Name: Ahmed Ali
Roll No: 22p-9318
Section: BS(SE)-5B

OPERATING SYSTEM PROJECT

TASK 1 CODE: CPU Scheduling Implementation

NOTE: The implementation of various CPU scheduling algorithms.

```c
#include<stdio.h> //standard I/O library for input and output operations
#include<stdlib.h>//standard library for functions like malloc, exit, etc.
#include<string.h>//library for string manipulation functions
#include<ctype.h>//library for character handling functions
//defining structure for task
typedef struct
{
    char name[10]; //name of the task (max 9 characters+null terminator)
    int priority;//priority of the task (higher value = higher priority)
    int cpu_burst;//CPU burst time required for the task
} task_t;
//prototypes for scheduling algorithms
void fcfs(task_t tasks[], int task_count);
void sjf(task_t tasks[], int task_count);
void priority_scheduling(task_t tasks[], int task_count);
void round_robin(task_t tasks[], int task_count, int time_quantum);
int load_tasks_from_file(const char* filename, task_t tasks[], int *task_count);
int main()
{
    task_t tasks[100]; //array to store up to 100 tasks
    int task_count=0; //no of tasks initialized to 0
    //load task list from the file schedule.txt
    if (!load_tasks_from_file("schedule.txt", tasks, &task_count))
    {
        printf("Error: Could not load tasks. Exiting program.\n");
        return 1; //exit the program if task loading fails
    }
    //call each scheduling algorithm and display the results
    printf("First-Come-First-Served (FCFS) Scheduling:\n");
    fcfs(tasks, task_count);
    printf("\nShortest Job First (SJF) Scheduling:\n");
    sjf(tasks, task_count);
```

```
33      printf("\nPriority Scheduling:\n");
34      priority_scheduling(tasks, task_count);
35      printf("\nRound Robin Scheduling:\n");
36      int time_quantum=5; //time quantum for Round Robin scheduling
37      round_robin(tasks, task_count, time_quantum);
38      return 0; //exit the program successfully
39  }
40  //load tasks from a file with error handling
41  int load_tasks_from_file(const char *filename, task_t tasks[], int *
        task_count)
42  {
43      FILE *file=fopen(filename, "r"); //open the file in read mode
44      if(file==NULL)
45      {
46          printf("Error: Unable to open file %s.\n", filename);
47          return 0; //return 0 to indicate failure
48      }
49      int line=0; //line number tracker for error reporting
50      while(fscanf(file, "%s %d %d", tasks[*task_count].name, &tasks[*
            task_count].priority, &tasks[*task_count].cpu_burst)!=EOF)
51      {
52          line++; //increment line number for each read
53          //validate task name length
54          if(strlen(tasks[*task_count].name)>9)
55          {
56              printf("Error: Task name too long on line %d. Skipping
                    this task.\n", line);
57              continue; //skip this task and move to the next line
58          }
59          //validate priority and CPU burst
60          if (tasks[*task_count].priority<=0 || tasks[*task_count].
                cpu_burst<=0)
61          {
62              printf("Error: Invalid priority or CPU burst time on
                    line %d. Skipping this task.\n", line);
63              continue; //skip this invalid task
64          }
65
66          (*task_count)++; //increment the count of valid tasks
67          if(*task_count>100)
68          {
69              printf("Warning: Task limit exceeded only the first 100
                    tasks will be processed.\n");
70              break; //stop reading more tasks
71          }
72      }
73      fclose(file); //close the file
```

```
74      if(*task_count==0)
75      {
76          printf("Error: No tasks found in the file\n");
77          return 0; // return 0 to indicate failure
78      }
79      return 1; //return 1 to indicate successful loading of tasks
80  }
81  //FCFS Scheduling Algorithm
82  void fcfs(task_t tasks[], int task_count)
83  {
84      int waiting_time=0, turn_around_time=0; //initialize waiting and
            turnaround times
85      printf("Task\tCPU Burst\tWaiting Time\tTurnaround Time\n");
