CSA0486 – OPERATING SYSTEMS FOR AI EMERGING TECHNOLOGY

EXPERIMENT 1

AIM:

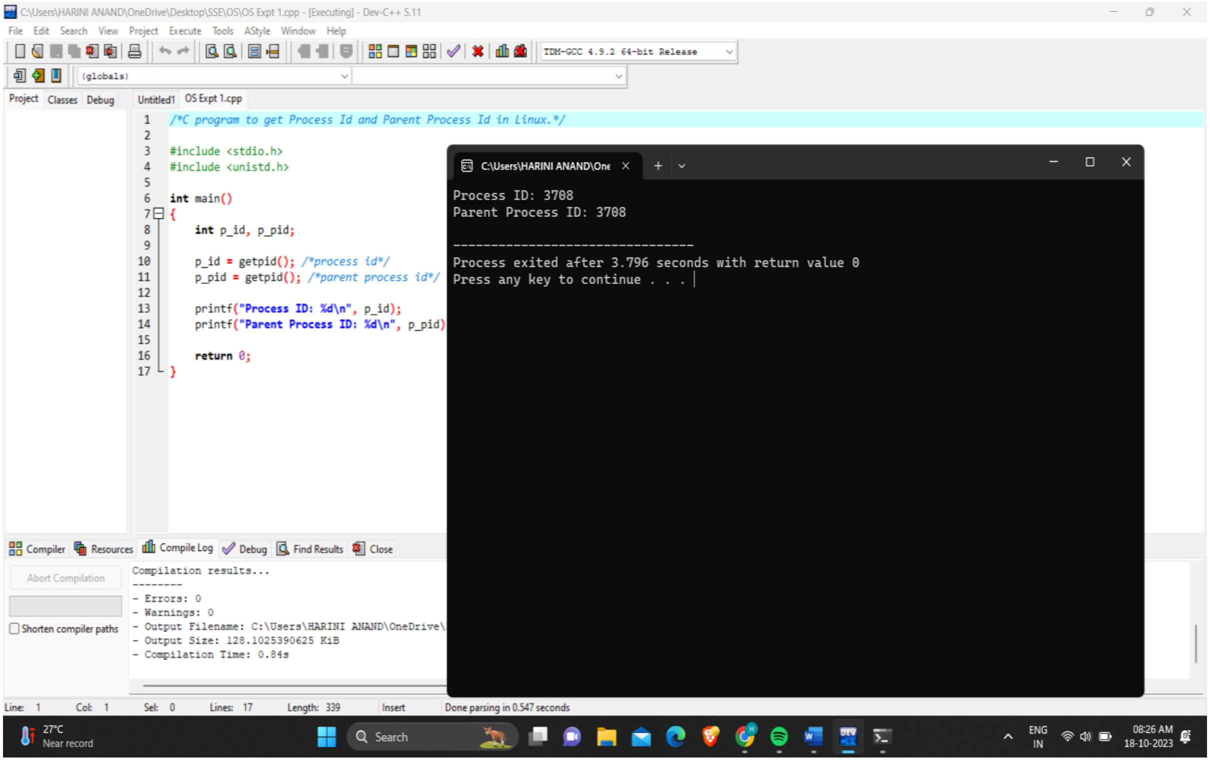
To create a new process by invoking the appropriate system call. Get the process identifier of the currently running process and its respective parent using system calls and display the same using a C program.

PROCEDURE:

```
/*C program to get Process Id and Parent Process Id in Linux.*/

#include <stdio.h>

#include <unistd.h>

int main()

{

   int p_id, p_pid;


   p_id = getpid(); /*process id*/

   p_pid = getpid(); /*parent process id*/


   printf("Process ID: %d\n", p_id);

   printf("Parent Process ID: %d\n", p_pid);


   return 0;

}
```

RESULT:

The Program is successfully verified.



EXPERIMENT 2

AIM:

To Identify the system calls to copy the content of one file to another and illustrate the same using a C program.

PROCEDURE:

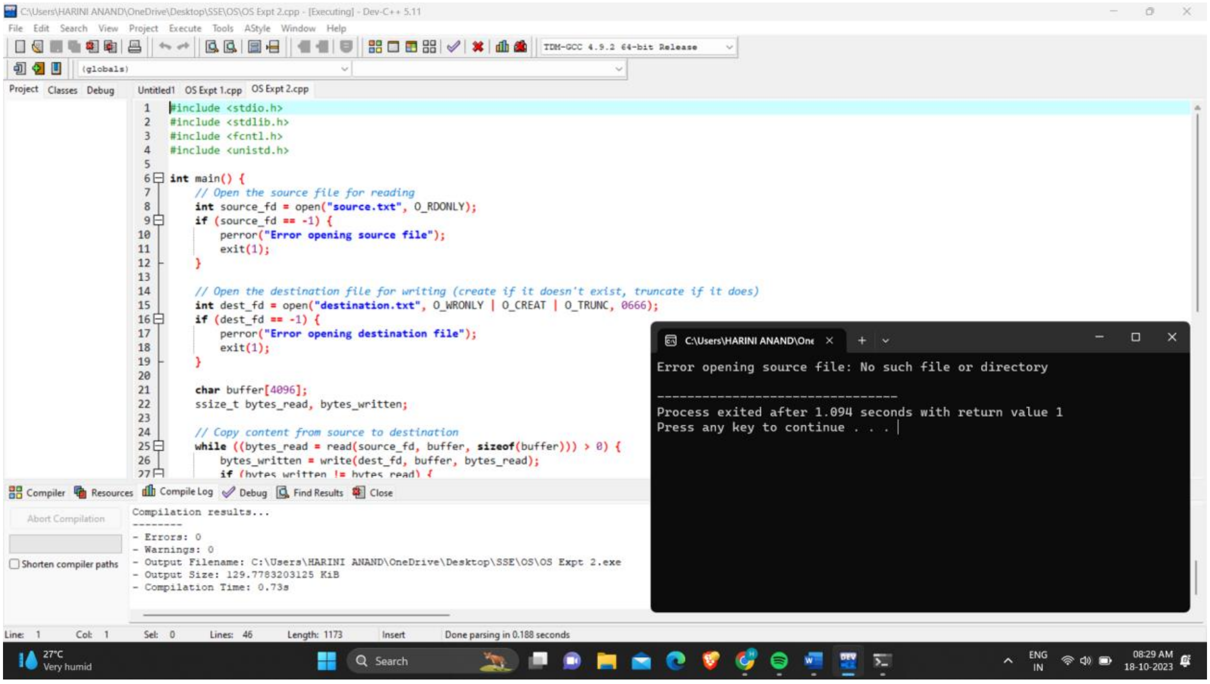
```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
    // Open the source file for reading
    int source_fd = open("source.txt", O_RDONLY);
    if (source_fd == -1) {
        perror("Error opening source file");
        exit(1);
    }
```

```c
// Open the destination file for writing (create if it doesn't exist, truncate if it does)
int dest_fd = open("destination.txt", O_WRONLY | O_CREAT | O_TRUNC, 0666);
if (dest_fd == -1) {
    perror("Error opening destination file");
    exit(1);
}

char buffer[4096];
ssize_t bytes_read, bytes_written;

// Copy content from source to destination
while ((bytes_read = read(source_fd, buffer, sizeof(buffer))) > 0) {
    bytes_written = write(dest_fd, buffer, bytes_read);
    if (bytes_written != bytes_read) {
        perror("Write error");
        exit(1);
    }
}

if (bytes_read == -1) {
    perror("Read error");
    exit(1);
}

// Close the files
close(source_fd);
close(dest_fd);

printf("File copy completed successfully.\n");

return 0;
```

}

RESULT:

The Program is successfully verified.



EXPERIMENT 3

AIM:

To Design a CPU scheduling program with C using First Come First Served technique with the following considerations. A All processes are activated at time 0. b. Assume that no process waits on I/O devices.

PROCEDURE:

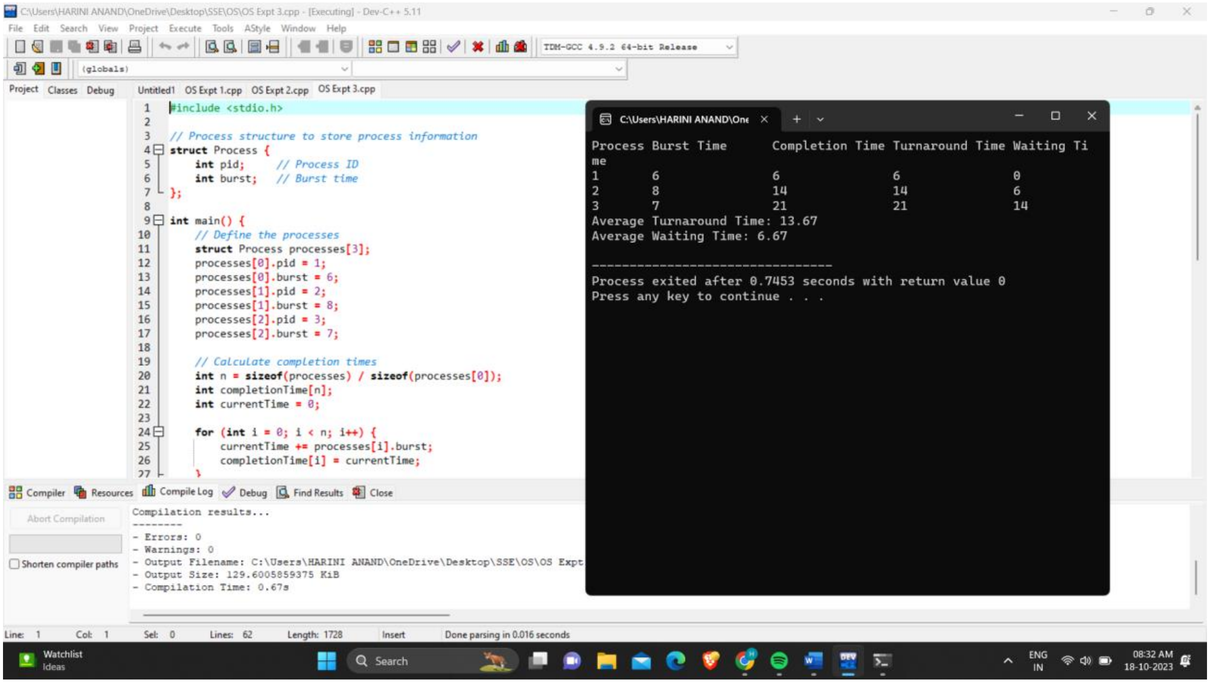
```
#include <stdio.h>


// Process structure to store process information
struct Process {
    int pid;      // Process ID
    int burst;    // Burst time
};


int main() {
    // Define the processes
```

```
struct Process processes[3];
processes[0].pid = 1;
processes[0].burst = 6;
processes[1].pid = 2;
processes[1].burst = 8;
processes[2].pid = 3;
processes[2].burst = 7;

// Calculate completion times
int n = sizeof(processes) / sizeof(processes[0]);
int completionTime[n];
int currentTime = 0;

for (int i = 0; i < n; i++) {
    currentTime += processes[i].burst;
    completionTime[i] = currentTime;
}

// Calculate turnaround time and waiting time
int turnaroundTime[n];
int waitingTime[n];

for (int i = 0; i < n; i++) {
    turnaroundTime[i] = completionTime[i];
    waitingTime[i] = turnaroundTime[i] - processes[i].burst;
}

// Calculate average turnaround time and average waiting time
float avgTurnaroundTime = 0;
float avgWaitingTime = 0;

for (int i = 0; i < n; i++) {
```

```c
        avgTurnaroundTime += turnaroundTime[i];
        avgWaitingTime += waitingTime[i];
    }

    avgTurnaroundTime /= n;
    avgWaitingTime /= n;

    // Display the results
    printf("Process\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting Time\n");

    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid, processes[i].burst, completionTime[i], turnaroundTime[i], waitingTime[i]);
    }

    printf("Average Turnaround Time: %.2f\n", avgTurnaroundTime);
    printf("Average Waiting Time: %.2f\n", avgWaitingTime);

    return 0;
}
```

RESULT:

The Program is successfully verified.



EXPERIMENT 4

AIM:

To Construct a scheduling program with C that selects the waiting process with the smallest execution time to execute next.

PROCEDURE:

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

struct Process {
    int id;          // Process ID
    int burst_time;  // Burst time
    int priority;    // Priority
};

// Function to perform Priority Scheduling
void priorityScheduling(struct Process processes[], int n) {
    int total_time = 0;
    int total_waiting_time = 0;
    int total_turnaround_time = 0;
```

```
    // Sort the processes based on priority (smallest priority first)
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].priority > processes[j + 1].priority) {
                // Swap the processes
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }

    printf("Process Execution Order:\n");
    printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");

    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time, total_time, total_time +
processes[i].burst_time);
        total_waiting_time += total_time;
        total_turnaround_time += total_time + processes[i].burst_time;
        total_time += processes[i].burst_time;
    }

    printf("Average Waiting Time: %.2f\n", (float)total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", (float)total_turnaround_time / n);
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
```

```
    struct Process processes[n];

    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Enter burst time for Process %d: ", i + 1);
        scanf("%d", &processes[i].burst_time);
        printf("Enter priority for Process %d: ", i + 1);
        scanf("%d", &processes[i].priority);
    }

    priorityScheduling(processes, n);

    return 0;
}
```

RESULT:

The Program is successfully verified.

EXPERIMENT 5

AIM:

To Construct a scheduling program with C that selects the waiting process with the highest priority to execute next.

PROCEDURE:

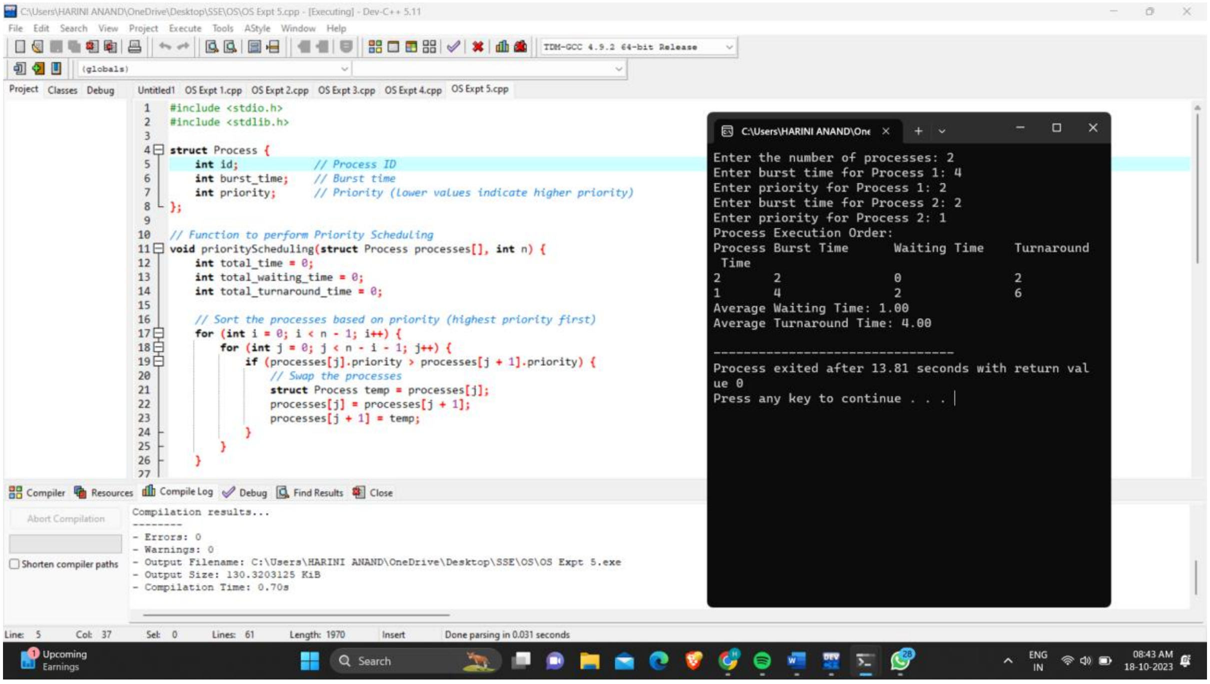
```
#include <stdio.h>
#include <stdlib.h>

struct Process {
    int id;           // Process ID
    int burst_time;   // Burst time
    int priority;     // Priority (lower values indicate higher priority)
};
```

