TITLE: Process /Thread Creation and Termination

THEORY:

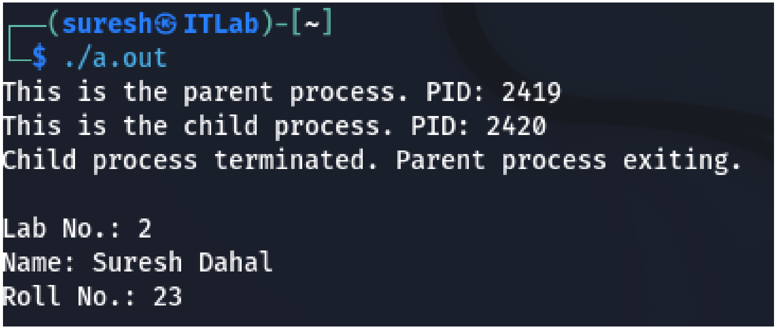
Process creation in Linux uses fork() to create a new process, and exit() terminates it. Threads are created using pthread_create() and terminated using pthread_exit(). These concepts allow multitasking and efficient resource management in the operating system.

a. WAP in C to demonstrate the process creation and termination in Linux.

Program:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();
    if (pid < 0) {
        printf("Fork failed\n");
        return 1;
    }
    else if (pid == 0) {
        printf("This is the child process. PID: %d\n", getpid());
        exit(0);
    }
    else {
        printf("This is the parent process. PID: %d\n", getpid());
        wait(NULL);  // Wait for child process to terminate
        printf("Child process terminated. Parent process exiting.\n");
    }
    printf("\nLab No.: 2\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No.: 23\n");

    return 0;
}
```

Output:



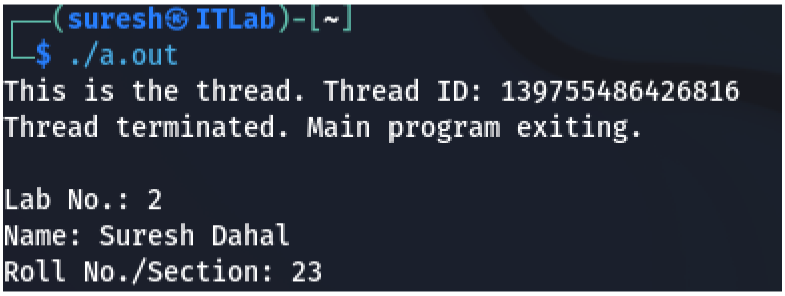
```
┌──(suresh㉿ITLab)-[~]
└─$ ./a.out
This is the parent process. PID: 2419
This is the child process. PID: 2420
Child process terminated. Parent process exiting.

Lab No.: 2
Name: Suresh Dahal
Roll No.: 23
```

b. **WAP in C to demonstrate the thread creation and termination in Linux.**

Program

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

void* threadFunction(void* arg) {
    printf("This is the thread. Thread ID: %ld\n", pthread_self());
    pthread_exit(NULL);
}

int main() {
    pthread_t thread;

    if (pthread_create(&thread, NULL, threadFunction, NULL) != 0) {
        printf("Thread creation failed\n");
        return 1;
    }

    pthread_join(thread, NULL);
    printf("Thread terminated. Main program exiting.\n");

    printf("\nLab No.: 2\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No.: 23\n");

    return 0;
}
```

Output



```
┌──(suresh㉿ITLab)-[~]
└─$ ./a.out
This is the thread. Thread ID: 139755486426816
Thread terminated. Main program exiting.

Lab No.: 2
Name: Suresh Dahal
Roll No./Section: 23
```

TITLE: Simulation of IPC Techniques

THEORY:

Inter-Process Communication (IPC) allows processes to exchange data and synchronize execution. Common IPC techniques include **shared memory** and **message passing**.

- **Shared Memory:** Multiple processes access a common memory segment for fast communication. Synchronization mechanisms like semaphores prevent data conflicts.

- **Message Passing:** Processes communicate by sending and receiving messages using system calls like msgsnd() and msgrcv(), ensuring controlled data exchange.

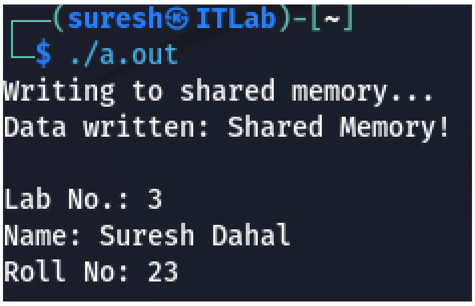
These techniques are essential for coordinating tasks in multi-process systems.

a. WAP in C to simulate shared memory concept for IPC.

Program

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

#define SHM_SIZE 1024  // Shared memory size

int main() {
    key_t key = ftok("shmfile", 65);
    int shmid = shmget(key, SHM_SIZE, 0666 | IPC_CREAT);
    if (shmid == -1) {
        perror("shmget failed");
        return 1;
    }

    char *shared_memory = (char *)shmat(shmid, NULL, 0);
    if (shared_memory == (char *)(-1)) {
        perror("shmat failed");
        return 1;
    }

    printf("Writing to shared memory...\n");
    strcpy(shared_memory, "Shared Memory!");

    printf("Data written: %s\n", shared_memory);

    shmdt(shared_memory);
    printf("\nLab No.: 3\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No: 23\n");

    return 0;
}
```

Output



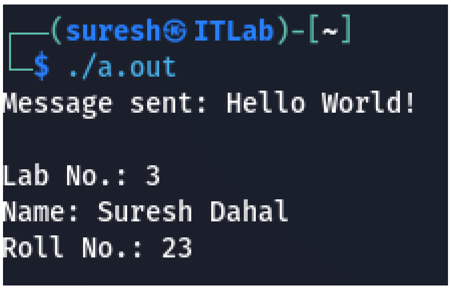
```
┌──(suresh㉿ITLab)-[~]
└─$ ./a.out
Writing to shared memory...
Data written: Shared Memory!