86      for (int i=0; i<task_count; i++)
87      {
88          turn_around_time+=tasks[i].cpu_burst; //increment turnaround
                time by current tasks CPU burst
89          printf("%s\t%d\t\t%d\t\t%d\n", tasks[i].name, tasks[i].
               cpu_burst, waiting_time, turn_around_time);
90          waiting_time+=tasks[i].cpu_burst; //update waiting time for
               the next task
91      }
92  }
93  //SJF Scheduling Algorithm(Non preemptive)
94  void sjf(task_t tasks[], int task_count)
95  {
96      //sort tasks by CPU burst time
97      for(int i=0; i<task_count-1; i++)
98      {
99          for(int j=i+1; j<task_count; j++)
100         {
101             if(tasks[i].cpu_burst>tasks[j].cpu_burst)
102             {
103                 task_t temp=tasks[i]; //swap tasks to sort them
104                 tasks[i]=tasks[j];
105                 tasks[j]=temp;
106             }
107         }
108     }
109     int waiting_time=0, turn_around_time=0; //initialize waiting and
            turnaround times
110     printf("Task\tCPU Burst\tWaiting Time\tTurnaround Time\n");
111     for(int i=0; i<task_count; i++)
112     {
113         turn_around_time+=tasks[i].cpu_burst; //increment turnaround
                time by CPU burst
114         printf("%s\t%d\t\t%d\t\t%d\n", tasks[i].name, tasks[i].
```

```
            cpu_burst, waiting_time, turn_around_time);
        waiting_time+=tasks[i].cpu_burst; //update waiting time for
            the next task
    }
}
//priority Scheduling Algorithm(Non preemptive)
void priority_scheduling(task_t tasks[], int task_count)
{
    //sort tasks by priority (higher priority first)
    for(int i=0; i<task_count-1; i++)
    {
        for(int j=i+1; j<task_count; j++)
        {
            if (tasks[i].priority<tasks[j].priority)
            {
                task_t temp=tasks[i]; //swap tasks to sort them
                tasks[i]=tasks[j];
                tasks[j]=temp;
            }
        }
    }
    int waiting_time=0, turn_around_time=0; //initialize waiting and
        turnaround times
    printf("Task\tPriority\tCPU Burst\tWaiting Time\tTurnaround Time
       \n");
    for(int i=0; i<task_count; i++)
    {
        turn_around_time+=tasks[i].cpu_burst; //increment turnaround
            time by CPU burst
        printf("%s\t%d\t\t%d\t\t%d\t\t%d\n", tasks[i].name, tasks[i
           ].priority, tasks[i].cpu_burst, waiting_time,
           turn_around_time);
        waiting_time+=tasks[i].cpu_burst; //update waiting time for
            the next task
    }
}
//RR Scheduling Algorithm
void round_robin(task_t tasks[], int task_count, int time_quantum)
{
    int remaining_burst[100]; //arrayto track remaining burst times
    for(int i=0; i<task_count; i++)
    {
        remaining_burst[i]=tasks[i].cpu_burst;
    }
    int time=0; //initialize elapsed time
    printf("Task\tCPU Burst\tRemaining Time\n");
    while(1)//loop until all tasks are completed
```

```
154     {
155         int done=1; //flag to check if all tasks are done
156         for(int i=0; i<task_count; i++)
157         {
158             if(remaining_burst[i]>0)
159             {
160                 done=0; //mark that work is pending
161                 if(remaining_burst[i]>time_quantum)
162                 {
163                     remaining_burst[i]-=time_quantum;//decrease
                        remaining burst by quantum
164                     time += time_quantum;//increment elapsed time
165                     printf("%s\t%d\t\t%d\n", tasks[i].name, tasks[i
                        ].cpu_burst, remaining_burst[i]);
166                 }
167                 else
168                 {
169                     time+=remaining_burst[i];//increment elapsed
                        time
170                     remaining_burst[i]=0;//mark the task as
                        completed
171                     printf("%s\t%d\t\t%d\n", tasks[i].name, tasks[i
                        ].cpu_burst, remaining_burst[i]);
172                 }
173             }
174         }
175         if(done)break;//exit loop if all tasks are completed
176     }
177 }
```