```
// Function to perform Priority Scheduling
void priorityScheduling(struct Process processes[], int n) {
    int total_time = 0;
    int total_waiting_time = 0;
    int total_turnaround_time = 0;

    // Sort the processes based on priority (highest priority first)
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].priority > processes[j + 1].priority) {
                // Swap the processes
                struct Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }

    printf("Process Execution Order:\n");
    printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");

    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time, total_time, total_time + processes[i].burst_time);
        total_waiting_time += total_time;
        total_turnaround_time += total_time + processes[i].burst_time;
        total_time += processes[i].burst_time;
    }

    printf("Average Waiting Time: %.2f\n", (float)total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", (float)total_turnaround_time / n);
}
```

```c
int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process processes[n];

    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Enter burst time for Process %d: ", i + 1);
        scanf("%d", &processes[i].burst_time);
        printf("Enter priority for Process %d: ", i + 1);
        scanf("%d", &processes[i].priority);
    }

    priorityScheduling(processes, n);

    return 0;
}
```

RESULT:

The Program is successfully verified.



EXPERIMENT 6

AIM:

To Construct a C program to implement pre-emptive priority scheduling algorithm.

PROCEDURE:

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

struct Process {
    int id;           // Process ID
    int burst_time;   // Burst time
    int priority;     // Priority (lower values indicate higher priority)
    int remaining_time; // Remaining time
};

// Function to perform Preemptive Priority Scheduling
void preemptivePriorityScheduling(struct Process processes[], int n) {
    int total_time = 0;
    int completed = 0;
```

```
    while (completed < n) {
        int highest_priority = -1;
        int highest_priority_idx = -1;

        // Find the process with the highest priority among the ready processes
        for (int i = 0; i < n; i++) {
            if (processes[i].burst_time > 0 && processes[i].priority < highest_priority) {
                highest_priority = processes[i].priority;
                highest_priority_idx = i;
            }
        }

        if (highest_priority_idx == -1) {
            // No process is ready to run, increase total_time
            total_time++;
        } else {
            // Execute the process with the highest priority for 1 time unit
            processes[highest_priority_idx].burst_time--;
            total_time++;

            // If the process is completed, calculate turnaround and waiting time
            if (processes[highest_priority_idx].burst_time == 0) {
                completed++;
                int turnaround_time = total_time;
                int waiting_time = turnaround_time - processes[highest_priority_idx].remaining_time;
                printf("Process %d completed (Priority %d): Turnaround Time = %d, Waiting Time =
%d\n",
                    processes[highest_priority_idx].id, processes[highest_priority_idx].priority,
turnaround_time, waiting_time);
            }
        }
    }
}
```

```
int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process processes[n];

    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Enter burst time for Process %d: ", i + 1);
        scanf("%d", &processes[i].burst_time);
        processes[i].remaining_time = processes[i].burst_time;
        printf("Enter priority for Process %d: ", i + 1);
        scanf("%d", &processes[i].priority);
    }

    preemptivePriorityScheduling(processes, n);

    return 0;
}
```

RESULT:

The Program is successfully verified.

EXPERIMENT 7

AIM:

To construct a C program to implement non-preemptive SJF algorithm.

PROCEDURE:

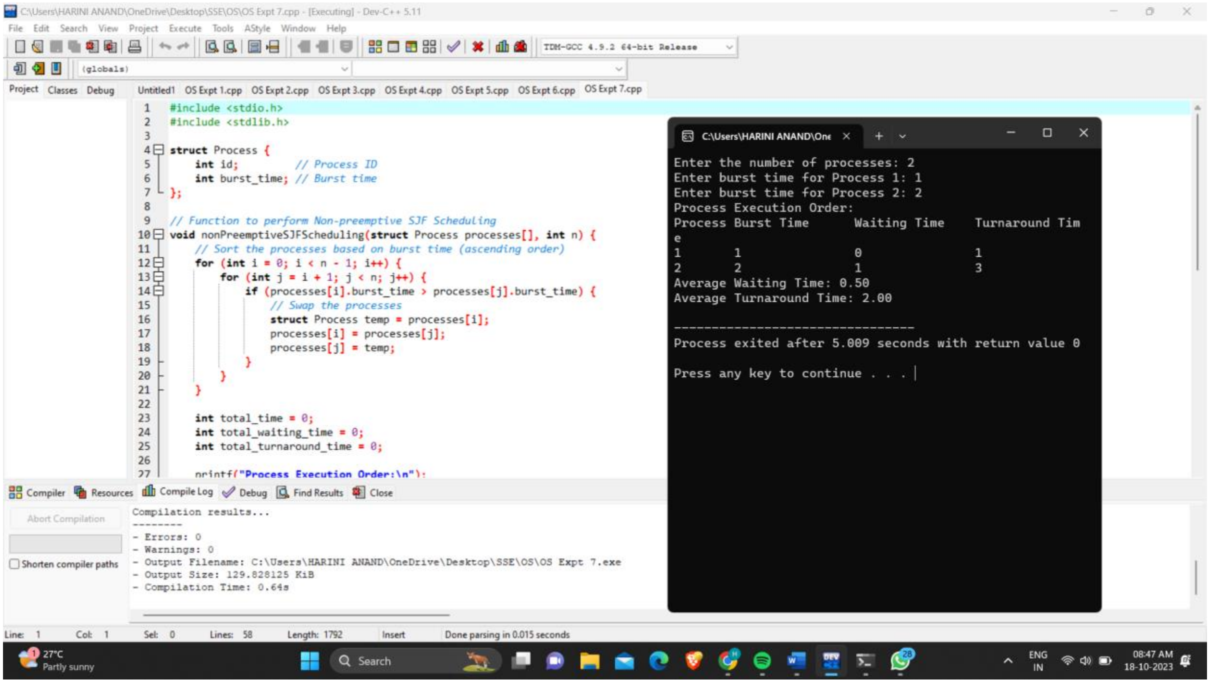
```
#include <stdio.h>
#include <stdlib.h>
```

```
struct Process {
    int id;         // Process ID
    int burst_time; // Burst time
};

// Function to perform Non-preemptive SJF Scheduling
void nonPreemptiveSJFScheduling(struct Process processes[], int n) {
    // Sort the processes based on burst time (ascending order)
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (processes[i].burst_time > processes[j].burst_time) {
                // Swap the processes
                struct Process temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }

    int total_time = 0;
    int total_waiting_time = 0;
    int total_turnaround_time = 0;

    printf("Process Execution Order:\n");
    printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");

    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", processes[i].id, processes[i].burst_time, total_time, total_time +
processes[i].burst_time);
        total_waiting_time += total_time;
        total_turnaround_time += total_time + processes[i].burst_time;
        total_time += processes[i].burst_time;
    }
```

```
    printf("Average Waiting Time: %.2f\n", (float)total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", (float)total_turnaround_time / n);
}

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process processes[n];

    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
        printf("Enter burst time for Process %d: ", i + 1);
        scanf("%d", &processes[i].burst_time);
    }

    nonPreemptiveSJFScheduling(processes, n);

    return 0;
}
```

RESULT:

The Program is successfully verified.



EXPERIMENT 8

AIM:

To Construct a C program to simulate Round Robin scheduling algorithm with C.

PROCEDURE:

```
#include <stdio.h>


// Define the maximum number of processes
#define MAX_PROCESSES 10


// Process structure to store process information
struct Process {
    int id;          // Process ID
    int burst_time;    // Burst time
    int remaining_time; // Remaining time
};


// Circular queue-like data structure to hold processes
struct Queue {
    struct Process* processes[MAX_PROCESSES];
```

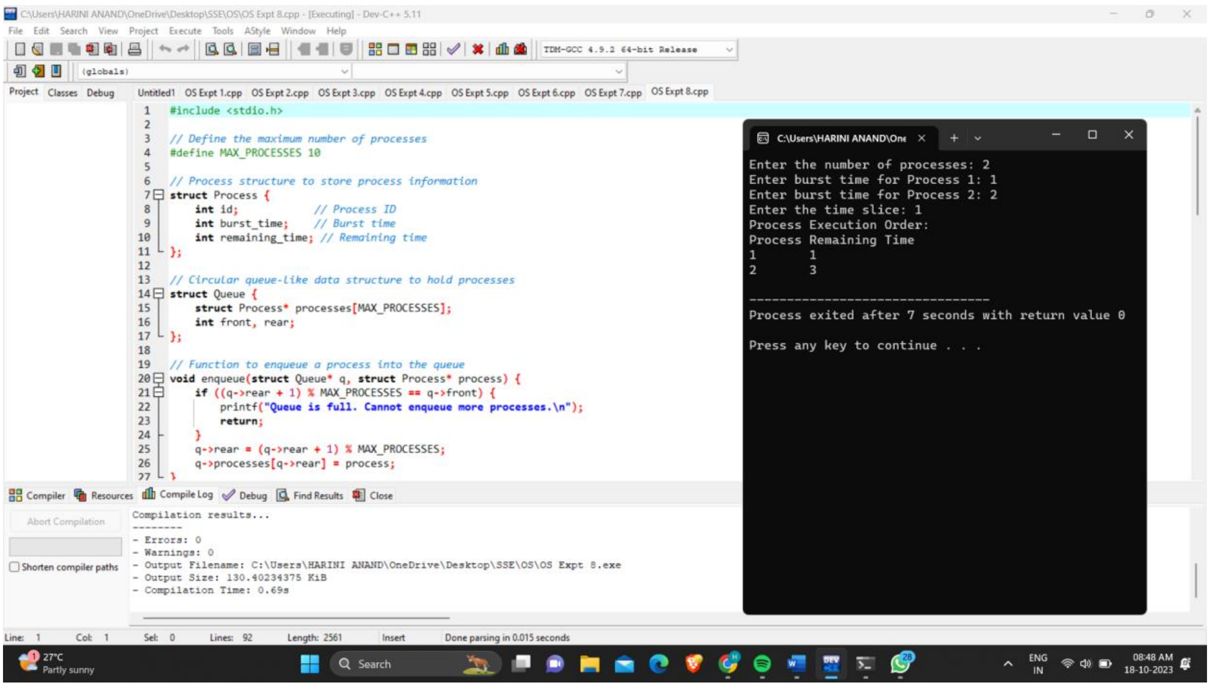
```
    int front, rear;
};

// Function to enqueue a process into the queue
void enqueue(struct Queue* q, struct Process* process) {
    if ((q->rear + 1) % MAX_PROCESSES == q->front) {
        printf("Queue is full. Cannot enqueue more processes.\n");
        return;
    }
    q->rear = (q->rear + 1) % MAX_PROCESSES;
    q->processes[q->rear] = process;
}

// Function to dequeue a process from the queue
struct Process* dequeue(struct Queue* q) {
    if (q->front == q->rear) {
        return NULL;
    }
    q->front = (q->front + 1) % MAX_PROCESSES;
    return q->processes[q->front];
}

// Function to simulate Round Robin scheduling
void roundRobinScheduling(struct Process processes[], int n, int time_slice) {
    struct Queue q;
    q.front = 0;
    q.rear = 0;

    int total_time = 0;

    // Enqueue all processes initially
    for (int i = 0; i < n; i++) {
```

```
        enqueue(&q, &processes[i]);
    }

    printf("Process Execution Order:\n");
    printf("Process\tRemaining Time\n");

    while (q.front != q.rear) {
        struct Process* current_process = dequeue(&q);

        if (current_process->remaining_time <= time_slice) {
            // Process can be completed within the time slice
            total_time += current_process->remaining_time;
            printf("%d\t%d\n", current_process->id, total_time);

        } else {
            // Process still has remaining time
            total_time += time_slice;
            current_process->remaining_time -= time_slice;
            enqueue(&q, current_process);
        }
    }
}

int main() {
    int n, time_slice;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    struct Process processes[n];

    for (int i = 0; i < n; i++) {
        processes[i].id = i + 1;
```

```
        printf("Enter burst time for Process %d: ", i + 1);
        scanf("%d", &processes[i].burst_time);
        processes[i].remaining_time = processes[i].burst_time;
    }

    printf("Enter the time slice: ");
    scanf("%d", &time_slice);

    roundRobinScheduling(processes, n, time_slice);

    return 0;
}
```

RESULT:

The Program is successfully verified.



EXPERIMENT 9

AIM:

To Illustrate the concept of inter-process communication using shared memory with a C program.

