAX_MESSAGE_LENGTH 1024
18
19 typedef struct {
20     int socket;
21     int id;
22     char messages[100][1024];
23     int msg_count;
24     int active;
25 } 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) <
51                 0) {
52                 perror("Send failed");
53                 log_error("Send failed");
54             }
55         }
56     }
57     sem_post(&server_semaphore); // Unlock with semaphore
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)) >
84         0) {
85         client_message[read_size] = '\0';
86
87         snprintf(display_message, sizeof(display_message), "Client %d: %s", client->id,
88             client_message);
89         store_client_message(client->id, display_message);
90         printf("\n%s\n", display_message);
91     }

```



```

91     sem_wait(&server_semaphore);
92     client->active = 0;
93     printf("\nClient %d disconnected\n", client->id);
94     sem_post(&server_semaphore);
95
96     close(client->socket);
97     return 0;
98 }
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);

```

```

159         for (int i = 0; i < client_count; i++) {
160             if (clients[i].active) {
161                 send(clients[i].socket, "SERVER_SHUTDOWN", 14, 0);
162                 close(clients[i].socket);
163             }
164         }
165         sem_post(&server_semaphore);
166         exit(0);
167         break;
168
169     default:
170         printf("Invalid choice!\n");
171         show_menu();
172     }
173 }
174 return 0;
175 }
176
177 int main() {
178     int server_fd, new_socket, c;
179     struct sockaddr_in server, client;
180     pthread_t thread_id, menu_thread;
181
182     // Initialize semaphore
183     if (sem_init(&server_semaphore, 0, 1) < 0) {
184         perror("Semaphore initialization failed");
185         return 1;
186     }
187
188     server_fd = socket(AF_INET, SOCK_STREAM, 0);
189     if (server_fd == -1) {
190         perror("Could not create socket");
191         log_error("Could not create socket");
192         return 1;
193     }
194
195     server.sin_family = AF_INET;
196     server.sin_addr.s_addr = INADDR_ANY;
197     server.sin_port = htons(PORT);
198
199     if (bind(server_fd, (struct sockaddr *)&server, sizeof(server)) < 0) {
200         perror("bind failed");
201         log_error("bind failed");
202         return 1;
203     }
204
205     listen(server_fd, 3);
206     printf("Server started on port %d\n", PORT);
207     printf("Showing all client activity - Use menu for additional options\n");
208
209     if (pthread_create(&menu_thread, NULL, server_menu, NULL) < 0) {
210         perror("could not create menu thread");
211         log_error("could not create menu thread");
212         return 1;
213     }
214
215     c = sizeof(struct sockaddr_in);
216     while (server_running && (new_socket = accept(server_fd, (struct sockaddr *)&client, (
socklen_t*)&c))) {
217         if (new_socket < 0) {
218             perror("accept failed");
219             log_error("accept failed");
220             continue;
221         }
222
223         sem_wait(&server_semaphore);
224         if (client_count >= MAX_CLIENTS) {
225             send(new_socket, "Server is full", 13, 0);

```

```

226         close(new_socket);
227         sem_post(&server_semaphore);
228         continue;
229     }
230
231     clients[client_count].socket = new_socket;
232     clients[client_count].id = client_count;
233     clients[client_count].active = 1;
234     clients[client_count].msg_count = 0;
235
236     if (pthread_create(&thread_id, NULL, handle_client, (void*)&clients[client_count]) <
237 0) {
238         perror("could not create thread");
239         log_error("could not create thread");
240         sem_post(&server_semaphore);
241         return 1;
242     }
243
244     client_count++;
245     sem_post(&server_semaphore);
246 }
247
248 sem_destroy(&server_semaphore);
249 return 0;

```

Listing 2: server code - Socket Programming Code

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  #include <semaphore.h>
9  #include <errno.h>
10
11 #define PORT 8080
12 #define LOG_FILE "client_log.txt"
13
14 int client_running = 1;
15 sem_t client_semaphore;
16
17 void log_error(const char *message) {
18     FILE *log_file = fopen(LOG_FILE, "a");
19     if (log_file == NULL) {
20         perror("Failed to open log file");
21         return;
22     }
23     fprintf(log_file, "ERROR: %s: %s\n", message, strerror(errno));
24     fclose(log_file);
25 }
26
27 void show_menu() {
28     printf("\n=== Client Menu ===\n");
29     printf("1. Send Message\n");
30     printf("2. Exit\n");
31     printf("Enter choice: ");
32 }
33
34 void *receive_messages(void *socket_desc) {
35     int sock = *(int*)socket_desc;
36     char server_message[1024];
37     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

```

```

106     if (pthread_create(&thread_id, NULL, receive_messages, (void*)&sock) < 0) {
107         perror("could not create thread");
108         log_error("could not create thread");
109         return 1;
110     }
111
112     while (client_running) {
113         show_menu();
114         fgets(input, sizeof(input), stdin);
115         choice = atoi(input);
116
117         sem_wait(&client_semaphore); // Protect shared resource access
118         switch (choice) {
119             case 1:
120                 printf("Enter message: ");
121                 fgets(message, 1024, stdin);
122                 message[strcspn(message, "\n")] = '\0';
123
124                 if (send(sock, message, strlen(message), 0) < 0) {
125                     perror("Send failed");
126                     log_error("Send failed");
127                     client_running = 0;
128                 } else {
129                     printf("Message sent to server\n");
130                 }
131                 break;
132
133             case 2:
134                 printf("Exiting...\n");
135                 client_running = 0;
136                 break;
137
138             default:
139                 printf("Invalid choice!\n");
140         }
141         sem_post(&client_semaphore); // Release semaphore
142     }
143
144     sem_destroy(&client_semaphore);
145     close(sock);
146     return 0;
147 }