TASK 2: Socket Programming Implementation

Part 1: Local System Socket Programming

SERVER CODE:

```c
#include<stdio.h> //standard input/output library
#include<stdlib.h> //standard library for memory allocation, process control
#include<string.h> //string handling library
#include<unistd.h> //provides access to the POSIX operating system API
#include<sys/socket.h>//provides socket functions
#include<netinet/in.h> //defines internet address
#include<pthread.h> //provides threading functionality
#define MAX_CLIENTS 4 //maximum number of clients allowed
#define BUFFER_SIZE 256 //size of the buffer for messages
int client_sockets[MAX_CLIENTS];  //array to store client sockets
int client_count=0; //keeps track of the number of connected clients
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;//mutex to ensure thread safe operations
//broadcast message from the server to all connected clients
void broadcast_to_clients(const char *message)
{
    pthread_mutex_lock(&mutex); //lock the mutex to ensure thread safety
    for (int i=0; i<client_count; i++) //iterate through all connected clients
    {
        // send the message to each client
        if(write(client_sockets[i], message, strlen(message))<0)
        {
            perror("error broadcasting to client"); // print error if write fails
        }
    }
    pthread_mutex_unlock(&mutex);//unlock the mutex
}
//handle communication with an individual client
void *handle_client(void *socket_desc)
{
    int client_socket = *(int *)socket_desc; //extract the client socket descriptor
    char buffer[BUFFER_SIZE]; //buffer to hold client messages
    while(1)//loop to continuously receive messages from the client
    {
        bzero(buffer, BUFFER_SIZE); //clear the buffer
```

```
35
36          //read the client message into the buffer
37          int n=read(client_socket, buffer, BUFFER_SIZE-1);
38          if(n<=0)//check if the client disconnected or an error
                occurred
39          {
40              printf("a client disconnected\n");
41              break; // exit the loop if the client disconnects
42          }
43
44          printf("message from client: %s", buffer);//print the
                received message
45      }
46      //remove the client socket from the list of active sockets
47      pthread_mutex_lock(&mutex); //lock the mutex
48      for(int i=0; i<client_count; i++) //find the client's socket in
            the list
49      {
50          if(client_sockets[i]==client_socket)
51          {
52              //shift the remaining sockets down to fill the gap
53              for(int j=i; j<client_count-1; j++)
54              {
55                  client_sockets[j] =client_sockets[j+1];
56              }
57              client_count--; //decrement the client count
58              break; //exit the loop
59          }
60      }
61      pthread_mutex_unlock(&mutex); //unlock mutex
62      close(client_socket); //close the client socket
63      free(socket_desc); //free the memory allocated for the socket
            descriptor
64      pthread_exit(NULL); //exit the thread
65  }
66  //allow the server to broadcast messages to all clients
67  void *server_broadcast(void *arg)
68  {
69      char buffer[BUFFER_SIZE]; //buffer to hold the broadcast message
70      while(1)//loop to continuously get server messages
71      {
72          bzero(buffer, BUFFER_SIZE);//clear the buffer
73          printf("server (broadcast): "); //prompt the server for
                input
74          fgets(buffer, BUFFER_SIZE-1, stdin); //read the input into
                the buffer
```

```
75          broadcast_to_clients(buffer); //broadcast the message to all
                clients
76          //check if the server wants to shut down
77          if (strncmp("down", buffer, 3)==0)
78          {
79              printf("server shutting down broadcast.\n");
80              exit(0);//exit the server process
81          }
82      }
83      pthread_exit(NULL);//exit the broadcast thread
84  }
85  int main(int argc, char *argv[])
86  {
87      int server_socket, client_socket, port_no; //socket descriptors
            and port number
88      socklen_t client_len; //length of the client address
89      struct sockaddr_in server_addr, client_addr; //server and client
            address structures
90      //prompt the user to enter the port number
91      printf("Enter the port number: ");
92      scanf("%d", &port_no);
93      server_socket=socket(AF_INET, SOCK_STREAM, 0); //create a socket
94      if (server_socket<0)
95      {
96          perror("error opening socket"); //print error if socket
                creation fails
97          exit(EXIT_FAILURE);
98      }
99      bzero((char *)&server_addr, sizeof(server_addr));//clear the
            server address structure
100     server_addr.sin_family=AF_INET;//set address family to IPv4
101     server_addr.sin_addr.s_addr=INADDR_ANY;//accept connections from
            any client
102     server_addr.sin_port=htons(port_no);// set the port number htons
            converts a port number from host byte order to network byte
            order for compatibility in network communication
103     //bind the socket to the specified port and address
104     if(bind(server_socket, (struct sockaddr *)&server_addr, sizeof(
            server_addr))<0)
105     {
106         perror("error on binding"); //print error if binding fails
107         close(server_socket); //close the server socket
108         exit(EXIT_FAILURE);
109     }
110     listen(server_socket, MAX_CLIENTS); //start listening for
            incoming connections
```

```
    printf("server started on port %d\n", port_no); //print the server status
    pthread_t broadcast_thread; //thread for broadcasting server messages
    pthread_create(&broadcast_thread, NULL, server_broadcast, NULL); //create the broadcast thread
    while(1)//loop to accept client connections
    {
        client_len=sizeof(client_addr);//set the length of the client address
        client_socket=accept(server_socket, (struct sockaddr *)&client_addr, &client_len); //accept a connection
        if(client_socket<0)
        {
            perror("error on accept"); //print error if accepting fails
            continue; //continue to the next iteration
        }
        pthread_mutex_lock(&mutex); //lock the mutex
        client_sockets[client_count++]=client_socket; //add the client to the list of active sockets
        pthread_mutex_unlock(&mutex); //unlock the mutex
        printf("client connected\n");
        pthread_t client_thread; //thread for handling the client
        int *new_sock=malloc(sizeof(int)); //allocate memory for the socket descriptor
        *new_sock=client_socket; //set the socket descriptor
        pthread_create(&client_thread, NULL, handle_client, (void *)new_sock); // create the client thread
    }
    close(server_socket); //close the server socket
    return 0; //return 0 to indicate successful execution
}
```