PROCEDURE:

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

#define SHM_KEY 12345 // The key to identify the shared memory segment
#define SHARED_MEMORY_SIZE sizeof(int) // Size of the shared memory segment

int main() {
    int shmid;
    int *shared_value; // Pointer to the shared integer value

    // Create a shared memory segment
    shmid = shmget(SHM_KEY, SHARED_MEMORY_SIZE, IPC_CREAT | 0666);
    if (shmid == -1) {
        perror("shmget");
        exit(1);
    }

    // Attach the shared memory segment to the process's address space
    shared_value = (int *)shmat(shmid, NULL, 0);
    if ((int)shared_value == -1) {
        perror("shmat");
        exit(1);
    }

    // Producer: Write a value to the shared memory
    printf("Producer: Enter an integer value: ");
    scanf("%d", shared_value);

    // Detach the shared memory segment
    shmdt(shared_value);
```

```
    // Create a new process for the consumer
    pid_t pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(1);
    } else if (pid == 0) {
        // This code is executed by the child process (consumer)

        // Attach the shared memory segment to the child process
        shared_value = (int *)shmat(shmid, NULL, 0);
        if ((int)shared_value == -1) {
            perror("shmat");
            exit(1);
        }

        // Consumer: Read and print the value from shared memory
        printf("Consumer: Received value: %d\n", *shared_value);

        // Detach the shared memory segment
        shmdt(shared_value);
    }

    // Wait for the child process to finish
    wait(NULL);

    // Mark the shared memory segment for removal
    shmctl(shmid, IPC_RMID, NULL);

    return 0;
}
```

RESULT:

The Program is successfully verified.

EXPERIMENT 10

AIM:

To Illustrate the concept of inter-process communication using message queue with a C program.

PROCEDURE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>

// Define the message structure
struct Message {
    long msg_type;
    char msg_text[100];
};

int main() {
    key_t key;
    int msgid;
    struct Message message;

    // Generate a unique key for the message queue
    key = ftok("msgqueue_example", 65);
    if (key == -1) {
        perror("ftok");
        exit(1);
```

```
    }

    // Create a message queue
    msgid = msgget(key, 0666 | IPC_CREAT);
    if (msgid == -1) {
        perror("msgget");
        exit(1);
    }

    // Sender: Write a message to the message queue
    message.msg_type = 1; // Message type (can be used for message filtering)
    strcpy(message.msg_text, "Hello, message queue!");
    msgsnd(msgid, &message, sizeof(message), 0);

    printf("Sender: Message sent\n");

    // Receiver: Read a message from the message queue
    msgrcv(msgid, &message, sizeof(message), 1, 0);
    printf("Receiver: Received message: %s\n", message.msg_text);

    // Remove the message queue
    msgctl(msgid, IPC_RMID, NULL);

    return 0;
}
```

RESULT:

The Program is successfully verified.

EXPERIMENT 11

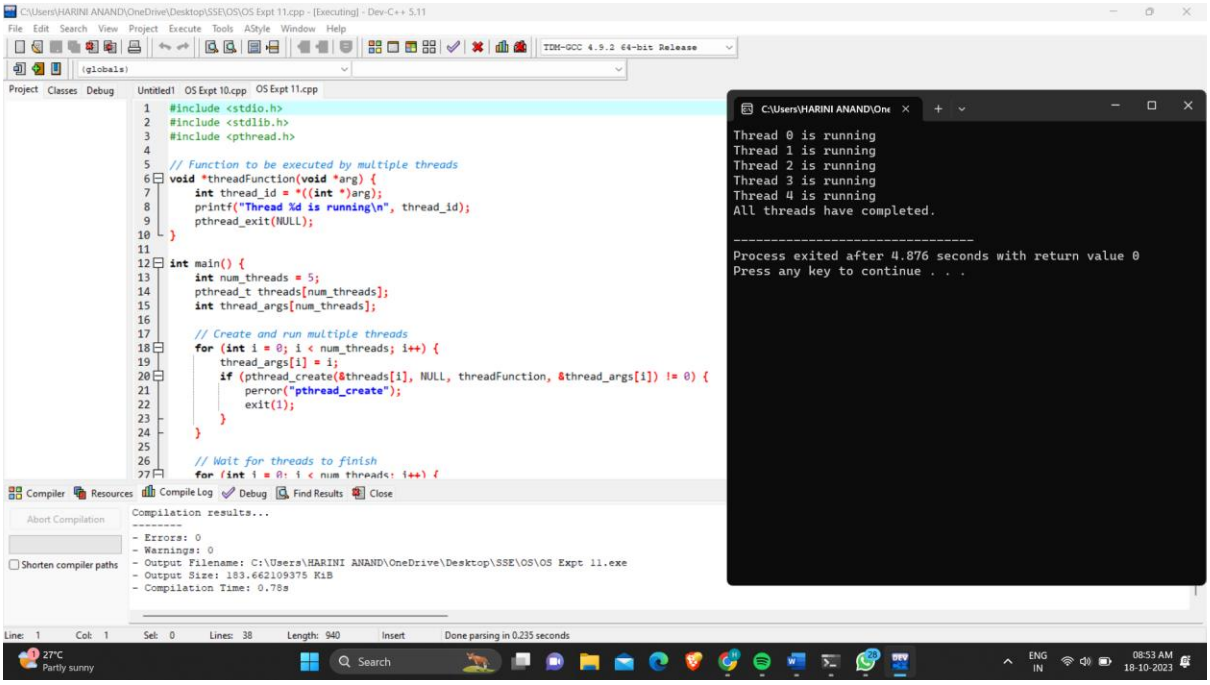
AIM:

To Illustrate the concept of multithreading using a C program.

PROCEDURE:

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

// Function to be executed by multiple threads
void *threadFunction(void *arg) {
    int thread_id = *((int *)arg);
    printf("Thread %d is running\n", thread_id);
    pthread_exit(NULL);
}

int main() {
    int num_threads = 5;
    pthread_t threads[num_threads];
    int thread_args[num_threads];

    // Create and run multiple threads
    for (int i = 0; i < num_threads; i++) {
        thread_args[i] = i;
        if (pthread_create(&threads[i], NULL, threadFunction, &thread_args[i]) != 0) {
            perror("pthread_create");
            exit(1);
        }
    }

    // Wait for threads to finish
    for (int i = 0; i < num_threads; i++) {
        if (pthread_join(threads[i], NULL) != 0) {
            perror("pthread_join");
            exit(1);
        }
    }
```

```
    printf("All threads have completed.\n");

    return 0;
}
```



RESULT:

The Program is successfully verified.

EXPERIMENT 12

AIM:

To Design a C program to simulate the concept of Dining-Philosophers problem

PROCEDURE:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define NUM_PHILOSOPHERS 5
#define EATING_TIME 1
#define THINKING_TIME 2
```

```c
pthread_mutex_t forks[NUM_PHILOSOPHERS];
pthread_t philosophers[NUM_PHILOSOPHERS];

void *philosopher(void *arg) {
    int philosopher_id = *(int *)arg;
    int left_fork = philosopher_id;
    int right_fork = (philosopher_id + 1) % NUM_PHILOSOPHERS;

    while (1) {
        // Thinking
        printf("Philosopher %d is thinking\n", philosopher_id);
        sleep(THINKING_TIME);

        // Pick up forks
        pthread_mutex_lock(&forks[left_fork]);
        pthread_mutex_lock(&forks[right_fork);

        // Eating
        printf("Philosopher %d is eating\n", philosopher_id);
        sleep(EATING_TIME);

        // Put down forks
        pthread_mutex_unlock(&forks[left_fork]);
        pthread_mutex_unlock(&forks[right_fork]);
    }
}

int main() {
    int i;
    int philosopher_ids[NUM_PHILOSOPHERS];
```

```
    // Initialize mutexes for forks
    for (i = 0; i < NUM_PHILOSOPHERS; i++) {
        pthread_mutex_init(&forks[i], NULL);
    }

    // Create philosopher threads
    for (i = 0; i < NUM_PHILOSOPHERS; i++) {
        philosopher_ids[i] = i;
        if (pthread_create(&philosophers[i], NULL, philosopher, &philosopher_ids[i]) != 0) {
            perror("pthread_create");
            exit(1);
        }
    }

    // Wait for philosopher threads to finish (which will never happen in this simulation)
    for (i = 0; i < NUM_PHILOSOPHERS; i++) {
        if (pthread_join(philosophers[i], NULL) != 0) {
            perror("pthread_join");
            exit(1);
        }
    }

    return 0;
}
```

RESULT:

The Program is successfully verified.

EXPERIMENT 13

AIM:

To Construct a C program for implementation the various memory allocation strategies.

PROCEDURE:

```
#include <stdio.h>
```

```c
#include <stdlib.h>

#define MEMORY_SIZE 100

// Structure to represent a memory block
struct MemoryBlock {
    int size;
    int allocated;
};

// Function to initialize the memory with a specified size
void initializeMemory(struct MemoryBlock memory[], int size) {
    memory[0].size = size;
    memory[0].allocated = 0;
}

// Function to display the current state of memory
void displayMemory(struct MemoryBlock memory[], int numBlocks) {
    printf("Memory Status:\n");
    for (int i = 0; i < numBlocks; i++) {
        printf("Block %d: Size = %d, Allocated = %d\n", i, memory[i].size, memory[i].allocated);
    }
    printf("\n");
}

// First Fit Allocation
int firstFit(struct MemoryBlock memory[], int numBlocks, int requestSize) {
    for (int i = 0; i < numBlocks; i++) {
        if (!memory[i].allocated && memory[i].size >= requestSize) {
            memory[i].allocated = 1;
            return i;
        }
```

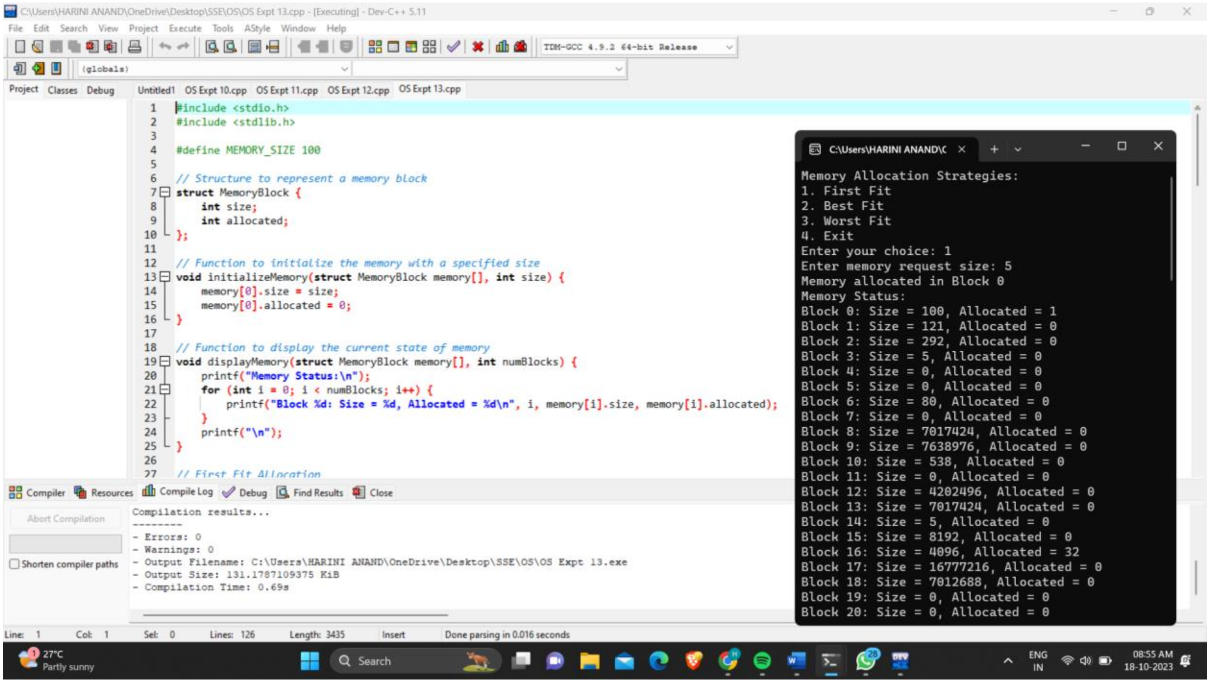
```c
    }
    return -1; // No suitable block found
}

// Best Fit Allocation
int bestFit(struct MemoryBlock memory[], int numBlocks, int requestSize) {
    int bestFitIndex = -1;
    int bestFitSize = MEMORY_SIZE;

    for (int i = 0; i < numBlocks; i++) {
        if (!memory[i].allocated && memory[i].size >= requestSize && memory[i].size < bestFitSize) {
            bestFitIndex = i;
            bestFitSize = memory[i].size;
        }
    }

    if (bestFitIndex != -1) {
        memory[bestFitIndex].allocated = 1;
    }

    return bestFitIndex;
}

// Worst Fit Allocation
int worstFit(struct MemoryBlock memory[], int numBlocks, int requestSize) {
    int worstFitIndex = -1;
    int worstFitSize = -1;

    for (int i = 0; i < numBlocks; i++) {
        if (!memory[i].allocated && memory[i].size >= requestSize && memory[i].size > worstFitSize)
{
            worstFitIndex = i;
            worstFitSize = memory[i].size;
```

```c
        }
    }

    if (worstFitIndex != -1) {
        memory[worstFitIndex].allocated = 1;
    }

    return worstFitIndex;
}

int main() {
    struct MemoryBlock memory[MEMORY_SIZE];

    int choice, requestSize;
    initializeMemory(memory, MEMORY_SIZE);

    while (1) {
        printf("Memory Allocation Strategies:\n");
        printf("1. First Fit\n");
        printf("2. Best Fit\n");
        printf("3. Worst Fit\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        if (choice == 4) {
            break;
        }

        printf("Enter memory request size: ");
        scanf("%d", &requestSize);
```

```
        int blockIndex = -1;

        switch (choice) {
            case 1:
                blockIndex = firstFit(memory, MEMORY_SIZE, requestSize);
                break;
            case 2:
                blockIndex = bestFit(memory, MEMORY_SIZE, requestSize);
                break;
            case 3:
                blockIndex = worstFit(memory, MEMORY_SIZE, requestSize);
                break;
            default:
                printf("Invalid choice.\n");
                continue;
        }

        if (blockIndex != -1) {
            printf("Memory allocated in Block %d\n", blockIndex);
        } else {
            printf("Memory allocation failed. No suitable block found.\n");
        }

        displayMemory(memory, MEMORY_SIZE);
    }

    return 0;
}
```

RESULT:

The Program is successfully verified.



EXPERIMENT 14

AIM:

To Construct a C program to organize the file using single level directory.