Lab No.: 3
Name: Suresh Dahal
Roll No: 23
```

b. WAP in C to simulate message passing concept for IPC.

Program

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

#define MAX 100

struct message {
    long msg_type;
    char msg_text[MAX];
};

int main() {
    key_t key = ftok("msgqueue", 65);
    int msgid = msgget(key, 0666 | IPC_CREAT);
    if (msgid == -1) {
        perror("msgget failed");
        return 1;
    }

    struct message msg;
    msg.msg_type = 1;
    strcpy(msg.msg_text, "Hello World!");
    if (msgsnd(msgid, &msg, sizeof(msg.msg_text), 0) == -1) {
        perror("msgsnd failed");
        return 1;
    }

    printf("Message sent: %s\n", msg.msg_text);

    printf("\nLab No.: 3\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No.: 23\n");

    return 0;
}
```

Output



```
┌──(suresh㉿ITLab)-[~]
└─$ ./a.out
Message sent: Hello World!

Lab No.: 3
Name: Suresh Dahal
Roll No.: 23
```

TITLE: Simulation of Process Scheduling Algorithms

THEORY

Process scheduling algorithms manage CPU execution to optimize performance. Common types include **FCFS**, **SJF**, **Round Robin**, and **Priority Scheduling** (preemptive/non-preemptive). The goal is to minimize waiting time, turnaround time, and response time while ensuring fair CPU utilization.

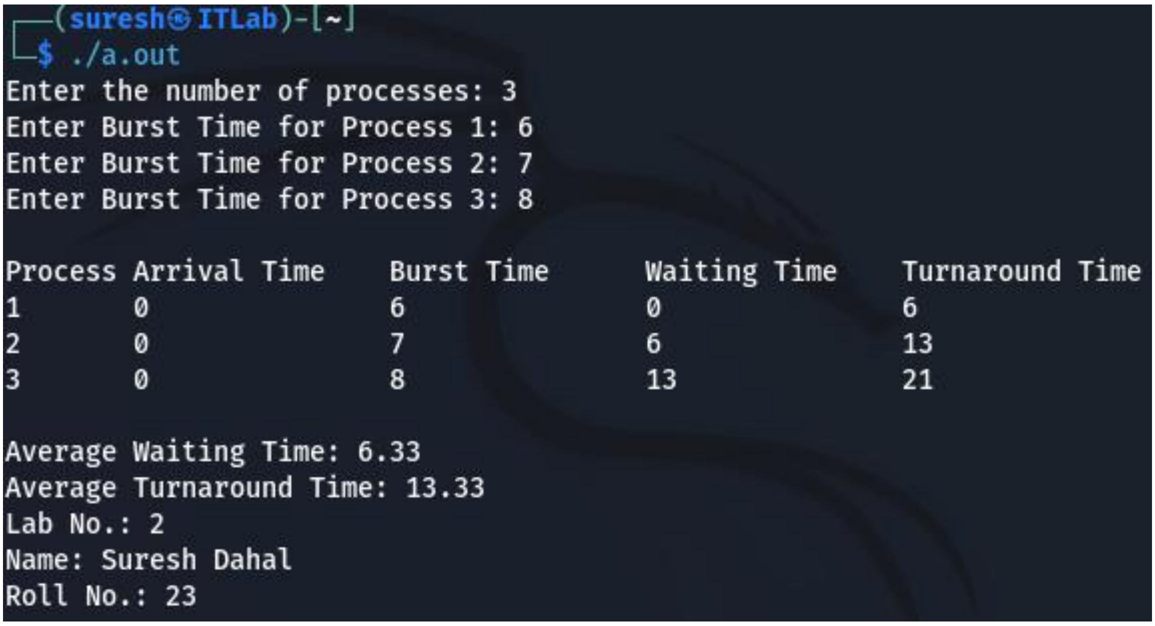
a. WAP in C to simulate FCFS CPU Scheduling Algorithm

Program

```
#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[]) {
    wt[0] = 0; // Waiting time for the first process is 0
    for (int i = 1; i < n; i++) {
        wt[i] = bt[i - 1] + wt[i - 1]; // Waiting time for other processes
    }
}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n; i++) {
        tat[i] = bt[i] + wt[i]; // Turnaround time = Burst time + Waiting time
    }
}

void findAverageTime(int processes[], int n, int bt[]) {
    int wt[n], tat[n];

    findWaitingTime(processes, n, bt, wt);  // Calculate waiting time
    findTurnAroundTime(processes, n, bt, wt, tat);  // Calculate turnaround time

    float total_wt = 0, total_tat = 0;

    printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i], 0, bt[i], wt[i], tat[i]);
    }

    printf("\nAverage Waiting Time: %.2f", total_wt / n);
    printf("\nAverage Turnaround Time: %.2f", total_tat / n);
}
```

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

    int processes[n], burst_time[n];

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

    findAverageTime(processes, n, burst_time);  // Calculate and display average times

    printf("\nLab No.: 2\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No.: 23\n");

    return 0;
}
```

Output



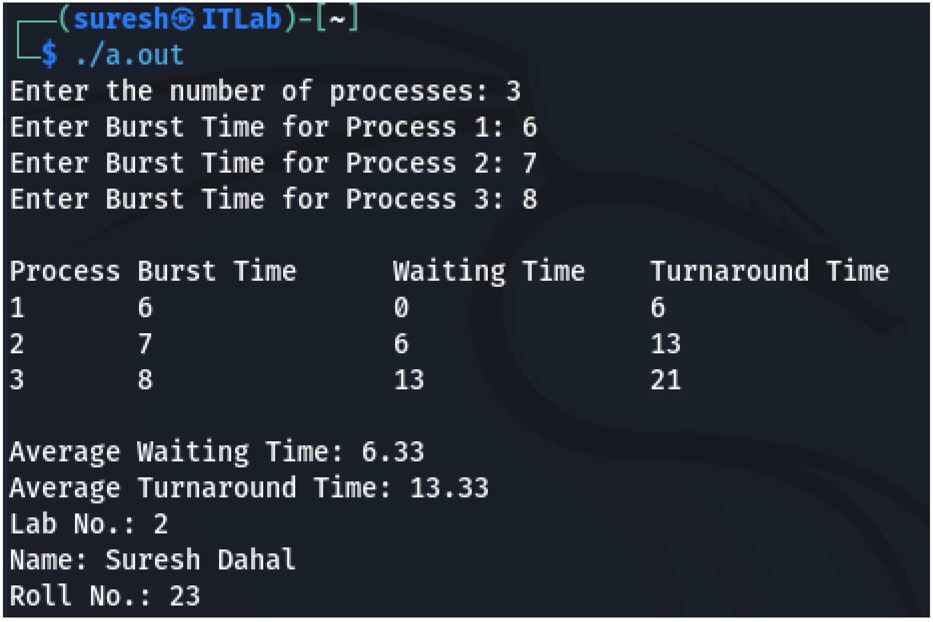
```
┌──(suresh㉿ITLab)-[~]
└─$ ./a.out
Enter the number of processes: 3
Enter Burst Time for Process 1: 6
Enter Burst Time for Process 2: 7
Enter Burst Time for Process 3: 8