CLIENT CODE:

```
#include<stdio.h>//standard input/output library
#include<stdlib.h>//standard library for memory allocation, process control
#include<unistd.h>//provides access to the POSIX operating system API
#include<string.h>//string handling library
#include<sys/types.h>//defines data types used in system calls
#include<sys/socket.h>//provides socket functions
#include<netinet/in.h>//defines internet address family
#include<netdb.h>//provides functions for host name resolution
```

```c
#include<pthread.h>//provides threading functionality
//function to handle errors and print error message
void error(const char *msg)
{
    perror(msg);  //print error message
    exit(1); //exit the program if an error occurs
}
int sockfd; //global socket descriptor for communication
//thread function to continuously read messages from the server
void *receive_messages(void *arg)
{
    char buffer[256]; //buffer to hold messages from the server
    int n; //variable to store the number of bytes read
    while(1)
    {
        bzero(buffer, 256); //clear the buffer to avoid leftover
           data
        //read message from the server
        n=read(sockfd, buffer, 255);
        if(n>0)
        {
            printf("\nServer Broadcast: %s", buffer); //print the
               server's message
            printf("Client: "); //re-prompt for client input
            fflush(stdout); //ensure the prompt is shown immediately
        }
        else if(n==0)
        { //check if the server disconnected
            printf("\nServer disconnected.\n");
            exit(0); //exit if the server disconnects
        }
        else
        {
            error("ERROR reading from socket"); //error handling
        }
    }
    return NULL; //return NULL upon thread completion
}
int main()
{
    int portno, n; //port number and variable for bytes read/written
    struct sockaddr_in serv_addr; //structure for server address
    struct hostent *server; //pointer for the server's DNS entry
    char buffer[256]; //buffer to hold data sent/received
    char hostname[256]; //to hold the server hostname

    //prompt the user for server hostname and port number
```

```
54      printf("Enter hostname: ");
55      scanf("%s", hostname); //read server hostname from user input
56      printf("Enter port number: ");
57      scanf("%d", &portno); //read port number from user input
58      getchar(); //consume the newline character left by scanf
59
60      //create socket for communication
61      sockfd=socket(AF_INET, SOCK_STREAM, 0);
62      if(sockfd<0)
63          error("ERROR opening socket"); //error handling
64
65      //get the server DNS entry using the hostname
66      server=gethostbyname(hostname);
67      if(server==NULL)
68      {
69          fprintf(stderr, "ERROR, no such host\n"); //error message if
                hostname is invalid
70          exit(1); //exit the program
71      }
72      //initialize the server address structure
73      bzero((char *)&serv_addr, sizeof(serv_addr)); //clear the
            structure
74      serv_addr.sin_family=AF_INET; //set address family to IPv4
75      bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr
            , server->h_length); //copy server address
76      serv_addr.sin_port=htons(portno); //set the server port number,
            htons converts a port number from host byte order to network
            byte order for compatibility in network communication
77
78      //connect to the server
79      if(connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(
            serv_addr))<0)
80          error("ERROR connecting"); //error handling
81
82      printf("Client: Connected to server\n");
83      //create a thread to handle incoming messages
84      pthread_t receive_thread; //thread ID for the receiver thread
85      pthread_create(&receive_thread, NULL, receive_messages, NULL);
            //create the thread
86      while(1)
87      {
88          bzero(buffer, 256); //clear the buffer before taking user
                input
89          printf("Client: "); //prompt the user for input
90          fgets(buffer, 255, stdin); //read user input
91          //send the user's message to the server
92          n=write(sockfd, buffer, strlen(buffer));
```

```
93          if(n<0)
94              error("ERROR writing to socket"); //error handling
95          //exit the client program if the user types "Bye"
96          if(strncmp("down", buffer, 3)==0)
97          {
98              printf("Client: Disconnected\n");
99              break; //exit the loop
100         }
101     }
102     close(sockfd); //close the socket
103     pthread_cancel(receive_thread); //terminate the receive thread
104     pthread_join(receive_thread, NULL); //ensure thread cleanup
105     return 0; //return 0 to indicate successful execution
106 }
```