PROCEDURE:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define MAX_FILES 10

#define MAX_FILENAME_LENGTH 20


struct File {

    char name[MAX_FILENAME_LENGTH];

    int size;

};


struct Directory {

    struct File files[MAX_FILES];

    int num_files;
```

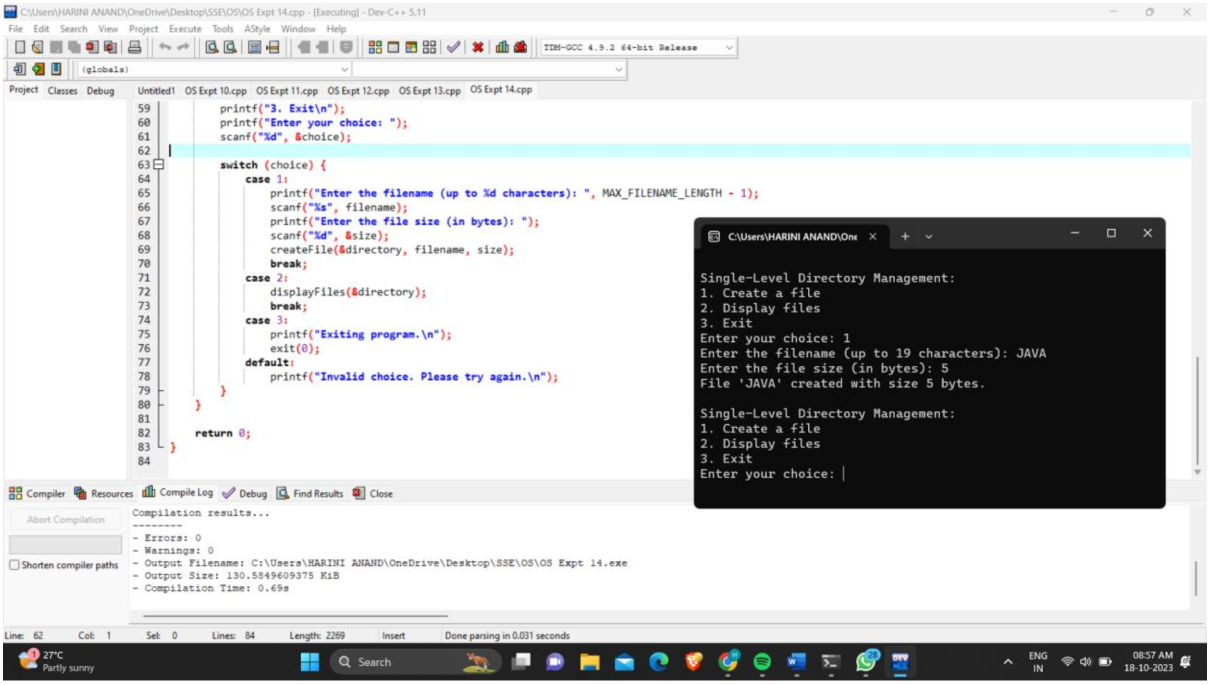
```
};

void initializeDirectory(struct Directory *dir) {
    dir->num_files = 0;
}

void createFile(struct Directory *dir, const char *filename, int size) {
    if (dir->num_files >= MAX_FILES) {
        printf("Directory is full. Cannot create more files.\n");
        return;
    }

    if (strlen(filename) >= MAX_FILENAME_LENGTH) {
        printf("Filename is too long. Maximum length is %d characters.\n",
MAX_FILENAME_LENGTH - 1);
        return;
    }

    strcpy(dir->files[dir->num_files].name, filename);
    dir->files[dir->num_files].size = size;
    dir->num_files++;

    printf("File '%s' created with size %d bytes.\n", filename, size);
}

void displayFiles(struct Directory *dir) {
    printf("Files in the directory:\n");
    for (int i = 0; i < dir->num_files; i++) {
        printf("%s (%d bytes)\n", dir->files[i].name, dir->files[i].size);
    }
}

int main() {
```

```
struct Directory directory;
initializeDirectory(&directory);

while (1) {
    int choice;
    char filename[MAX_FILENAME_LENGTH];
    int size;

    printf("\nSingle-Level Directory Management:\n");
    printf("1. Create a file\n");
    printf("2. Display files\n");
    printf("3. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("Enter the filename (up to %d characters): ", MAX_FILENAME_LENGTH - 1);
            scanf("%s", filename);
            printf("Enter the file size (in bytes): ");
            scanf("%d", &size);
            createFile(&directory, filename, size);
            break;
        case 2:
            displayFiles(&directory);
            break;
        case 3:
            printf("Exiting program.\n");
            exit(0);
        default:
            printf("Invalid choice. Please try again.\n");
    }
```

```
    }

    return 0;
}
```

RESULT:

The Program is successfully verified.



EXPERIMENT 15

AIM:

To Design a C program to organize the file using two level directory structure.

PROCEDURE:

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

#define MAX_FILES 20
#define MAX_FILENAME_LENGTH 20
#define MAX_DIRECTORIES 10
#define MAX_SUBDIRECTORIES 5
```

```
struct File {
    char name[MAX_FILENAME_LENGTH];
    int size;
};

struct Directory {
    struct File files[MAX_FILES];
    int num_files;
};

struct Subdirectory {
    char name[MAX_FILENAME_LENGTH];
    struct Directory directory;
    int used;
};

struct TwoLevelDirectory {
    struct Subdirectory subdirectories[MAX_SUBDIRECTORIES];
    int num_subdirectories;
};

void initializeDirectory(struct Directory *dir) {
    dir->num_files = 0;
}

void initializeTwoLevelDirectory(struct TwoLevelDirectory *twoLevelDir) {
    twoLevelDir->num_subdirectories = 0;
}

void createFile(struct Directory *dir, const char *filename, int size) {
    if (dir->num_files >= MAX_FILES) {
        printf("Directory is full. Cannot create more files.\n");
```

```
        return;
    }

    if (strlen(filename) >= MAX_FILENAME_LENGTH)
          {
        printf("Filename is too long. Maximum length is %d characters.\n",
MAX_FILENAME_LENGTH - 1);
                 }
```

RESULT:

The Program is successfully verified.

EXPERIMENT 16

AIM:

To Develop a C program for implementing random access file for processing the employee details.

PROCEDURE:

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

struct employee {
  int id;
  char name[20];
  char department[20];
  int salary;
};

void main() {
  FILE *fp;
  struct employee emp;
  int choice;

  // Open the file in read and write mode
```

```
fp = fopen("employees.dat", "rb+");

// Do while loop to continue the program
do {
  // Print the menu
  printf("\n1. Add employee");
  printf("\n2. Search employee");
  printf("\n3. Update employee");
  printf("\n4. Delete employee");
  printf("\n5. Exit");

  // Get the choice from the user
  printf("\nEnter your choice: ");
  scanf("%d", &choice);

  // Switch case to perform the selected operation
  switch (choice) {
    case 1:
      // Add employee
      printf("\nEnter employee id: ");
      scanf("%d", &emp.id);
      printf("\nEnter employee name: ");
      scanf("%s", emp.name);
      printf("\nEnter employee department: ");
      scanf("%s", emp.department);
      printf("\nEnter employee salary: ");
      scanf("%d", &emp.salary);

      // Write the employee data to the file
      fwrite(&emp, sizeof(emp), 1, fp);
      break;
```

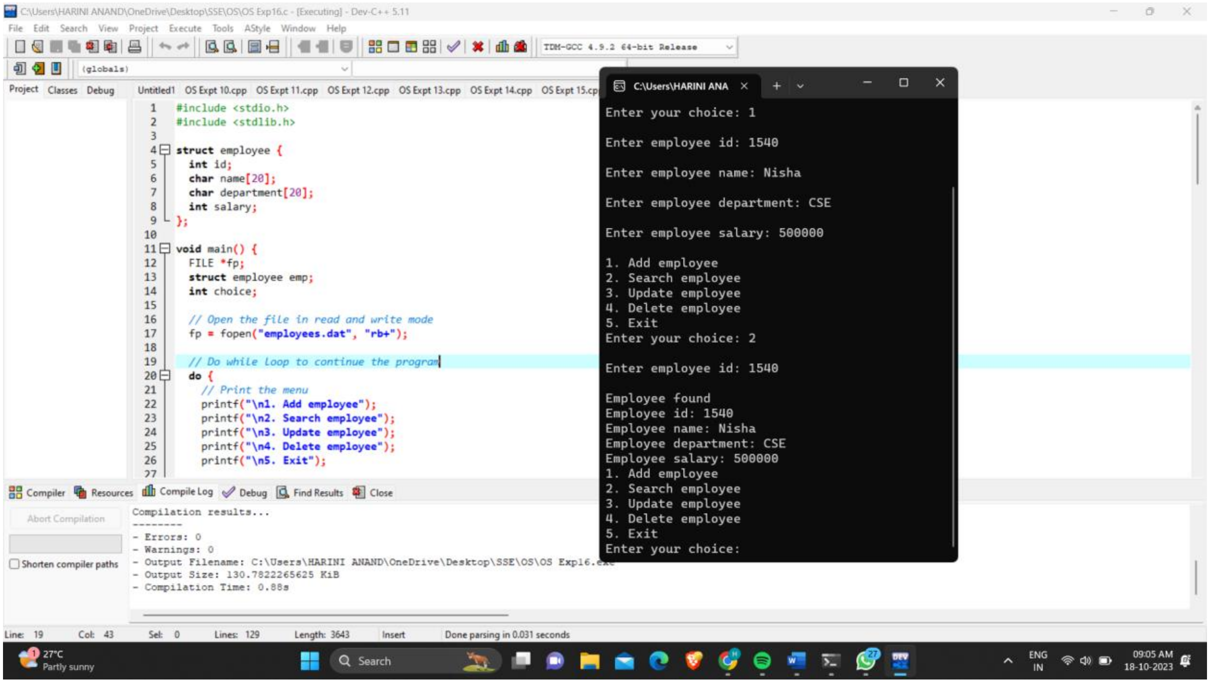
```
case 2:
  // Search employee
  printf("\nEnter employee id: ");
  scanf("%d", &emp.id);

  // Find the employee record in the file
  fseek(fp, (emp.id - 1) * sizeof(emp), SEEK_SET);
  fread(&emp, sizeof(emp), 1, fp);

  // If employee record is found, print the details
  if (emp.id != 0) {
    printf("\nEmployee found");
    printf("\nEmployee id: %d", emp.id);
    printf("\nEmployee name: %s", emp.name);
    printf("\nEmployee department: %s", emp.department);
    printf("\nEmployee salary: %d", emp.salary);
  } else {
    printf("\nEmployee not found");
  }
  break;

case 3:
  // Update employee
  printf("\nEnter employee id: ");
  scanf("%d", &emp.id);

  // Find the employee record in the file
  fseek(fp, (emp.id - 1) * sizeof(emp), SEEK_SET);
  fread(&emp, sizeof(emp), 1, fp);

  // If employee record is found, update the details
  if (emp.id != 0) {
```

```
    printf("\nEnter new employee name: ");
    scanf("%s", emp.name);
    printf("\nEnter new employee department: ");
    scanf("%s", emp.department);
    printf("\nEnter new employee salary: ");
    scanf("%d", &emp.salary);

    // Write the updated employee data to the file
    fwrite(&emp, sizeof(emp), 1, fp);
    printf("\nEmployee updated successfully");
  } else {
    printf("\nEmployee not found");
  }
  break;

case 4:
  // Delete employee
  printf("\nEnter employee id: ");
  scanf("%d", &emp.id);

  // Find the employee record in the file
  fseek(fp, (emp.id - 1) * sizeof(emp), SEEK_SET);
  fread(&emp, sizeof(emp), 1, fp);

  // If employee record is found, delete the record
  if (emp.id != 0) {
    //fseek(fp, (emp.id - 1) * sizeof(emp), SEEK_DEL);
    emp.id = 0;
    fwrite(&emp, sizeof(emp), 1, fp);
    printf("\nEmployee deleted successfully");
  } else {
    printf("\nEmployee not found");
```

```
        }
        break;

      case 5:
        // Exit the program
        break;

      default:
        // Print invalid choice message
        printf("\nInvalid choice");
        break;
    }
  } while (choice != 5);

  // Close the file
  fclose(fp);
}
```

RESULT:

The Program is successfully verified.



EXPERIMENT 17

AIM:

To Illustrate the deadlock avoidance concept by simulating Banker's algorithm with C.

PROCEDURE:

```
#include <stdio.h>

int main() {

  int n = 5;
  int m = 3;
  int available[m];
  int max[n][m];
  int allocation[n][m];
  int need[n][m];

  available[0] = 5;
  available[1] = 10;
  available[2] = 15;


  max[0][0] = 2;
  max[0][1] = 3;
  max[0][2] = 4;
  max[1][0] = 3;
  max[1][1] = 5;
  max[1][2] = 6;
  max[2][0] = 4;
  max[2][1] = 6;
  max[2][2] = 8;


  allocation[0][0] = 1;
```

```
allocation[0][1] = 2;
allocation[0][2] = 3;
allocation[1][0] = 0;
allocation[1][1] = 1;
allocation[1][2] = 2;
allocation[2][0] = 3;
allocation[2][1] = 4;
allocation[2][2] = 5;


for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {
    need[i][j] = max[i][j] - allocation[i][j];
  }
}

int isSafe = 1;
for (int i = 0; i < n; i++) {
  int k = 0;
  while (k < m) {
    if (need[i][k] > available[k]) {
      isSafe = 0;
      break;
    }
    k++;
  }
  if (!isSafe) {
    break;
  }
}
```

```
  if (isSafe) {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
        allocation[i][j] += need[i][j];
        available[j] -= need[i][j];
      }
    }
  } else {

    printf("The system is in a deadlock state.\n");
  }

  return 0;
}
```

RESULT:

The Program is successfully verified.

EXPERIMENT 18

AIM

To Construct a C program to simulate producer-consumer problem using semaphores.

PROCEDURE:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define BUFFER_SIZE 5
#define NUM_ITEMS 10

int buffer[BUFFER_SIZE];
sem_t empty, full;
```

```
void* producer(void* arg) {
    int item;
    for (int i = 0; i < NUM_ITEMS; i++) {
        item = rand() % 100; // Generate a random item
        sem_wait(&empty);    // Wait if the buffer is full
        buffer[i % BUFFER_SIZE] = item;
        printf("Produced: %d\n", item);
        sem_post(&full);     // Signal that an item has been produced
    }
    pthread_exit(NULL);
}

void* consumer(void* arg) {
    int item;
    for (int i = 0; i < NUM_ITEMS; i++) {
        sem_wait(&full);     // Wait if the buffer is empty
        item = buffer[i % BUFFER_SIZE];
        printf("Consumed: %d\n", item);
        sem_post(&empty);    // Signal that an item has been consumed
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t producer_thread, consumer_thread;

    sem_init(&empty, 0, BUFFER_SIZE); // Initialize empty semaphore to buffer size
    sem_init(&full, 0, 0);            // Initialize full semaphore to 0

    // Create producer and consumer threads
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);
```

```
    // Wait for the threads to finish
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    sem_destroy(&empty);
    sem_destroy(&full);

    return 0;
}
```

RESULT:

The Program is successfully verified.

EXPERIMENT 19

AIM:

To Design a C program to implement process synchronization using mutex locks.