Process Arrival Time    Burst Time      Waiting Time    Turnaround Time
1       0               6               0               6
2       0               7               6               13
3       0               8               13              21

Average Waiting Time: 6.33
Average Turnaround Time: 13.33
Lab No.: 2
Name: Suresh Dahal
Roll No.: 23
```

b. **WAP in C to simulate SJF CPU Scheduling Algorithm**

Program

```
#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[]) {
    wt[0] = 0; // Waiting time for the first process is 0
    for (int i = 1; i < n; i++) {
        wt[i] = bt[i - 1] + wt[i - 1]; // Waiting time for other processes
    }
}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n; i++) {
        tat[i] = bt[i] + wt[i]; // Turnaround time = Burst time + Waiting time
    }
}

void findAverageTime(int processes[], int n, int bt[]) {
    int wt[n], tat[n];

    findWaitingTime(processes, n, bt, wt);  // Calculate waiting time
    findTurnAroundTime(processes, n, bt, wt, tat);  // Calculate turnaround time

    float total_wt = 0, total_tat = 0;

    printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf("%d\t%d\t\t%d\t\t%d\n", processes[i], bt[i], wt[i], tat[i]);
    }

    printf("\nAverage Waiting Time: %.2f", total_wt / n);
    printf("\nAverage Turnaround Time: %.2f", total_tat / n);
}

// Function to sort processes according to burst time (SJF)
void sortByBurstTime(int processes[], int n, int bt[]) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (bt[i] > bt[j]) {
                // Swap burst times
                int temp = bt[i];
                bt[i] = bt[j];
                bt[j] = temp;

                // Swap corresponding processes
                temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }
}

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

    int processes[n], burst_time[n];

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

    sortByBurstTime(processes, n, burst_time);  // Sort processes by burst time

    findAverageTime(processes, n, burst_time);  // Calculate and display average times

    printf("\nLab No.: 2\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No.: 23\n");

    return 0;
}
```

Output



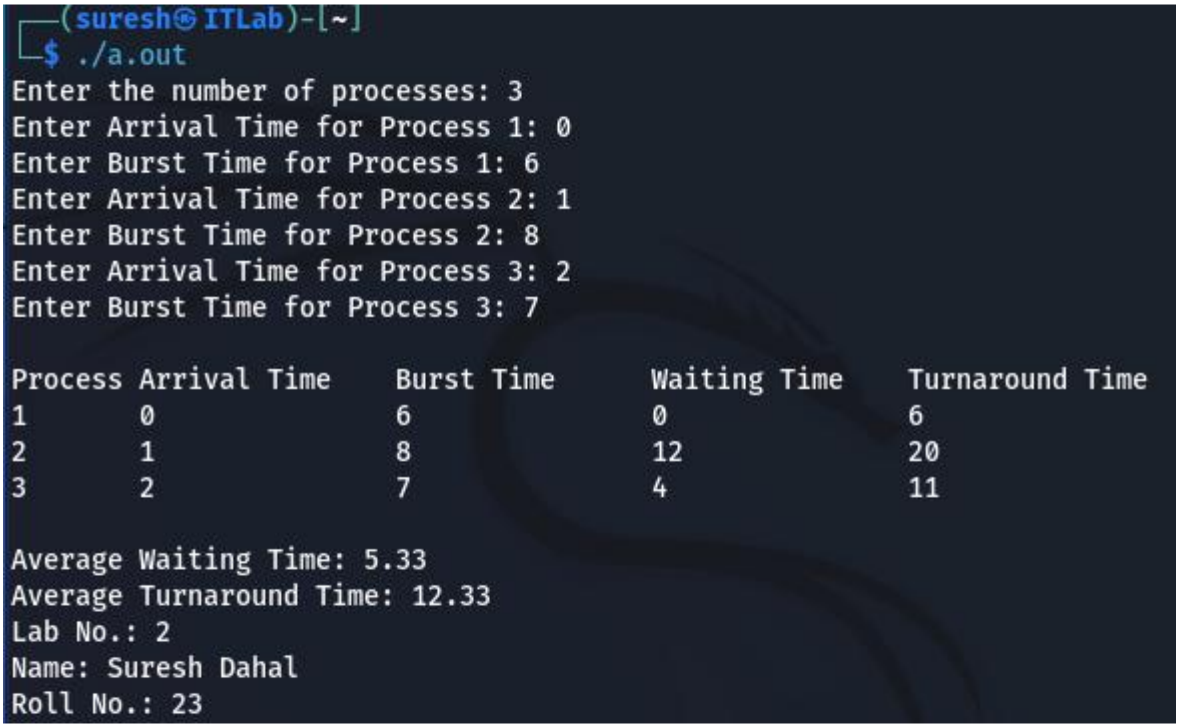
```
┌──(suresh㉿ITLab)-[~]
└─$ ./a.out
Enter the number of processes: 3
Enter Burst Time for Process 1: 6
Enter Burst Time for Process 2: 7
Enter Burst Time for Process 3: 8

Process Burst Time      Waiting Time    Turnaround Time
1       6               0               6
2       7               6               13
3       8               13              21

Average Waiting Time: 6.33
Average Turnaround Time: 13.33
Lab No.: 2
Name: Suresh Dahal
Roll No.: 23
```

c. WAP in C to simulate SRTF CPU Scheduling Algorithm

Program

```
#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int at[], int wt[]) {
    int remaining_bt[n], completed = 0, current_time = 0, min_time, shortest, i;
    for (i = 0; i < n; i++) {
        remaining_bt[i] = bt[i];
    }

    while (completed < n) {
        min_time = 999999; // Arbitrarily large number for comparison
        shortest = -1;

        for (i = 0; i < n; i++) {
            if (at[i] <= current_time && remaining_bt[i] < min_time && remaining_bt[i] > 0)
{
                min_time = remaining_bt[i];
                shortest = i;
            }
        }

        if (shortest != -1) {
            remaining_bt[shortest]--;
            current_time++;

            if (remaining_bt[shortest] == 0) {
                completed++;
                wt[shortest] = current_time - at[shortest] - bt[shortest];
            }
        } else {
            current_time++;
        }
    }
}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n; i++) {
        tat[i] = bt[i] + wt[i];
    }
}

void findAverageTime(int processes[], int n, int bt[], int at[]) {
    int wt[n], tat[n];

    findWaitingTime(processes, n, bt, at, wt);
    findTurnAroundTime(processes, n, bt, wt, tat);

    float total_wt = 0, total_tat = 0;

    printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i], at[i], bt[i], wt[i], tat[i]);
    }

    printf("\nAverage Waiting Time: %.2f", total_wt / n);
    printf("\nAverage Turnaround Time: %.2f", total_tat / n);
}