PART 2: Distributed System Socket Programming

The implementation is completed, as both server and client codes are in running condition. The representation of PART 2 will be done during the viva, *In'Sha'Allah*.

SCREENSHOTS OF OUTPUT SEE BELOW PAGES

```
(base) amei-302@amei302-HP-EliteBook-840-G3:~/Desktop/OS Project$ ./task1.out
First-Come-First-Served (FCFS) Scheduling:
Task    CPU Burst       Waiting Time    Turnaround Time
T1      20              0               20
T2      25              20              45
T3      25              45              70
T4      15              70              85
T5      10              85              95

Shortest Job First (SJF) Scheduling:
Task    CPU Burst       Waiting Time    Turnaround Time
T5      10              0               10
T4      15              10              25
T1      20              25              45
T2      25              45              70
T3      25              70              95

Priority Scheduling:
Task    Priority        CPU Burst       Waiting Time    Turnaround Time
T5      10              10              0               10
T1      4               20              10              30
T4      3               15              30              45
T3      3               25              45              70
T2      2               25              70              95

Round Robin Scheduling:
Task    CPU Burst       Remaining Time
T5      10              5
T1      20              15
T4      15              10
T3      25              20
T2      25              20
T5      10              0
T1      20              10
T4      15              5
T3      25              15
T2      25              15
T1      20              5
T4      15              0
T3      25              10
T2      25              10
T1      20              0
T3      25              5
T2      25              5
T3      25              0
T2      25              0
(base) amei-302@amei302-HP-EliteBook-840-G3:~/Desktop/OS Project$
```