PROCEDURE:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Define a global mutex
pthread_mutex_t mutex;

// Shared resource
int sharedResource = 0;

// Function for the first thread
void *thread1_function(void *arg) {
```

```
    for (int i = 0; i < 5; i++) {
        pthread_mutex_lock(&mutex); // Lock the mutex

        // Critical section: Increment the shared resource
        sharedResource++;
        printf("Thread 1: Incremented sharedResource to %d\n", sharedResource);

        pthread_mutex_unlock(&mutex); // Unlock the mutex

        // Simulate some work
        sleep(1);
    }

    return NULL;
}

// Function for the second thread
void *thread2_function(void *arg) {
    for (int i = 0; i < 5; i++) {
        pthread_mutex_lock(&mutex); // Lock the mutex

        // Critical section: Decrement the shared resource
        sharedResource--;
        printf("Thread 2: Decremented sharedResource to %d\n", sharedResource);

        pthread_mutex_unlock(&mutex); // Unlock the mutex

        // Simulate some work
        sleep(1);
    }

    return NULL;
```

```
}

int main() {
    pthread_t thread1, thread2;

    // Initialize the mutex
    if (pthread_mutex_init(&mutex, NULL) != 0) {
        printf("Mutex initialization failed.\n");
        return 1;
    }

    // Create threads
    if (pthread_create(&thread1, NULL, thread1_function, NULL) != 0 ||
        pthread_create(&thread2, NULL, thread2_function, NULL) != 0) {
        printf("Thread creation failed.\n");
        return 1;
    }

    // Wait for threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    // Destroy the mutex
    pthread_mutex_destroy(&mutex);

    printf("Both threads have completed.\n");
    return 0;
}
```

RESULT:

The Program is successfully verified.

EXPERIMENT 20

AIM:

To Construct a C program to simulate Reader-Writer problem using Semaphores.

PROCEDURE:

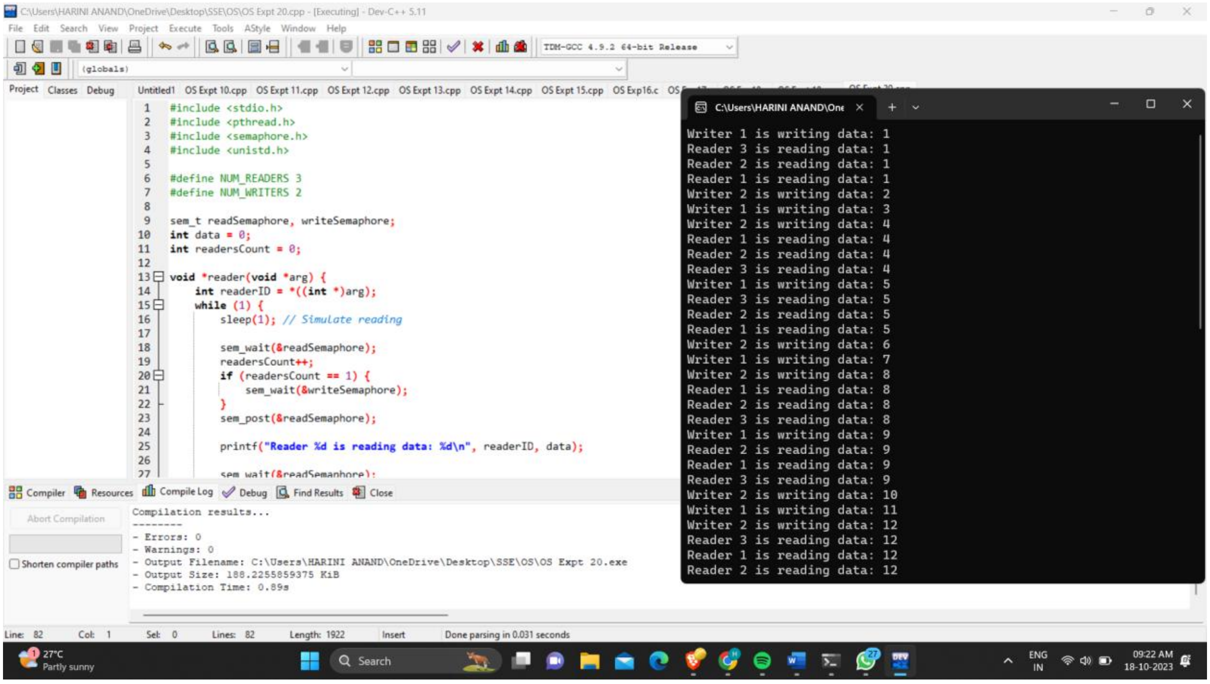
```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define NUM_READERS 3
#define NUM_WRITERS 2

sem_t readSemaphore, writeSemaphore;
int data = 0;
int readersCount = 0;

void *reader(void *arg) {
    int readerID = *((int *)arg);
    while (1) {
        sleep(1); // Simulate reading

        sem_wait(&readSemaphore);
        readersCount++;
        if (readersCount == 1) {
            sem_wait(&writeSemaphore);
        }
        sem_post(&readSemaphore);

        printf("Reader %d is reading data: %d\n", readerID, data);

        sem_wait(&readSemaphore);
        readersCount--;
        if (readersCount == 0) {
```

```
            sem_post(&writeSemaphore);
        }
        sem_post(&readSemaphore);
    }
    return NULL;
}

void *writer(void *arg) {
    int writerID = *((int *)arg);
    while (1) {
        sleep(1); // Simulate writing

        sem_wait(&writeSemaphore);
        data++;
        printf("Writer %d is writing data: %d\n", writerID, data);
        sem_post(&writeSemaphore);
    }
    return NULL;
}

int main() {
    pthread_t readers[NUM_READERS];
    pthread_t writers[NUM_WRITERS];
    int readerIDs[NUM_READERS];
    int writerIDs[NUM_WRITERS];

    sem_init(&readSemaphore, 0, 1);
    sem_init(&writeSemaphore, 0, 1);

    for (int i = 0; i < NUM_READERS; i++) {
        readerIDs[i] = i + 1;
        pthread_create(&readers[i], NULL, reader, &readerIDs[i]);
```

```
    }

    for (int i = 0; i < NUM_WRITERS; i++) {
        writerIDs[i] = i + 1;
        pthread_create(&writers[i], NULL, writer, &writerIDs[i]);
    }

    for (int i = 0; i < NUM_READERS; i++) {
        pthread_join(readers[i], NULL);
    }

    for (int i = 0; i < NUM_WRITERS; i++) {
        pthread_join(writers[i], NULL);
    }

    sem_destroy(&readSemaphore);
    sem_destroy(&writeSemaphore);

    return 0;
}
```

RESULT:

The Program is successfully verified.



EXPERIMENT 21

AIM:

To

PROCEDURE:

```
#include <stdio.h>


#define MEMORY_SIZE 100
#define MAX_PROCESS 10


int memory[MEMORY_SIZE];
int processSize[MAX_PROCESS];
int processAllocated[MAX_PROCESS];


void initializeMemory() {
    for (int i = 0; i < MEMORY_SIZE; i++) {
        memory[i] = 0; // Initialize all memory locations to 0
    }
}


void worstFit(int n) {
```

```
    for (int i = 0; i < n; i++) {
        int worstBlockSize = -1;
        int worstBlockIndex = -1;

        for (int j = 0; j < MEMORY_SIZE; j++) {
            int blockSize = 0;
            while (j < MEMORY_SIZE && memory[j] == 0) {
                blockSize++;
                j++;
            }

            if (blockSize > worstBlockSize) {
                worstBlockSize = blockSize;
                worstBlockIndex = j - blockSize;
            }
        }

        if (worstBlockIndex != -1 && worstBlockSize >= processSize[i]) {
            for (int j = worstBlockIndex; j < worstBlockIndex + processSize[i]; j++) {
                memory[j] = i + 1;
            }
            processAllocated[i] = worstBlockIndex;
        } else {
            processAllocated[i] = -1; // Process could not be allocated
        }
    }
}

void displayMemory() {
    printf("Memory Allocation:\n");
    for (int i = 0; i < MEMORY_SIZE; i++) {
        if (memory[i] == 0) {
```

```
            printf("- ");
        } else {
            printf("%d ", memory[i]);
        }
    }
    printf("\n");
}

int main() {
    int n;

    printf("Enter the number of processes: ");
    scanf("%d", &n);

    printf("Enter the size of each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d", &processSize[i]);
    }

    initializeMemory();
    worstFit(n);

    printf("\nMemory after worst fit allocation:\n");
    displayMemory();

    printf("\nProcess Allocation:\n");
    for (int i = 0; i < n; i++) {
        if (processAllocated[i] != -1) {
            printf("Process %d is allocated at position %d\n", i + 1, processAllocated[i]);
        } else {
            printf("Process %d could not be allocated\n", i + 1);
```

```
        }
    }

    return 0;
}
```

RESULT:

The Program is successfully verified.

EXPERIMENT 22

AIM:

To Construct a C program to implement best fit algorithm of memory management.

PROCEDURE:

```
#include <stdio.h>

#define MEMORY_SIZE 100
#define MAX_PROCESS 10

int memory[MEMORY_SIZE];
int processSize[MAX_PROCESS];
int processAllocated[MAX_PROCESS];

void initializeMemory() {
    for (int i = 0; i < MEMORY_SIZE; i++) {
        memory[i] = 0; // Initialize all memory locations to 0
    }
}

void bestFit(int n) {
    for (int i = 0; i < n; i++) {
        int bestBlockSize = MEMORY_SIZE + 1; // Initialize to a large value
        int bestBlockIndex = -1;
```

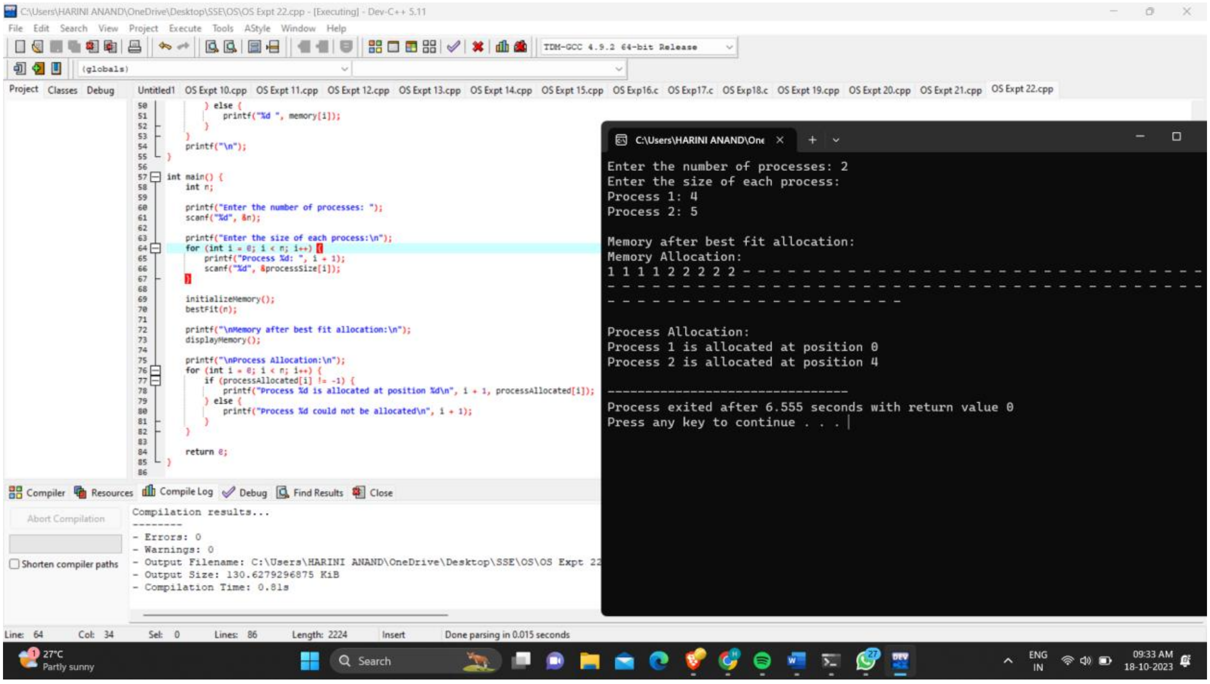
```
        for (int j = 0; j < MEMORY_SIZE; j++) {
            int blockSize = 0;
            while (j < MEMORY_SIZE && memory[j] == 0) {
                blockSize++;
                j++;
            }

            if (blockSize >= processSize[i] && blockSize < bestBlockSize) {
                bestBlockSize = blockSize;
                bestBlockIndex = j - blockSize;
            }
        }

        if (bestBlockIndex != -1) {
            for (int j = bestBlockIndex; j < bestBlockIndex + processSize[i]; j++) {
                memory[j] = i + 1;
            }
            processAllocated[i] = bestBlockIndex;
        } else {
            processAllocated[i] = -1; // Process could not be allocated
        }
    }
}

void displayMemory() {
    printf("Memory Allocation:\n");
    for (int i = 0; i < MEMORY_SIZE; i++) {
        if (memory[i] == 0) {
            printf("- ");
        } else {
            printf("%d ", memory[i]);
        }
```

```c
    }
    printf("\n");
}

int main() {
    int n;

    printf("Enter the number of processes: ");
    scanf("%d", &n);

    printf("Enter the size of each process:\n");
    for (int i = 0; i < n; i++) {
        printf("Process %d: ", i + 1);
        scanf("%d", &processSize[i]);
    }

    initializeMemory();
    bestFit(n);

    printf("\nMemory after best fit allocation:\n");
    displayMemory();

    printf("\nProcess Allocation:\n");
    for (int i = 0; i < n; i++) {
        if (processAllocated[i] != -1) {
            printf("Process %d is allocated at position %d\n", i + 1, processAllocated[i]);
        } else {
            printf("Process %d could not be allocated\n", i + 1);
        }
    }

    return 0;
```

}

RESULT:

The Program is successfully verified.



EXPERIMENT 22

AIM:

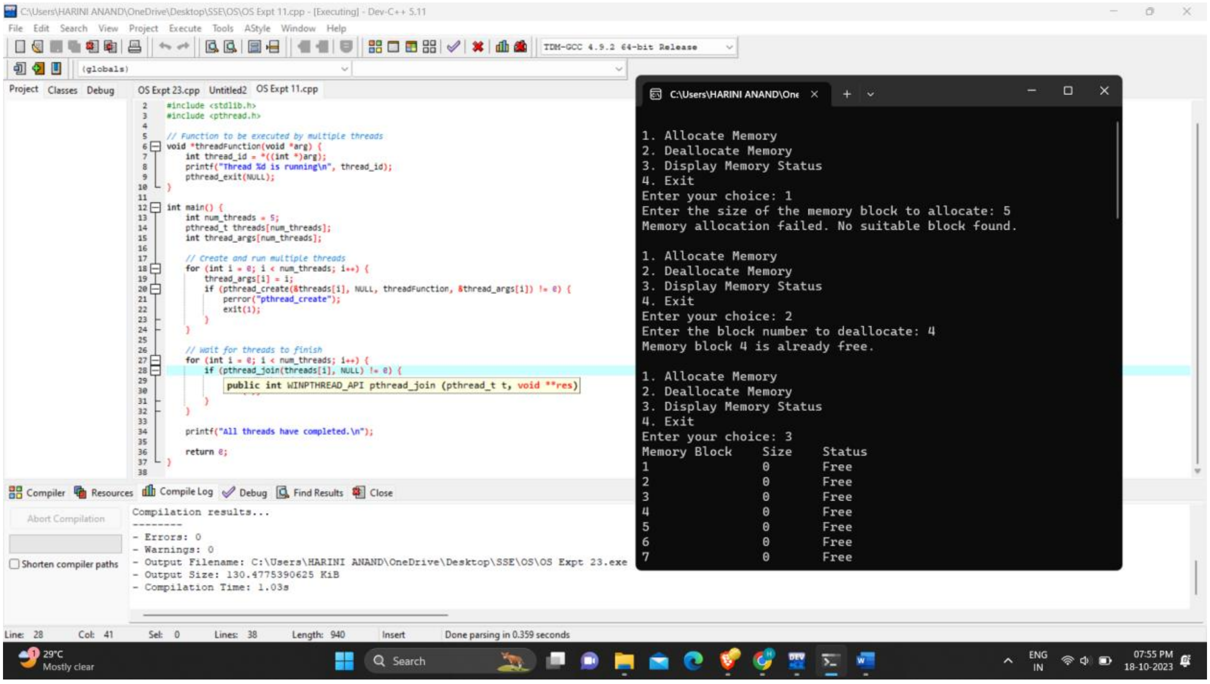
To Construct a C program to implement first fit algorithm of memory management.

PROCEDURE:

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>


// Function to be executed by multiple threads

void *threadFunction(void *arg) {

    int thread_id = *((int *)arg);

    printf("Thread %d is running\n", thread_id);

    pthread_exit(NULL);

}


int main() {
```

```c
    int num_threads = 5;
    pthread_t threads[num_threads];
    int thread_args[num_threads];

    // Create and run multiple threads
    for (int i = 0; i < num_threads; i++) {
        thread_args[i] = i;
        if (pthread_create(&threads[i], NULL, threadFunction, &thread_args[i]) != 0) {
            perror("pthread_create");
            exit(1);
        }
    }

    // Wait for threads to finish
    for (int i = 0; i < num_threads; i++) {
        if (pthread_join(threads[i], NULL) != 0) {
            perror("pthread_join");
            exit(1);
        }
    }

    printf("All threads have completed.\n");

    return 0;
}
```

RESULT:



EXPERIMENT 24

AIM:

To Design a C program to demonstrate UNIX system calls for file management.