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

    int processes[n], burst_time[n], arrival_time[n];

    for (int i = 0; i < n; i++) {
        processes[i] = i + 1;
        printf("Enter Arrival Time for Process %d: ", i + 1);
        scanf("%d", &arrival_time[i]);
        printf("Enter Burst Time for Process %d: ", i + 1);
        scanf("%d", &burst_time[i]);
    }

    findAverageTime(processes, n, burst_time, arrival_time);   // Calculate and display average times

    printf("\nLab No.: 2\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No.: 23\n");

    return 0;
}
```

Output



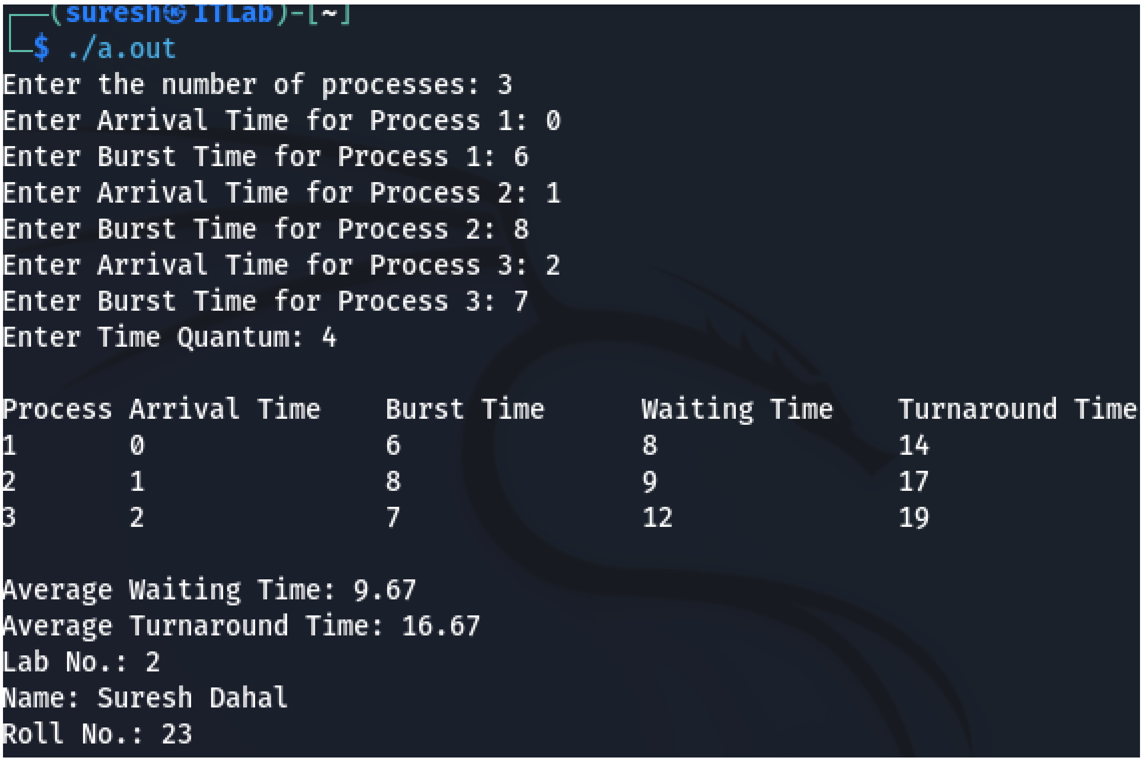
```
┌──(suresh㉿ITLab)-[~]
└─$ ./a.out
Enter the number of processes: 3
Enter Arrival Time for Process 1: 0
Enter Burst Time for Process 1: 6
Enter Arrival Time for Process 2: 1
Enter Burst Time for Process 2: 8
Enter Arrival Time for Process 3: 2
Enter Burst Time for Process 3: 7

Process Arrival Time    Burst Time      Waiting Time    Turnaround Time
1       0               6               0               6
2       1               8               12              20
3       2               7               4               11

Average Waiting Time: 5.33
Average Turnaround Time: 12.33
Lab No.: 2
Name: Suresh Dahal
Roll No.: 23
```

d. WAP in C to simulate Round Robin CPU Scheduling Algorithm

Program

```
#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int at[], int wt[], int quantum) {
    int rem_bt[n], completed = 0, current_time = 0;
    for (int i = 0; i < n; i++) {
        rem_bt[i] = bt[i];
    }
    while (completed < n) {
        for (int i = 0; i < n; i++) {
            if (rem_bt[i] > 0) {
                if (rem_bt[i] > quantum) {
                    rem_bt[i] -= quantum;
                    current_time += quantum;
                } else {
                    current_time += rem_bt[i];
                    wt[i] = current_time - at[i] - bt[i];
                    rem_bt[i] = 0;
                    completed++;
                }
            }
        }
    }
}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n; i++) {
        tat[i] = bt[i] + wt[i];
    }
}

void findAverageTime(int processes[], int n, int bt[], int at[], int quantum) {
    int wt[n], tat[n];

    findWaitingTime(processes, n, bt, at, wt, quantum);  // Calculate waiting time
    findTurnAroundTime(processes, n, bt, wt, tat);  // Calculate turnaround time

    float total_wt = 0, total_tat = 0;

    printf("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", processes[i], at[i], bt[i], wt[i], tat[i]);
    }

    printf("\nAverage Waiting Time: %.2f", total_wt / n);
    printf("\nAverage Turnaround Time: %.2f", total_tat / n);
}

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

    for (int i = 0; i < n; i++) {
        processes[i] = i + 1;
        printf("Enter Arrival Time for Process %d: ", i + 1);
        scanf("%d", &arrival_time[i]);
        printf("Enter Burst Time for Process %d: ", i + 1);
        scanf("%d", &burst_time[i]);
    }

    printf("Enter Time Quantum: ");
    scanf("%d", &quantum);

    findAverageTime(processes, n, burst_time, arrival_time, quantum);  // Calculate and display average times

    printf("\nLab No.: 2\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No.: 23\n");

    return 0;
}
```

Output



```
┌──(suresh㉿ITLab)-[~]
└─$ ./a.out
Enter the number of processes: 3
Enter Arrival Time for Process 1: 0
Enter Burst Time for Process 1: 6
Enter Arrival Time for Process 2: 1
Enter Burst Time for Process 2: 8
Enter Arrival Time for Process 3: 2
Enter Burst Time for Process 3: 7
Enter Time Quantum: 4