Figure 1: Output of Task 1: CPU Scheduling Implementation

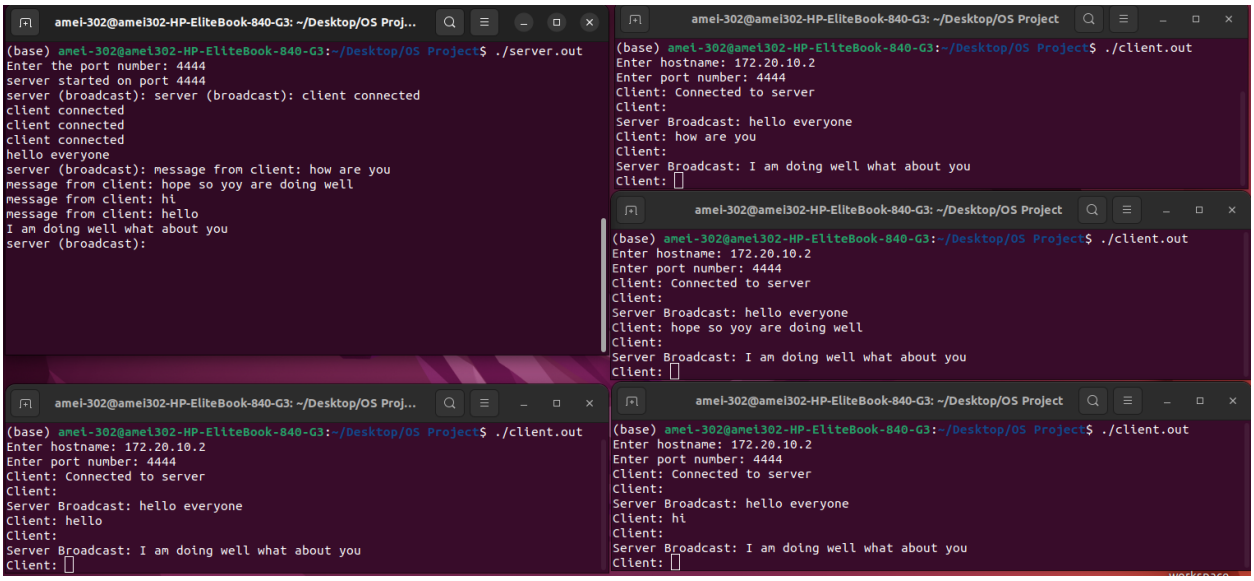

Figure 2: Output of TASK 2: Part 1: Local System Socket Programming

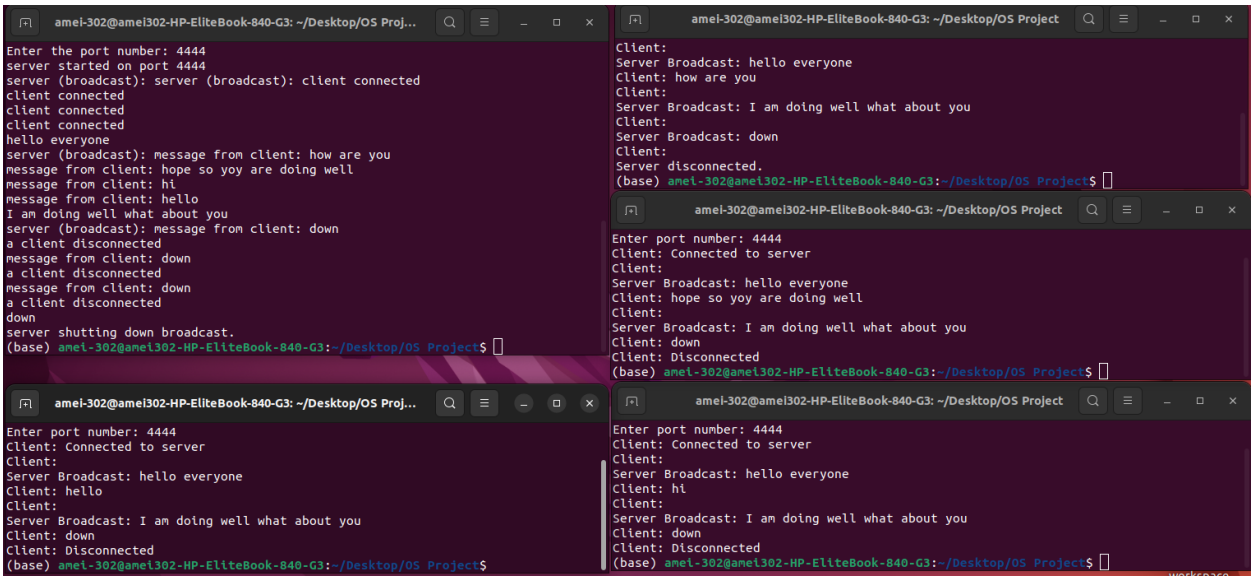

Figure 3: Output of TASK 2: Part 1: Local System Socket Programming, prompting down to disconnect