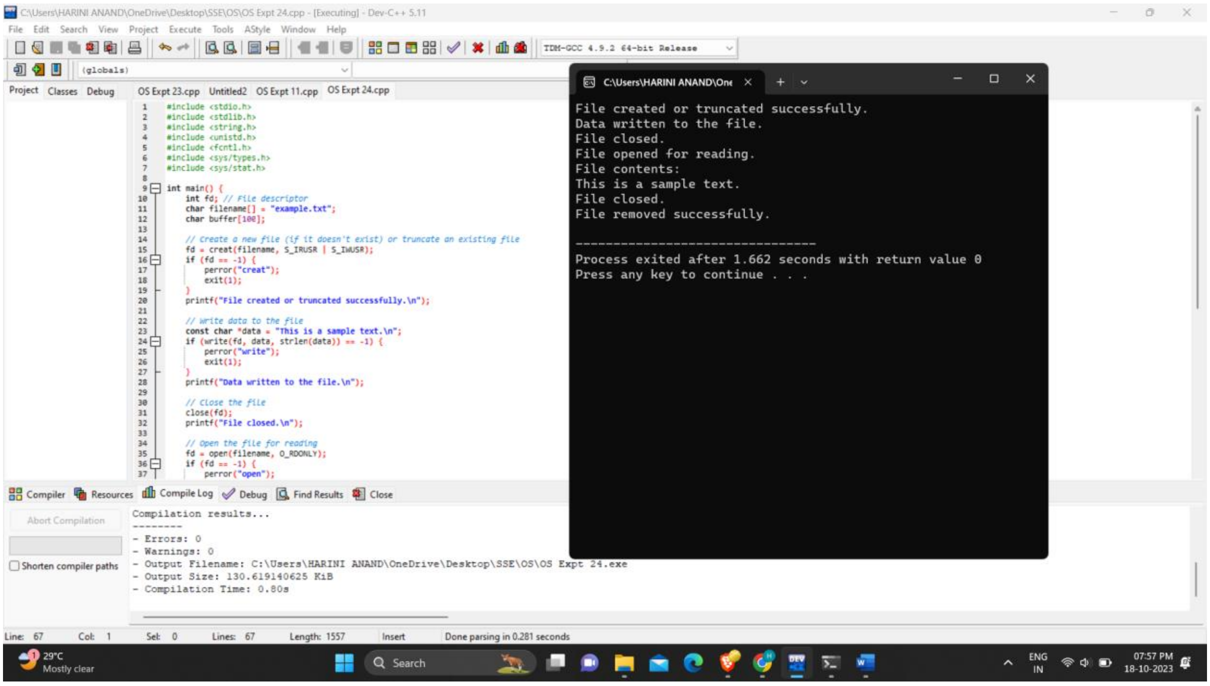
PROCEDURE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

int main() {
    int fd; // File descriptor
    char filename[] = "example.txt";
    char buffer[100];

    // Create a new file (if it doesn't exist) or truncate an existing file
    fd = creat(filename, S_IRUSR | S_IWUSR);
    if (fd == -1) {
```

```
    perror("creat");
    exit(1);
}
printf("File created or truncated successfully.\n");

// Write data to the file
const char *data = "This is a sample text.\n";
if (write(fd, data, strlen(data)) == -1) {
    perror("write");
    exit(1);
}
printf("Data written to the file.\n");

// Close the file
close(fd);
printf("File closed.\n");

// Open the file for reading
fd = open(filename, O_RDONLY);
if (fd == -1) {
    perror("open");
    exit(1);
}
printf("File opened for reading.\n");

// Read and display the file contents
ssize_t bytesRead;
printf("File contents:\n");
while ((bytesRead = read(fd, buffer, sizeof(buffer))) > 0) {
    write(STDOUT_FILENO, buffer, bytesRead);
}
```

```
    if (bytesRead == -1) {
        perror("read");
        exit(1);
    }

    // Close the file
    close(fd);
    printf("File closed.\n");

    // Remove the file
    if (remove(filename) == -1) {
        perror("remove");
        exit(1);
    }
    printf("File removed successfully.\n");

    return 0;
}
```

RESULT:



EXPERIMENT 25

AIM:

To Construct a C program to implement the I/O system calls of UNIX (fcntl, seek, stat, opendir, readdir)

PROCEDURE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>

int main() {
    // fcntl - File control
    int fd = open("example.txt", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    if (fd == -1) {
        perror("open");
        exit(1);
    }
    // Perform an operation with fcntl (for example, setting the file to non-blocking)
    int flags = fcntl(fd, F_GETFL);
    flags |= O_NONBLOCK;
    if (fcntl(fd, F_SETFL, flags) == -1) {
        perror("fcntl");
        exit(1);
    }
    close(fd);

    // lseek - File offset control
    fd = open("example.txt", O_RDWR);
```

```
if (fd == -1) {
    perror("open");
    exit(1);
}
off_t offset = lseek(fd, 10, SEEK_SET); // Move the file offset to the 10th byte
if (offset == -1) {
    perror("lseek");
    exit(1);
}
close(fd);

// stat - File status
struct stat fileStat;
if (stat("example.txt", &fileStat) == -1) {
    perror("stat");
    exit(1);
}
printf("File size: %lld bytes\n", (long long)fileStat.st_size);

// opendir and readdir - Directory operations
DIR *dir = opendir(".");
if (dir == NULL) {
    perror("opendir");
    exit(1);
}

struct dirent *entry;
printf("Files in the current directory:\n");
while ((entry = readdir(dir)) != NULL) {
    printf("%s\n", entry->d_name);
}
```

```c
    closedir(dir);

    return 0;
}
```

RESULT:

The program is Successfully verified.

EXPERIMENT 26

AIM:

To Construct a C program to implement the file management operations.

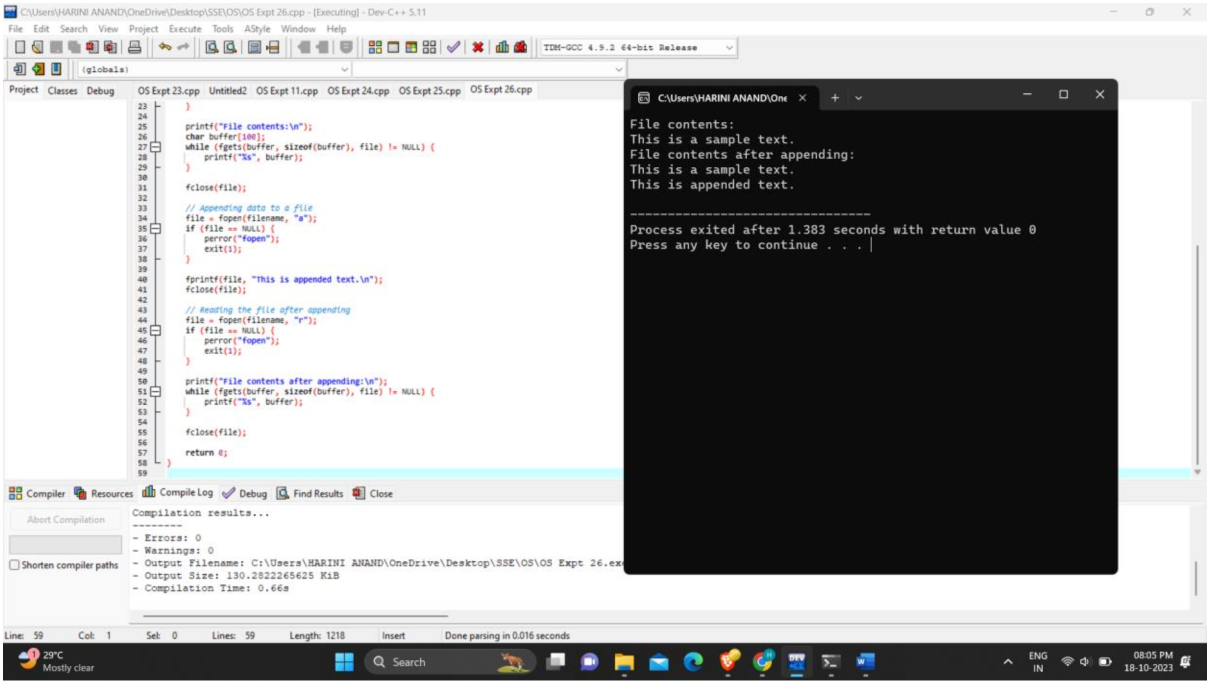
PROCEDURE:

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

int main() {
    FILE *file;
    char filename[] = "example.txt";

    // Creating and writing to a file
    file = fopen(filename, "w");
    if (file == NULL) {
        perror("fopen");
        exit(1);
    }

    fprintf(file, "This is a sample text.\n");
    fclose(file);

    // Opening and reading from a file
    file = fopen(filename, "r");
    if (file == NULL) {
        perror("fopen");
        exit(1);
```

```
}

printf("File contents:\n");
char buffer[100];
while (fgets(buffer, sizeof(buffer), file) != NULL) {
    printf("%s", buffer);
}

fclose(file);

// Appending data to a file
file = fopen(filename, "a");
if (file == NULL) {
    perror("fopen");
    exit(1);
}

fprintf(file, "This is appended text.\n");
fclose(file);

// Reading the file after appending
file = fopen(filename, "r");
if (file == NULL) {
    perror("fopen");
    exit(1);
}

printf("File contents after appending:\n");
while (fgets(buffer, sizeof(buffer), file) != NULL) {
    printf("%s", buffer);
}
```

```
    fclose(file);

    return 0;
}
```

RESULT:



The Program is Successfully verified.

EXPERIMENT 27

AIM:

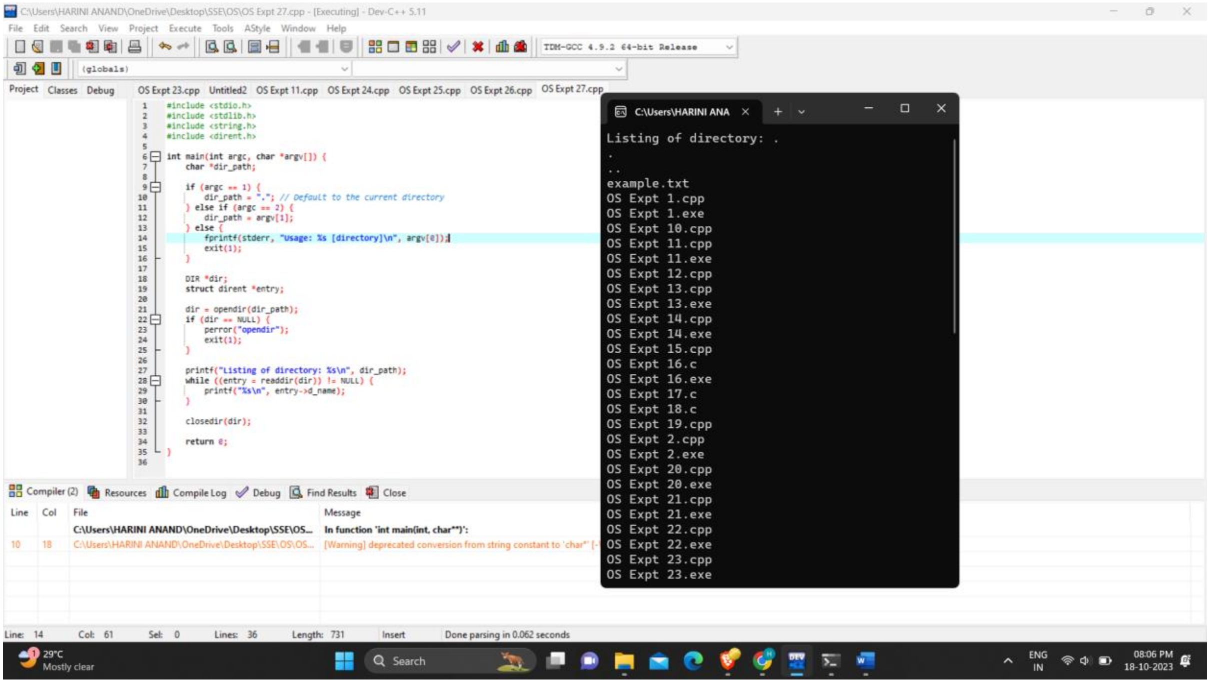
To Develop a C program for simulating the function of ls UNIX Command.

PROCEDURE:

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

int main(int argc, char *argv[]) {
    char *dir_path;

    if (argc == 1) {
        dir_path = "."; // Default to the current directory
```

```
    } else if (argc == 2) {
        dir_path = argv[1];
    } else {
        fprintf(stderr, "Usage: %s [directory]\n", argv[0]);
        exit(1);
    }

    DIR *dir;
    struct dirent *entry;

    dir = opendir(dir_path);
    if (dir == NULL) {
        perror("opendir");
        exit(1);
    }

    printf("Listing of directory: %s\n", dir_path);
    while ((entry = readdir(dir)) != NULL) {
        printf("%s\n", entry->d_name);
    }

    closedir(dir);

    return 0;
}
```

RESULT:



The Program is Successfully verified.