Process Arrival Time    Burst Time      Waiting Time    Turnaround Time
1       0               6               8               14
2       1               8               9               17
3       2               7               12              19

Average Waiting Time: 9.67
Average Turnaround Time: 16.67
Lab No.: 2
Name: Suresh Dahal
Roll No.: 23
```

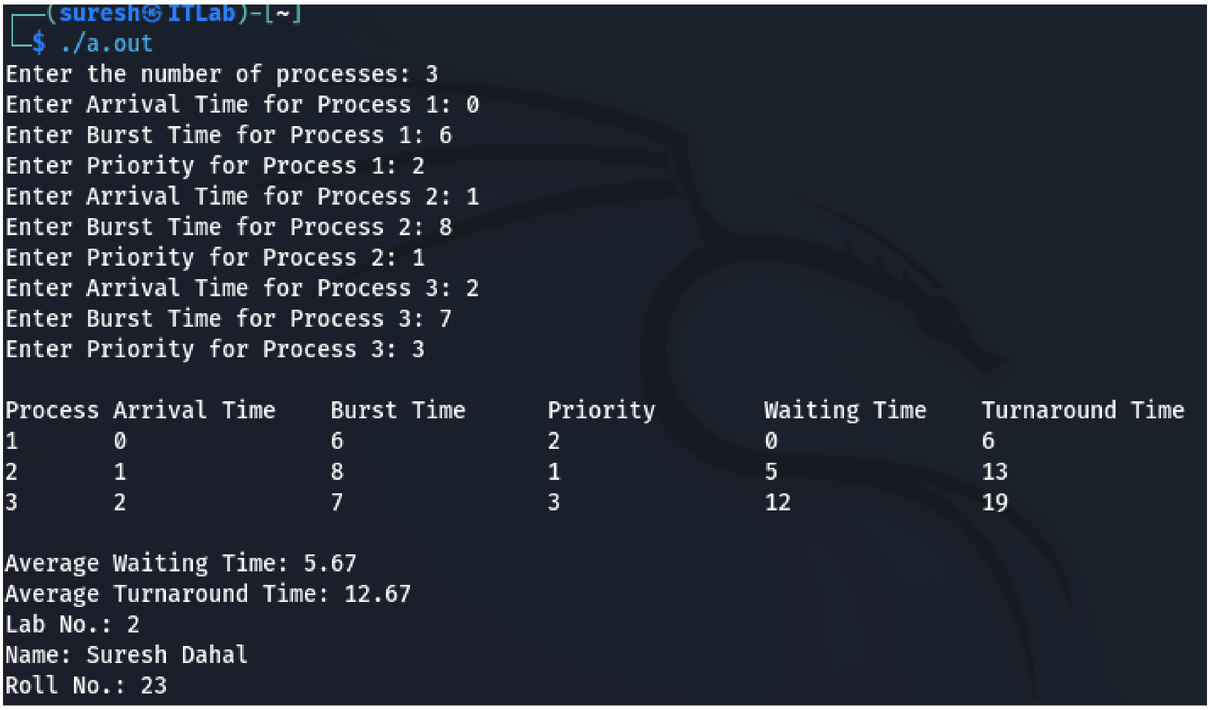
e. **WAP in C to simulate Non-Preemptive Priority Scheduling Algorithm**

Program

```
#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int at[], int wt[], int priority[]) {
    int completed = 0, current_time = 0;
    int remaining = n;
    int finish_time[n], remaining_bt[n];
    int is_completed[n]; // To track whether a process is completed

    for (int i = 0; i < n; i++) {
        remaining_bt[i] = bt[i];
        is_completed[i] = 0;
    }

    while (remaining > 0) {
        int min_priority = 99999;
        int idx = -1;

        for (int i = 0; i < n; i++) {
            // Find the process with the minimum priority that is ready and not yet completed
            if (at[i] <= current_time && is_completed[i] == 0 && priority[i] < min_priority) {
                min_priority = priority[i];
                idx = i;
            }
        }

        if (idx != -1) {
            is_completed[idx] = 1; // Mark process as completed
            finish_time[idx] = current_time + bt[idx];
            current_time += bt[idx];
            remaining--;
        } else {
            current_time++; // Increment time if no process is ready
        }
    }

    // Calculate waiting time
    for (int i = 0; i < n; i++) {
        wt[i] = finish_time[i] - at[i] - bt[i];
    }
}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n; i++) {
        tat[i] = bt[i] + wt[i];
    }
}

void findAverageTime(int processes[], int n, int bt[], int at[], int priority[]) {
    int wt[n], tat[n];

    findWaitingTime(processes, n, bt, at, wt, priority); // Calculate waiting time
    findTurnAroundTime(processes, n, bt, wt, tat); // Calculate turnaround time

    float total_wt = 0, total_tat = 0;

    printf("\nProcess\tArrival Time\tBurst Time\tPriority\tWaiting Time\tTurnaround
Time\n");
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i], at[i], bt[i], priority[i], wt[i],
tat[i]);
    }

    printf("\nAverage Waiting Time: %.2f", total_wt / n);
    printf("\nAverage Turnaround Time: %.2f", total_tat / n);
}

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

    int processes[n], burst_time[n], arrival_time[n], priority[n];

    for (int i = 0; i < n; i++) {
        processes[i] = i + 1;
        printf("Enter Arrival Time for Process %d: ", i + 1);
        scanf("%d", &arrival_time[i]);
        printf("Enter Burst Time for Process %d: ", i + 1);
        scanf("%d", &burst_time[i]);
        printf("Enter Priority for Process %d: ", i + 1);
        scanf("%d", &priority[i]);
    }

    findAverageTime(processes, n, burst_time, arrival_time, priority); // Calculate and
display average times
```

```
    printf("\nLab No.: 2\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No.: 23\n");

    return 0;
}
```

Output



```
┌──(suresh㉿ITLab)-[~]
└─$ ./a.out
Enter the number of processes: 3
Enter Arrival Time for Process 1: 0
Enter Burst Time for Process 1: 6
Enter Priority for Process 1: 2
Enter Arrival Time for Process 2: 1
Enter Burst Time for Process 2: 8
Enter Priority for Process 2: 1
Enter Arrival Time for Process 3: 2
Enter Burst Time for Process 3: 7
Enter Priority for Process 3: 3

Process Arrival Time    Burst Time      Priority        Waiting Time    Turnaround Time
1       0               6               2               0               6
2       1               8               1               5               13
3       2               7               3               12              19