```

Listing 3: client code - Socket Programming Code

```
(base) tazmeen@tazmeen:~/FINAL$ ./server
Server started on port 8080
Showing all client activity - Use Menu for additional options

=== Server Menu ===
1. Send Broadcast Message
2. View Individual Client Messages
3. Exit
Enter choice :
Client 0 connected
Client 1 connected
Client 2 connected
1
Enter message to broadcast: hello everyone
Message broadcasted
Client 2: HI
Client 0: hello
Client 1: hello

(server_log.txt)
tazmeen@tazmeen:~/FINAL$ ./client
Connected to server

=== Client Menu ===
1. Send Message
2. Exit
Enter choice:
SERVER: hello everyone
1
Enter message: hello
Message sent to server

tazmeen@tazmeen:~/FINAL$ ./client
Connected to server

=== Client Menu ===
1. Send Message
2. Exit
Enter choice:
SERVER: hello everyone
1
Enter message: hello
Message sent to server

tazmeen@tazmeen:~/FINAL$ ./client
Connected to server

=== Client Menu ===
1. Send Message
2. Exit
Enter choice:
SERVER: hello everyone
1
Enter message: hello
Message sent to server
```

Figure 6: local socket programming

```
(base) tazmeen@tazmeen:~/FINAL$ ./server
Client 1 connected
Client 2 connected
1
Enter message to broadcast: hello everyone
Message broadcasted
Client 2: HI
Client 0: hello
Client 1: hello
2
Active clients:
Client 0
Client 1
Client 2
Enter client ID to view messages: 1
Messages from Client 1:
Client 1: hello

(server_log.txt)
tazmeen@tazmeen:~/FINAL$ ./client
Connected to server

=== Client Menu ===
1. Send Message
2. Exit
Enter choice:
SERVER: hello everyone
1
Enter message: hello
Message sent to server

tazmeen@tazmeen:~/FINAL$ ./client
Connected to server

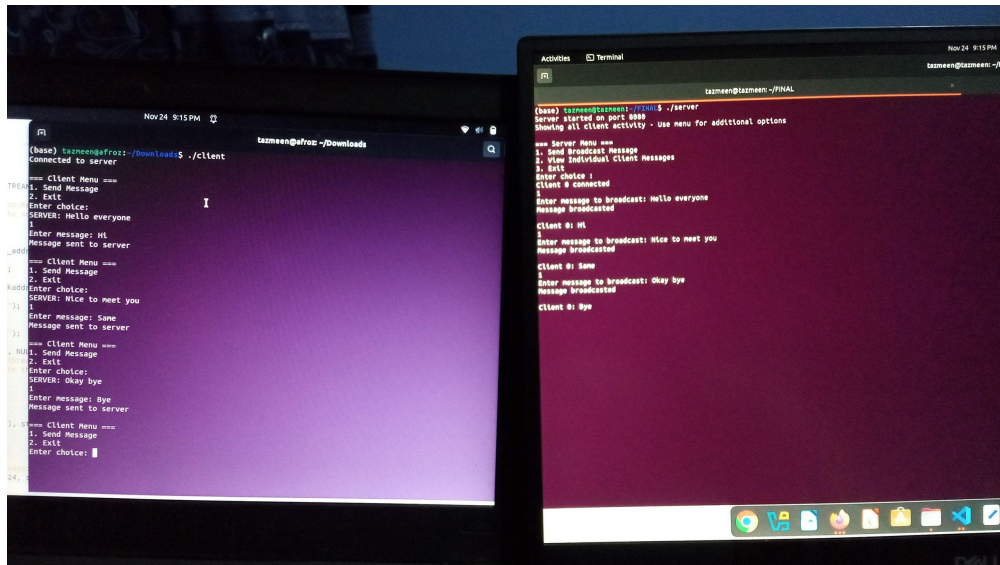
=== Client Menu ===
1. Send Message
2. Exit
Enter choice:
SERVER: hello everyone
1
Enter message: hello
Message sent to server

tazmeen@tazmeen:~/FINAL$ ./client
Connected to server

=== Client Menu ===
1. Send Message
2. Exit
Enter choice:
SERVER: hello everyone
1
Enter message: hello
Message sent to server
```

Figure 7: local socket programming

Part 2: Distributed System Socket Programming



```
(base) tazmeen@froz:~/Downloads$ ./client
Connected to server

== Client Menu ==
1. Send Message
2. Exit
Enter choice: 1
SERVER: Hello everyone
Enter Message: Hi
Message sent to server

== Client Menu ==
1. Send Message
2. Exit
Enter choice: 1
SERVER: Nice to meet you
Enter Message: Same
Message sent to server

== Client Menu ==
1. Send Message
2. Exit
Enter choice: 1
SERVER: okay bye
Enter Message: Bye
Message sent to server

== Client Menu ==
1. Send Message
2. Exit
Enter choice: 1

(base) tazmeen@tazmeen:~/FINAL$ ./server
Server started on port 8080
Showing all client activity . Use menu for additional options

== Server Menu ==
1. Send Broadcast Message
2. View Individual Client Messages
3. Exit
Enter choice: 1
Client 0 connected
Enter message to broadcast: Hello everyone
Message broadcasted
Client 0: Hi
Enter message to broadcast: Nice to meet you
Message broadcasted
Client 0: Same
Enter message to broadcast: Okay bye
Message broadcasted
Client 0: Bye
```

Figure 8: distributed system socket programming