EXPERIMENT 28

AIM:

To Write a C program for simulation of GREP UNIX command.

PROCEDURE:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


int main(int argc, char *argv[]) {

    if (argc != 3) {

        fprintf(stderr, "Usage: %s <pattern> <filename>\n", argv[0]);

        exit(1);

    }


    char *pattern = argv[1];

    char *filename = argv[2];


    FILE *file = fopen(filename, "r");
```

```
    if (file == NULL) {
        perror("fopen");
        exit(1);
    }

    char *line = NULL;
    size_t len = 0;
    ssize_t read;

    while ((read = getline(&line, &len, file)) != -1) {
        if (strstr(line, pattern) != NULL) {
            printf("%s", line);
        }
    }

    if (line) {
        free(line);
    }

    fclose(file);
    return 0;
}
```

RESULT:

The Program is Successfully verified.

EXPERIMENT 29

AIM:

To Write a C program to simulate the solution of Classical Process Synchronization Problem

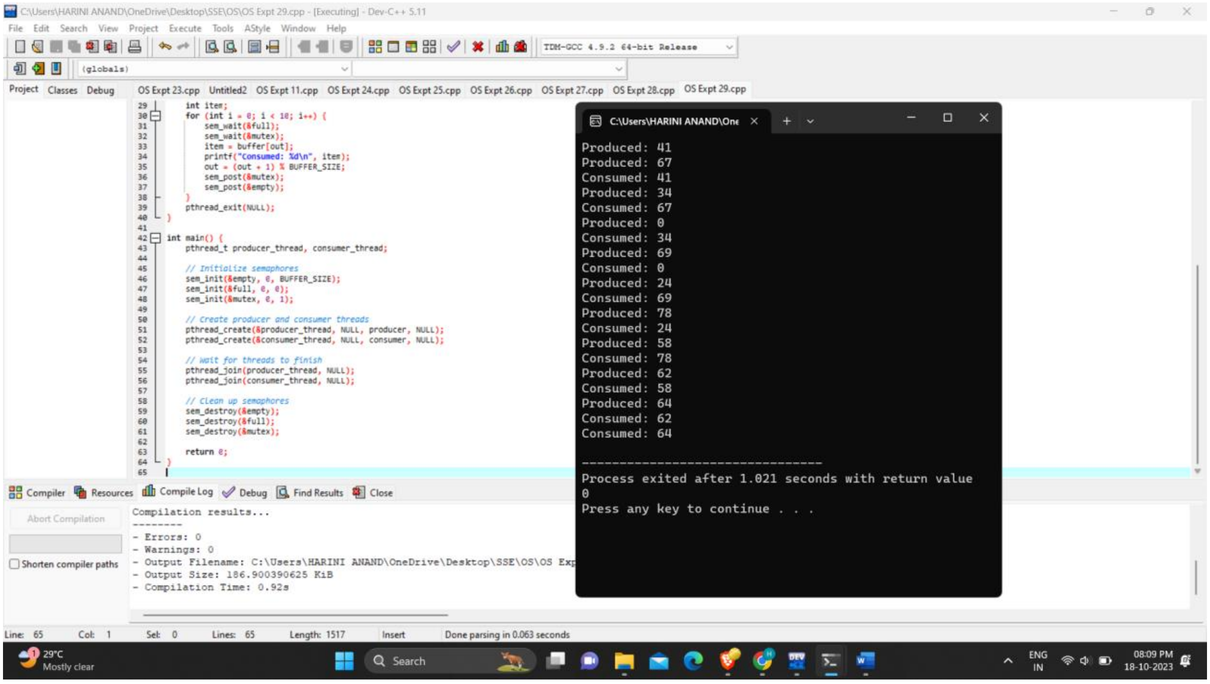
PROCEDURE:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
```

```c
#include <semaphore.h>

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int in = 0, out = 0;

sem_t empty, full, mutex;

void *producer(void *arg) {
    int item;
    for (int i = 0; i < 10; i++) {
        item = rand() % 100; // Produce an item
        sem_wait(&empty);
        sem_wait(&mutex);
        buffer[in] = item;
        printf("Produced: %d\n", item);
        in = (in + 1) % BUFFER_SIZE;
        sem_post(&mutex);
        sem_post(&full);
    }
    pthread_exit(NULL);
}

void *consumer(void *arg) {
    int item;
    for (int i = 0; i < 10; i++) {
        sem_wait(&full);
        sem_wait(&mutex);
        item = buffer[out];
        printf("Consumed: %d\n", item);
        out = (out + 1) % BUFFER_SIZE;
```

```
        sem_post(&mutex);
        sem_post(&empty);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t producer_thread, consumer_thread;

    // Initialize semaphores
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);

    // Create producer and consumer threads
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    // Wait for threads to finish
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    // Clean up semaphores
    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return 0;
}
```

RESULT:



The Program is Successfully verified.

EXPERIMENT 30

AIM:

To Write C programs to demonstrate the following thread related concepts.

(i) create (ii) join (iii) equal (iv) exit

PROCEDURE:

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <unistd.h>


void* threadFunction(void* arg) {

    int threadNumber = *(int*)arg;

    printf("Thread %d created. Thread ID: %lu\n", threadNumber, pthread_self());


    // Simulate work by sleeping for a user-specified time
```

```
    unsigned int sleepTime;
    printf("Enter sleep time for Thread %d (in seconds): ", threadNumber);
    scanf("%u", &sleepTime);
    sleep(sleepTime);

    printf("Thread %d finished.\n", threadNumber);
    pthread_exit(NULL);
}

int main() {
    int numThreads;

    printf("Enter the number of threads: ");
    scanf("%d", &numThreads);

    pthread_t threads[numThreads];
    int threadNumbers[numThreads];

    for (int i = 0; i < numThreads; i++) {
        threadNumbers[i] = i;
        if (pthread_create(&threads[i], NULL, threadFunction, &threadNumbers[i]) != 0) {
            printf("Failed to create Thread %d.\n", i);
            exit(1);
        }
    }

    for (int i = 0; i < numThreads; i++) {
        pthread_join(threads[i], NULL);
        printf("Thread %d joined and finished.\n", i);
    }

    return 0;
```

```
}
```

RESULT:

The Program is successfully verified.

EXPERIMENT 31

AIM:

To Construct a C program to simulate the First in First Out paging technique of memory management.

PROCEDURE:

```
#include <stdio.h>
#include <stdbool.h>

#define MAX_FRAMES 3

int pageQueue[MAX_FRAMES]; // Frames in memory
int pageQueueFront = 0;    // Index of the front of the page queue
int pageFaults = 0;        // Counter for page faults
int pageHits = 0;          // Counter for page hits

// Initialize the page queue with sentinel values -1
void initializePageQueue() {
    for (int i = 0; i < MAX_FRAMES; i++) {
        pageQueue[i] = -1;
    }
}

// Check if a page is already in memory
bool isPageInMemory(int page) {
    for (int i = 0; i < MAX_FRAMES; i++) {
        if (pageQueue[i] == page) {
            return true;
        }
```

```
    }
    return false;
}

// Simulate the FIFO Page Replacement algorithm
void simulateFIFO(int pages[], int numPages) {
    for (int i = 0; i < numPages; i++) {
        int page = pages[i];

        if (isPageInMemory(page)) {
            pageHits++;
        } else {
            pageQueue[pageQueueFront] = page;
            pageQueueFront = (pageQueueFront + 1) % MAX_FRAMES; // Move the front to the next
frame
            pageFaults++;
        }

        printf("Page Queue: ");
        for (int j = 0; j < MAX_FRAMES; j++) {
            printf("%d ", pageQueue[j]);
        }
        printf("\n");
    }
}

int main() {
    int numPages;

    printf("Enter the number of pages in the reference string: ");
    scanf("%d", &numPages);

    int pages[numPages];
```

```
    printf("Enter the page reference string:\n");
    for (int i = 0; i < numPages; i++) {
        scanf("%d", &pages[i]);
    }

    initializePageQueue();
    simulateFIFO(pages, numPages);

    printf("Total Page Faults: %d\n", pageFaults);
    printf("Total Page Hits: %d\n", pageHits);

    return 0;
}
```

RESULT:

The Program is successfully verified.

EXPERIMENT 32

AIM:

To Construct a C program to simulate the First in First Out paging technique of memory management.

PROCEDURE:

```
#include <stdio.h>
#include <stdbool.h>

#define MAX_FRAMES 3

int pageQueue[MAX_FRAMES]; // Frames in memory
int pageCounter = 0;       // Counter for page usage
int pageFaults = 0;        // Counter for page faults
int pageHits = 0;          // Counter for page hits
```

```
// Initialize the page queue with sentinel values -1
void initializePageQueue() {
    for (int i = 0; i < MAX_FRAMES; i++) {
        pageQueue[i] = -1;
    }
}

// Find the least recently used page to replace
int findLRUPage(int pages[]) {
    int minPageCounter = pageCounter;
    int index = 0;

    for (int i = 0; i < MAX_FRAMES; i++) {
        int page = pageQueue[i];
        int j;

        for (j = 0; j < MAX_FRAMES; j++) {
            if (page == pages[j] && pageCounter < minPageCounter) {
                minPageCounter = pageCounter;
                index = i;
                break;
            }
        }
    }

    return index;
}

// Simulate the LRU Page Replacement algorithm
void simulateLRU(int pages[], int numPages) {
    for (int i = 0; i < numPages; i++) {
        int page = pages[i];
```

```
        bool pageFound = false;
        for (int j = 0; j < MAX_FRAMES; j++) {
            if (pageQueue[j] == page) {
                pageFound = true;
                pageHits++;
                pageCounter = i;
                break;
            }
        }

        if (!pageFound) {
            int index = findLRUPage(pages); // Find the least recently used page to replace
            pageQueue[index] = page;         // Replace the page
            pageCounter = i;
            pageFaults++;
        }

        printf("Page Queue: ");
        for (int j = 0; j < MAX_FRAMES; j++) {
            printf("%d ", pageQueue[j]);
        }
        printf("\n");
    }
}

int main() {
    int numPages;

    printf("Enter the number of pages in the reference string: ");
    scanf("%d", &numPages);
```

```
    int pages[numPages];

    printf("Enter the page reference string:\n");
    for (int i = 0; i < numPages; i++) {
        scanf("%d", &pages[i]);
    }

    initializePageQueue();
    int pageCounter[numPages];

    for (int i = 0; i < numPages; i++) {
        pageCounter[i] = -1;
    }

    simulateLRU(pages, numPages);

    printf("Total Page Faults: %d\n", pageFaults);
    printf("Total Page Hits: %d\n", pageHits);

    return 0;
}
```

RESULT:

The Program is successfully verified.

EXPERIMENT 33

AIM:

To Construct a C program to simulate the optimal paging technique of memory management.

PROCEDURE:

```
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
```

```c
#define MAX_FRAMES 3

int pageQueue[MAX_FRAMES]; // Frames in memory
int pageFaults = 0;        // Counter for page faults
int pageHits = 0;          // Counter for page hits

// Initialize the page queue with a sentinel value -1
void initializePageQueue() {
    for (int i = 0; i < MAX_FRAMES; i++) {
        pageQueue[i] = -1;
    }
}

// Find the optimal page to replace
int findOptimalPage(int startIndex, int pages[], int numPages) {
    int index = -1;
    int farthest = -1;

    for (int i = 0; i < MAX_FRAMES; i++) {
        int j;
        for (j = startIndex; j < numPages; j++) {
            if (pageQueue[i] == pages[j]) {
                if (j > farthest) {
                    farthest = j;
                    index = i;
                }
                break;
            }
        }
        if (j == numPages) {
            return i; // The page is not used anymore, so replace it
        }
```

```c
    }

    if (index == -1) {
        index = 0;
    }

    return index;
}

// Simulate the Optimal Page Replacement algorithm
void simulateOptimal(int pages[], int numPages) {
    for (int i = 0; i < numPages; i++) {
        int page = pages[i];

        bool pageFound = false;
        for (int j = 0; j < MAX_FRAMES; j++) {
            if (pageQueue[j] == page) {
                pageFound = true;
                pageHits++;
                break;
            }
        }

        if (!pageFound) {
            int index = findOptimalPage(i + 1, pages, numPages); // Find the page to replace
            pageQueue[index] = page;                    // Replace the page
            pageFaults++;
        }

        printf("Page Queue: ");
        for (int j = 0; j < MAX_FRAMES; j++) {
            printf("%d ", pageQueue[j]);
```

```
        }
        printf("\n");
    }
}

int main() {
    int numPages;

    printf("Enter the number of pages in the reference string: ");
    scanf("%d", &numPages);

    int pages[numPages];

    printf("Enter the page reference string:\n");
    for (int i = 0; i < numPages; i++) {
        scanf("%d", &pages[i]);
    }

    initializePageQueue();
    simulateOptimal(pages, numPages);

    printf("Total Page Faults: %d\n", pageFaults);
    printf("Total Page Hits: %d\n", pageHits);

    return 0;
}
```

RESULT:

The Program is successfully verified.

EXPERIMENT 34

AIM:

To Consider a file system where the records of the file are stored one after another both physically and logically. A record of the file can only be accessed by reading all the previous records. Design a C program to simulate the file allocation strategy.

PROCEDURE:

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

#define MAX_RECORDS 100

// Data structure for a file
struct File {
    char name[100]; // File name
    char records[MAX_RECORDS][100]; // Array to store records
    int numRecords; // Number of records in the file
};

struct File files[10]; // Array to hold files (up to 10 files)

// Create a new file
int createFile(char name[]) {
    for (int i = 0; i < 10; i++) {
        if (strlen(files[i].name) == 0) {
            strcpy(files[i].name, name);
            files[i].numRecords = 0;
            return i;
        }
    }
    return -1; // No free slots for a new file
}

// Add a record to a file
```

```c
int addRecord(int fileIndex, char record[]) {
    struct File* file = &files[fileIndex];
    if (file->numRecords < MAX_RECORDS) {
        strcpy(file->records[file->numRecords], record);
        file->numRecords++;
        return 1;
    }
    return 0; // No space for a new record
}

// Display all records in a file
void displayFileRecords(int fileIndex) {
    struct File* file = &files[fileIndex];
    printf("File: %s\n", file->name);
    printf("Records:\n");
    for (int i = 0; i < file->numRecords; i++) {
        printf("%s\n", file->records[i]);
    }
}

int main() {
    int choice, fileIndex;
    char name[100], record[100];

    while (1) {
        printf("1. Create a file\n2. Add a record\n3. Display file records\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the file name: ");
```

```
            scanf("%s", name);
            fileIndex = createFile(name);
            if (fileIndex != -1) {
                printf("File created with index %d\n", fileIndex);
            } else {
                printf("No free slots for a new file\n");
            }
            break;
        case 2:
            printf("Enter the file index: ");
            scanf("%d", &fileIndex);
            printf("Enter the record: ");
            scanf("%s", record);
            if (addRecord(fileIndex, record)) {
                printf("Record added to the file\n");
            } else {
                printf("No space for a new record\n");
            }
            break;
        case 3:
            printf("Enter the file index: ");
            scanf("%d", &fileIndex);
            displayFileRecords(fileIndex);
            break;
        case 4:
            exit(0);
        default:
            printf("Invalid choice\n");
    }
}

return 0;
```

```
}
```

RESULT:

The Program is successfully verified.