Average Waiting Time: 5.67
Average Turnaround Time: 12.67
Lab No.: 2
Name: Suresh Dahal
Roll No.: 23
```

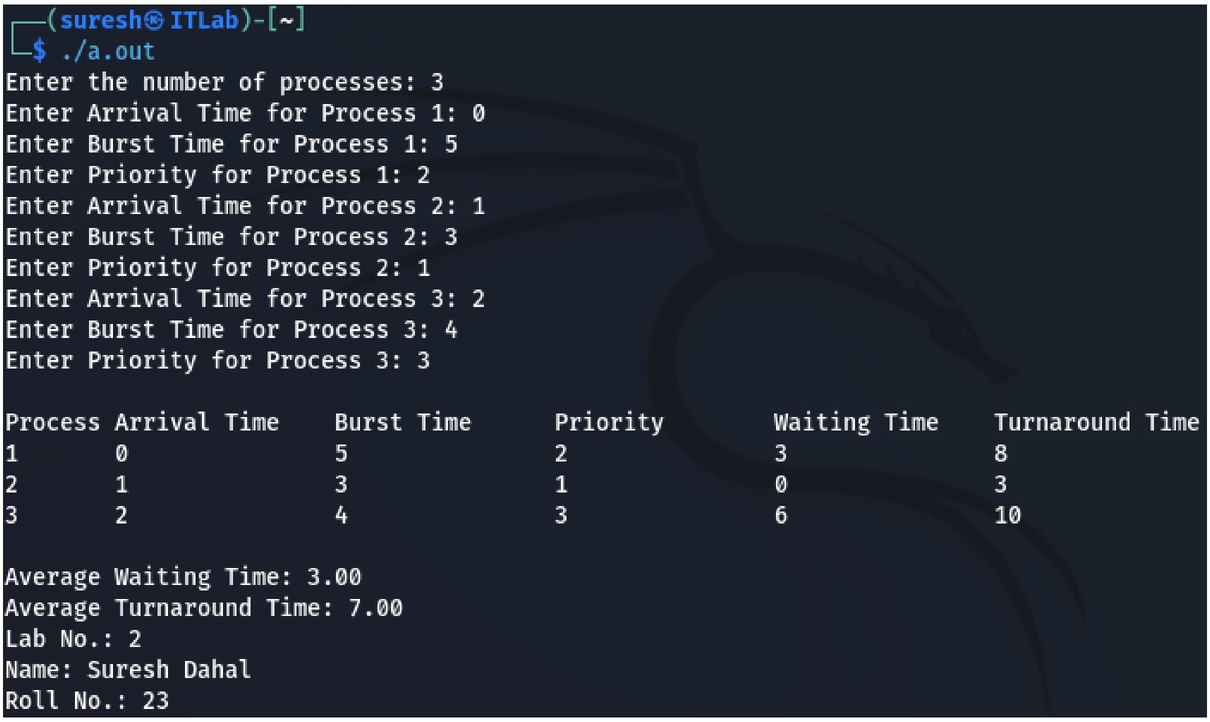
f. WAP in C to simulate Preemptive Priority Scheduling Algorithm

Program

```
#include <stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int at[], int wt[], int priority[]) {
    int remaining = n;
    int remaining_bt[n];
    int finish_time[n];
    int is_completed[n];
    int current_time = 0;
    int min_priority, idx;

    for (int i = 0; i < n; i++) {
        remaining_bt[i] = bt[i];
        is_completed[i] = 0;
    }
    while (remaining > 0) {
        min_priority = 9999;
        idx = -1;

        for (int i = 0; i < n; i++) {
            if (at[i] <= current_time && is_completed[i] == 0 && priority[i] < min_priority) {
                min_priority = priority[i];
                idx = i;
            }
        }

        if (idx != -1) {
            remaining_bt[idx]--;
            if (remaining_bt[idx] == 0) {
                is_completed[idx] = 1;
                remaining--;
                finish_time[idx] = current_time + 1;
            }
            current_time++;
        } else {
            current_time++; // Increment time if no process is ready
        }
    }

    // Calculate waiting time
    for (int i = 0; i < n; i++) {
        wt[i] = finish_time[i] - at[i] - bt[i];
    }
}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n; i++) {
        tat[i] = bt[i] + wt[i];
    }
}

void findAverageTime(int processes[], int n, int bt[], int at[], int priority[]) {
    int wt[n], tat[n];

    findWaitingTime(processes, n, bt, at, wt, priority); // Calculate waiting time
    findTurnAroundTime(processes, n, bt, wt, tat); // Calculate turnaround time

    float total_wt = 0, total_tat = 0;

    printf("\nProcess\tArrival Time\tBurst Time\tPriority\tWaiting Time\tTurnaround
Time\n");
```

```
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i], at[i], bt[i], priority[i], wt[i],
tat[i]);
    }

    printf("\nAverage Waiting Time: %.2f", total_wt / n);
    printf("\nAverage Turnaround Time: %.2f", total_tat / n);
}

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

    int processes[n], burst_time[n], arrival_time[n], priority[n];

    for (int i = 0; i < n; i++) {
        processes[i] = i + 1;
        printf("Enter Arrival Time for Process %d: ", i + 1);
        scanf("%d", &arrival_time[i]);
        printf("Enter Burst Time for Process %d: ", i + 1);
        scanf("%d", &burst_time[i]);
        printf("Enter Priority for Process %d: ", i + 1);
        scanf("%d", &priority[i]);
    }

    findAverageTime(processes, n, burst_time, arrival_time, priority); // Calculate and
display average times

    printf("\nLab No.: 2\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No.: 23\n");

    return 0;
}
```

Output



```
┌──(suresh㉿ITLab)-[~]
└─$ ./a.out
Enter the number of processes: 3
Enter Arrival Time for Process 1: 0
Enter Burst Time for Process 1: 5
Enter Priority for Process 1: 2
Enter Arrival Time for Process 2: 1
Enter Burst Time for Process 2: 3
Enter Priority for Process 2: 1
Enter Arrival Time for Process 3: 2
Enter Burst Time for Process 3: 4
Enter Priority for Process 3: 3

Process Arrival Time    Burst Time      Priority        Waiting Time    Turnaround Time
1       0               5               2               3               8
2       1               3               1               0               3
3       2               4               3               6               10

Average Waiting Time: 3.00
Average Turnaround Time: 7.00
Lab No.: 2
Name: Suresh Dahal
Roll No.: 23
```

TITLE: Simulation of Deadlock Avoidance and Deadlock Detection Algorithms

THEORY

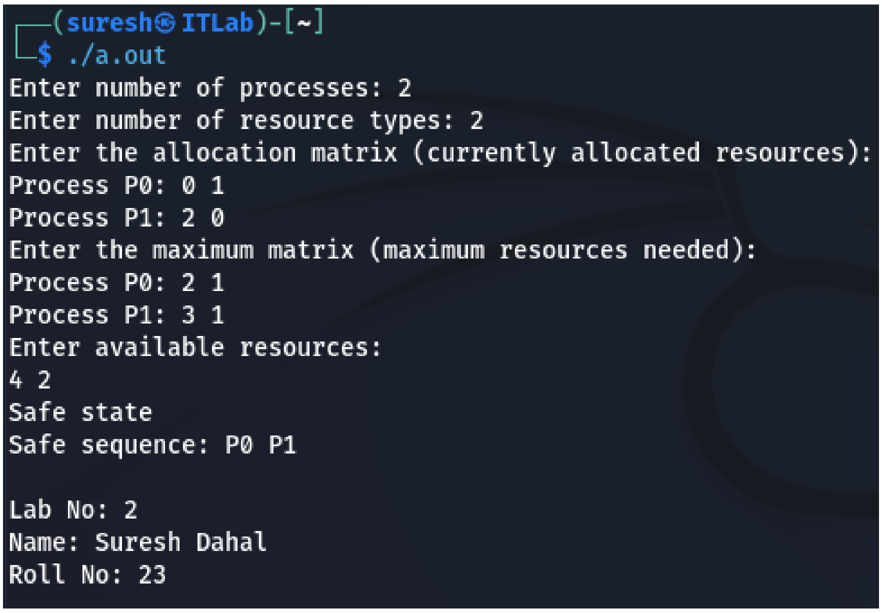
Deadlock avoidance ensures that resources are allocated in a way that avoids deadlock, often using techniques like the Banker's Algorithm. Deadlock detection periodically checks if a deadlock has occurred and takes corrective actions, such as terminating processes or rolling back operations.

a. **WAP to implement Bankers Algorithm for multiple type of resources to decide safe/unsafe state.**

Program

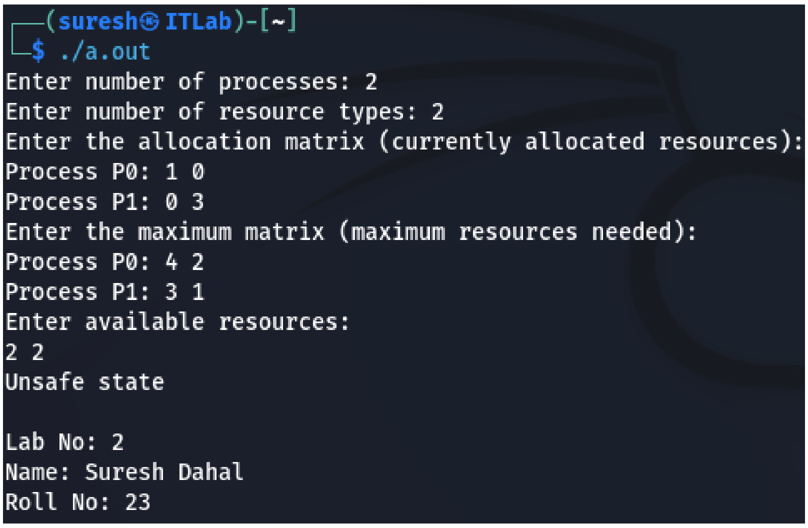
```
#include <stdio.h>
#include <stdbool.h>
#define MAX 10
#define RESOURCE_TYPES 3

Void calculateNeed(int need[MAX][RESOURCE_TYPES], int
max[MAX][RESOURCE_TYPES], int allocation[MAX][RESOURCE_TYPES], int
n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < RESOURCE_TYPES; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}

bool isLessThanOrEqual(int need[], int work[], int m) {
    for (int i = 0; i < m; i++) {
        if (need[i] > work[i]) {
            return false;
        }
    }
    return true;
}
void bankersAlgorithm(int allocation[MAX][RESOURCE_TYPES], int
max[MAX][RESOURCE_TYPES], int available[], int n) {
    int need[MAX][RESOURCE_TYPES];
    int work[RESOURCE_TYPES];
    bool finish[MAX];
    int safeSeq[MAX];
    int count = 0;

    // Calculate the Need matrix
    calculateNeed(need, max, allocation, n);
    for (int i = 0; i < n; i++) {
        finish[i] = false;
    }
    for (int i = 0; i < RESOURCE_TYPES; i++) {
        work[i] = available[i];
    }

    // Start checking for safe sequence
    while (count < n) {
        bool progressMade = false;
        for (int i = 0; i < n; i++) {
            // Find a process that has not finished and can proceed
            if (!finish[i] && isLessThanOrEqual(need[i], work, RESOURCE_TYPES)) {
                // If it can proceed, pretend it finishes and release resources
                for (int j = 0; j < RESOURCE_TYPES; j++) {
                    work[j] += allocation[i][j];
                }
                safeSeq[count++] = i;
                finish[i] = true;
                progressMade = true;
                break;
            }
        }
        if (!progressMade) {
            // No process could proceed, unsafe state
            printf("Unsafe state\n");
            return;
        }
    }

    // If all processes finished
    printf("Safe state\nSafe sequence: ");
    for (int i = 0; i < n; i++) {
        printf("P%d ", safeSeq[i]);
    }
    printf("\n");
}

int main() {
    int n, m;

    // Take number of processes and resources
    printf("Enter number of processes: ");
    scanf("%d", &n);
    printf("Enter number of resource types: ");
    scanf("%d", &m);
    int allocation[MAX][RESOURCE_TYPES], max[MAX][RESOURCE_TYPES],
available[RESOURCE_TYPES];

    // Input allocation matrix
    printf("Enter the allocation matrix (currently allocated resources):\n");
    for (int i = 0; i < n; i++) {
        printf("Process P%d: ", i);
        for (int j = 0; j < m; j++) {
            scanf("%d", &allocation[i][j]);
        }
    }

    // Input max matrix
    printf("Enter the maximum matrix (maximum resources needed):\n");
    for (int i = 0; i < n; i++) {
        printf("Process P%d: ", i);
        for (int j = 0; j < m; j++) {
            scanf("%d", &max[i][j]);
        }
    }

    // Input available resources
    printf("Enter available resources:\n");
    for (int i = 0; i < m; i++) {
        scanf("%d", &available[i]);
    }

    // Run Banker's Algorithm
    bankersAlgorithm(allocation, max, available, n);