EXPERIMENT 35

AIM:

To Consider a file system that brings all the file pointers together into an index block. The ith entry in the index block points to the ith block of the file. Design a C program to simulate the file allocation strategy.

PROCEDURE:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_BLOCKS 100

// Data structure for a disk block
struct DiskBlock {
    int data; // Data stored in the block
};

// Data structure for an index block
struct IndexBlock {
    int blockPointers[MAX_BLOCKS]; // Pointers to file blocks
    int numBlocks;                 // Number of blocks in the file
};

// Data structure for a file
struct File {
    char name[100];           // File name
    struct IndexBlock index;   // Index block for the file
};

struct DiskBlock disk[MAX_BLOCKS];  // Simulated disk blocks
```

```
struct File files[10];              // Array to hold files (up to 10 files)

// Initialize the disk blocks
void initializeDisk() {
    for (int i = 0; i < MAX_BLOCKS; i++) {
        disk[i].data = -1; // Initialize data to -1 to represent empty block
    }
}

// Create a new file
int createFile(char name[]) {
    for (int i = 0; i < 10; i++) {
        if (files[i].index.numBlocks == 0) {
            struct File newFile;
             strcpy(newFile.name, name);
            newFile.index.numBlocks = 0;
            files[i] = newFile;
            return i;
        }
    }
    return -1;  // No free slots for a new file
}

// Allocate a block to a file
void allocateBlock(int fileIndex, int blockData) {
    struct IndexBlock* index = &files[fileIndex].index;
    if (index->numBlocks < MAX_BLOCKS) {
        index->blockPointers[index->numBlocks] = blockData;
        index->numBlocks++;
    } else {
        printf("File is full. Cannot allocate more blocks.\n");
    }
```

```
}

// Display the blocks allocated to a file
void displayFileBlocks(int fileIndex) {
    struct IndexBlock* index = &files[fileIndex].index;
    printf("File: %s\n", files[fileIndex].name);
    printf("Blocks allocated: ");
    for (int i = 0; i < index->numBlocks; i++) {
        printf("%d -> ", index->blockPointers[i]);
    }
    printf("End of file.\n");
}

int main() {
    initializeDisk();

    int choice, fileIndex, blockData;
    char name[100];

    while (1) {
        printf("1. Create a file\n2. Allocate a block\n3. Display file blocks\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the file name: ");
                scanf("%s", name);
                fileIndex = createFile(name);
                if (fileIndex != -1) {
                    printf("File created with index %d\n", fileIndex);
                } else {
```

```
                printf("No free slots for a new file\n");
            }
            break;
        case 2:
            printf("Enter the file index: ");
            scanf("%d", &fileIndex);
            printf("Enter the block data: ");
            scanf("%d", &blockData);
            allocateBlock(fileIndex, blockData);
            break;
        case 3:
            printf("Enter the file index: ");
            scanf("%d", &fileIndex);
            displayFileBlocks(fileIndex);
            break;
        case 4:
            exit(0);
        default:
            printf("Invalid choice\n");
        }
    }

    return 0;
}
```

RESULT:

The Program is successfully verified.

EXPERIMENT 36

AIM:

To With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block. Design a C program to simulate the file allocation strategy.

PROCEDURE:

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

#define MAX_BLOCKS 100

// Data structure for a disk block
struct DiskBlock {
    int data;              // Data stored in the block
    struct DiskBlock* next;  // Pointer to the next block
};

// Data structure for a file
struct File {
    char name[100];          // File name
    struct DiskBlock* first;   // Pointer to the first block
    struct DiskBlock* last;    // Pointer to the last block
};

struct DiskBlock disk[MAX_BLOCKS];  // Simulated disk blocks
struct File files[10];             // Array to hold files (up to 10 files)

// Initialize the disk blocks
void initializeDisk() {
    for (int i = 0; i < MAX_BLOCKS; i++) {
        disk[i].data = -1; // Initialize data to -1 to represent empty block
        disk[i].next = NULL;
    }
}
```

```
// Create a new file
int createFile(char name[]) {
    for (int i = 0; i < 10; i++) {
        if (files[i].first == NULL) {
            struct File newFile;
            strcpy(newFile.name, name);
            newFile.first = NULL;
            newFile.last = NULL;
            files[i] = newFile;
            return i;
        }
    }
    return -1;  // No free slots for a new file
}

// Allocate a block to a file
void allocateBlock(int fileIndex, int blockData) {
    struct DiskBlock* newBlock = &disk[blockData];
    if (files[fileIndex].first == NULL) {
        files[fileIndex].first = newBlock;
        files[fileIndex].last = newBlock;
    } else {
        files[fileIndex].last->next = newBlock;
        files[fileIndex].last = newBlock;
    }
    newBlock->next = NULL;
    newBlock->data = blockData;
}

// Display the blocks allocated to a file
void displayFileBlocks(int fileIndex) {
    struct DiskBlock* current = files[fileIndex].first;
```

```
    printf("File: %s\n", files[fileIndex].name);
    printf("Blocks allocated: ");
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("End of file.\n");
}

int main() {
    initializeDisk();

    int choice, fileIndex, blockData;
    char name[100];

    while (1) {
        printf("1. Create a file\n2. Allocate a block\n3. Display file blocks\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the file name: ");
                scanf("%s", name);
                fileIndex = createFile(name);
                if (fileIndex != -1) {
                    printf("File created with index %d\n", fileIndex);
                } else {
                    printf("No free slots for a new file\n");
                }
                break;
            case 2:
```

```
            printf("Enter the file index: ");
            scanf("%d", &fileIndex);
            printf("Enter the block data: ");
            scanf("%d", &blockData);
            allocateBlock(fileIndex, blockData);
            break;
          case 3:
            printf("Enter the file index: ");
            scanf("%d", &fileIndex);
            displayFileBlocks(fileIndex);
            break;
          case 4:
            exit(0);
          default:
            printf("Invalid choice\n");
        }
    }

    return 0;
}
```

RESULT:

The Program is successfully verified.

EXPERIMENT 37

AIM:

To Construct a C program to simulate the First Come First Served disk scheduling algorithm.

PROCEDURE:

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

#define MAX_BLOCKS 100
```

```
// Data structure for a disk block
struct DiskBlock {
    int data;                // Data stored in the block
    struct DiskBlock* next;  // Pointer to the next block
};

// Data structure for a file
struct File {
    char name[100];           // File name
    struct DiskBlock* first;   // Pointer to the first block
    struct DiskBlock* last;    // Pointer to the last block
};

struct DiskBlock disk[MAX_BLOCKS];  // Simulated disk blocks
struct File files[10];              // Array to hold files (up to 10 files)

// Initialize the disk blocks
void initializeDisk() {
    for (int i = 0; i < MAX_BLOCKS; i++) {
        disk[i].data = -1; // Initialize data to -1 to represent empty block
        disk[i].next = NULL;
    }
}

// Create a new file
int createFile(char name[]) {
    for (int i = 0; i < 10; i++) {
        if (files[i].first == NULL) {
            struct File newFile;
            strcpy(newFile.name, name);
            newFile.first = NULL;
```

```
            newFile.last = NULL;
            files[i] = newFile;
            return i;
        }
    }
    return -1;  // No free slots for a new file
}

// Allocate a block to a file
void allocateBlock(int fileIndex, int blockData) {
    struct DiskBlock* newBlock = &disk[blockData];
    if (files[fileIndex].first == NULL) {
        files[fileIndex].first = newBlock;
        files[fileIndex].last = newBlock;
    } else {
        files[fileIndex].last->next = newBlock;
        files[fileIndex].last = newBlock;
    }
    newBlock->next = NULL;
    newBlock->data = blockData;
}

// Display the blocks allocated to a file
void displayFileBlocks(int fileIndex) {
    struct DiskBlock* current = files[fileIndex].first;
    printf("File: %s\n", files[fileIndex].name);
    printf("Blocks allocated: ");
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("End of file.\n");
```

```
}

int main() {
    initializeDisk();

    int choice, fileIndex, blockData;
    char name[100];

    while (1) {
        printf("1. Create a file\n2. Allocate a block\n3. Display file blocks\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the file name: ");
                scanf("%s", name);
                fileIndex = createFile(name);
                if (fileIndex != -1) {
                    printf("File created with index %d\n", fileIndex);
                } else {
                    printf("No free slots for a new file\n");
                }
                break;
            case 2:
                printf("Enter the file index: ");
                scanf("%d", &fileIndex);
                printf("Enter the block data: ");
                scanf("%d", &blockData);
                allocateBlock(fileIndex, blockData);
                break;
            case 3:
```

```
            printf("Enter the file index: ");
            scanf("%d", &fileIndex);
            displayFileBlocks(fileIndex);
            break;
          case 4:
            exit(0);
          default:
            printf("Invalid choice\n");
        }
    }

    return 0;
}
```

RESULT:

The Program is successfully verified.

EXPERIMENT 38

AIM:

To Design a C program to simulate SCAN disk scheduling algorithm.

PROCEDURE:

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

#define SIZE 100

void scan(int requests[], int n, int head, int direction) {
    int seek_sequence[SIZE];
    int seek_count = 0;

    int left = 0, right = SIZE - 1;

    if (direction == 1) {
```

```
        // Find the rightmost request
        for (int i = 0; i < n; i++) {
            if (requests[i] > head && requests[i] <= right)
                right = requests[i];
        }
    } else {
        // Find the leftmost request
        for (int i = 0; i < n; i++) {
            if (requests[i] < head && requests[i] >= left)
                left = requests[i];
        }
    }

    // Scan in the specified direction
    while (1) {
        if (direction == 1) {
            for (int i = head; i <= right; i++) {
                seek_sequence[seek_count++] = i;
            }
            head = right;
            // No more requests in the right direction
            if (seek_count >= n)
                break;
            // Move to the end
            seek_sequence[seek_count++] = SIZE - 1;
            direction = 0;
        } else {
            for (int i = head; i >= left; i--) {
                seek_sequence[seek_count++] = i;
            }
            head = left;
            // No more requests in the left direction
```

```
            if (seek_count >= n)
                break;
            // Move to the beginning
            seek_sequence[seek_count++] = 0;
            direction = 1;
        }
    }

    printf("Seek Sequence: ");
    for (int i = 0; i < seek_count; i++) {
        printf("%d ", seek_sequence[i]);
    }

    printf("\nTotal Seek Time = %d\n", seek_count);
}

int main() {
    int requests[SIZE];
    int n, head, direction;

    printf("Enter the number of requests: ");
    scanf("%d", &n);

    if (n > SIZE || n <= 0) {
        printf("Invalid number of requests.\n");
        return 1;
    }

    printf("Enter the request queue:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
    }
```

```
    printf("Enter the initial head position: ");
    scanf("%d", &head);

    if (head < 0 || head >= SIZE) {
        printf("Invalid head position.\n");
        return 1;
    }

    printf("Enter the direction (1 for right, 0 for left): ");
    scanf("%d", &direction);

    if (direction != 0 && direction != 1) {
        printf("Invalid direction.\n");
        return 1;
    }

    scan(requests, n, head, direction);

    return 0;
}
```

RESULT:

EXPERIMENT 39

AIM:

To Develop a C program to simulate C-SCAN disk scheduling algorithm.

PROCEDURE:

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

#define SIZE 100

void cscan(int requests[], int n, int head, int direction) {
```

```
int seek_sequence[SIZE];
int seek_count = 0;

// Sort the request array in ascending order
for (int i = 0; i < n - 1; i++) {
    for (int j = i + 1; j < n; j++) {
        if (requests[i] > requests[j]) {
            int temp = requests[i];
            requests[i] = requests[j];
            requests[j] = temp;
        }
    }
}

int start, end;
if (direction == 1) {
    start = head;
    end = SIZE - 1;
} else {
    start = 0;
    end = head;
}

// Move the head to the end of the disk in the specified direction
for (int i = start; i <= end; i++) {
    seek_sequence[seek_count++] = requests[i];
}

if (direction == 1) {
    seek_sequence[seek_count++] = SIZE - 1;
    seek_sequence[seek_count++] = 0;
} else {
```

```
        seek_sequence[seek_count++] = 0;
        seek_sequence[seek_count++] = SIZE - 1;
    }

    // Move the head to the beginning of the disk in the specified direction
    for (int i = end; i >= start; i--) {
        seek_sequence[seek_count++] = requests[i];
    }

    printf("Seek Sequence: ");
    for (int i = 0; i < seek_count; i++) {
        printf("%d ", seek_sequence[i]);
    }

    printf("\nTotal Seek Time = %d\n", seek_count);
}

int main() {
    int requests[SIZE];
    int n, head, direction;

    printf("Enter the number of requests: ");
    scanf("%d", &n);

    if (n > SIZE || n <= 0) {
        printf("Invalid number of requests.\n");
        return 1;
    }

    printf("Enter the request queue:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &requests[i]);
```

```
    }

    printf("Enter the initial head position: ");
    scanf("%d", &head);

    if (head < 0 || head >= SIZE) {
        printf("Invalid head position.\n");
        return 1;
    }

    printf("Enter the direction (1 for right, 0 for left): ");
    scanf("%d", &direction);

    if (direction != 0 && direction != 1) {
        printf("Invalid direction.\n");
        return 1;
    }
    cscan(requests, n, head, direction);

    return 0;
}
```

RESULT:

The Program is successfully verified.

EXPERIMENT 40

AIM:

To Illustrate the various File Access Permission and different types users in Linux.

PROCEDURE:

```
#include <stdio.h>
#include <sys/stat.h>
```

```
int main() {
    // Define the file path and the desired permissions
    const char *filepath = "example.txt";
    mode_t permissions = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH; // Read and write for owner,
read for group and others

    // Set the file permissions using chmod
    if (chmod(filepath, permissions) == 0) {
        printf("File permissions set successfully.\n");
    } else {
        perror("chmod");
        return 1;
    }

    // Display the file permissions
    struct stat fileStat;
    if (stat(filepath, &fileStat) == 0) {
        printf("File permissions: %o\n", fileStat.st_mode & 0777);
    } else {
        perror("stat");
        return 1;
    }

    return 0;
}
```

RESULT:

The Program is successfully verified.