    // Print lab info at the end
    printf("\nLab No: 2\nName: Suresh Dahal\nRoll No: 23\n");

    return 0;
}
```

Output



```
┌──(suresh㉿ITLab)-[~]
└─$ ./a.out
Enter number of processes: 2
Enter number of resource types: 2
Enter the allocation matrix (currently allocated resources):
Process P0: 0 1
Process P1: 2 0
Enter the maximum matrix (maximum resources needed):
Process P0: 2 1
Process P1: 3 1
Enter available resources:
4 2
Safe state
Safe sequence: P0 P1

Lab No: 2
Name: Suresh Dahal
Roll No: 23
```

Figure 1 Safe state



```
┌──(suresh㉿ITLab)-[~]
└─$ ./a.out
Enter number of processes: 2
Enter number of resource types: 2
Enter the allocation matrix (currently allocated resources):
Process P0: 1 0
Process P1: 0 3
Enter the maximum matrix (maximum resources needed):
Process P0: 4 2
Process P1: 3 1
Enter available resources:
2 2
Unsafe state

Lab No: 2
Name: Suresh Dahal
Roll No: 23
```

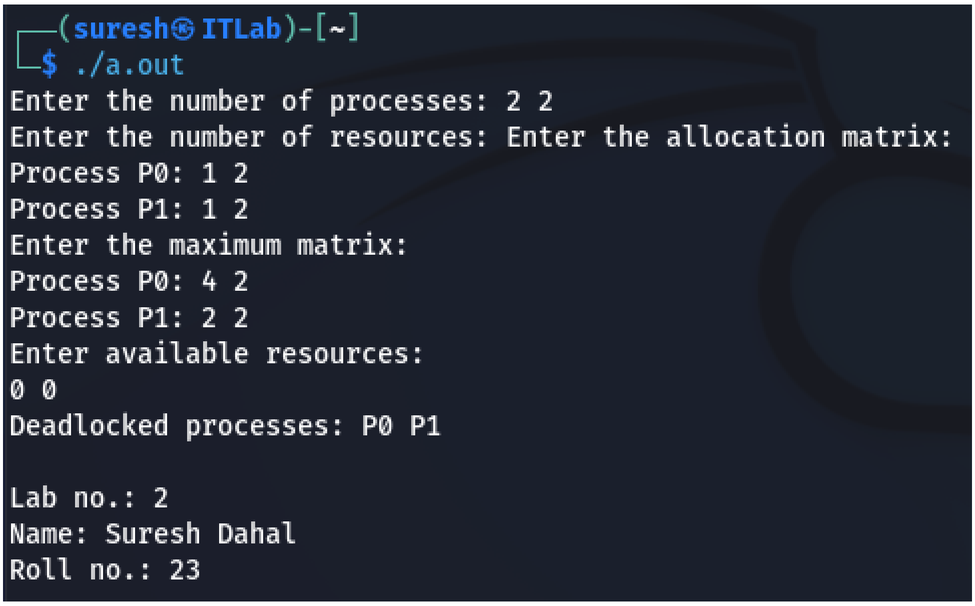
Figure 2 unsafe state

b. **WAP for deadlock detection in the system having multiple type of resources. The program should list the deadlocked process in case of deadlock detection results true.**

Program

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

#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

int processes, resources;
int allocation[MAX_PROCESSES][MAX_RESOURCES];
int maximum[MAX_PROCESSES][MAX_RESOURCES];
int available[MAX_RESOURCES];
int need[MAX_PROCESSES][MAX_RESOURCES];

void findDeadlocks() {
    int work[MAX_RESOURCES];
    bool finish[MAX_PROCESSES] = {0};
    int deadlock[MAX_PROCESSES];
    int deadlockCount = 0;

    // Initialize work with available resources
    for (int i = 0; i < resources; i++) {
        work[i] = available[i];
    }

    while (1) {
        bool progressMade = false;

        // Try to find a process that can complete
        for (int p = 0; p < processes; p++) {
            if (!finish[p]) {
                bool canProceed = true;
                // Check if the process can proceed with available resources
                for (int r = 0; r < resources; r++) {
                    if (need[p][r] > work[r]) {
                        canProceed = false;
                        break;
                    }
                }
                if (canProceed) {
                    // If the process can proceed, add its allocated resources to work
                    for (int r = 0; r < resources; r++) {
                        work[r] += allocation[p][r];
                    }
                    finish[p] = true;
                    progressMade = true;
                    break;
                }
            }
        }

        if (!progressMade) {
            // If no progress can be made, we have found deadlocked processes
            for (int i = 0; i < processes; i++) {
                if (!finish[i]) {
                    deadlock[deadlockCount++] = i;
                }
            }
            break;
        }
    }

    if (deadlockCount > 0) {
        printf("Deadlocked processes: ");
        for (int i = 0; i < deadlockCount; i++) {
            printf("P%d ", deadlock[i]);
        }
        printf("\n");
    } else {
        printf("No deadlock detected.\n");
    }
}

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

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

    // Input the allocation matrix
    printf("Enter the allocation matrix:\n");
    for (int i = 0; i < processes; i++) {
        printf("Process P%d: ", i);
        for (int j = 0; j < resources; j++) {
            scanf("%d", &allocation[i][j]);
        }
    }

    // Input the maximum matrix
    printf("Enter the maximum matrix:\n");
    for (int i = 0; i < processes; i++) {
        printf("Process P%d: ", i);
        for (int j = 0; j < resources; j++) {
            scanf("%d", &maximum[i][j]);
        }
    }

    // Calculate the need matrix
    for (int i = 0; i < processes; i++) {
        for (int j = 0; j < resources; j++) {
            need[i][j] = maximum[i][j] - allocation[i][j];
        }
    }

    // Input the available resources
    printf("Enter available resources:\n");
    for (int i = 0; i < resources; i++) {
        scanf("%d", &available[i]);
    }

    // Call the deadlock detection function
    findDeadlocks();

    printf(“Lab no.: 2\nName: Suresh Dahal\nRoll no.: 23”);


    return 0;
}
```

Output



```
┌──(suresh㉿ITLab)-[~]
└─$ ./a.out
Enter the number of processes: 2 2
Enter the number of resources: Enter the allocation matrix:
Process P0: 1 2
Process P1: 1 2
Enter the maximum matrix:
Process P0: 4 2
Process P1: 2 2
Enter available resources:
0 0
Deadlocked processes: P0 P1

Lab no.: 2
Name: Suresh Dahal
Roll no.: 23
```

TITLE: Simulation of Page Replacement Algorithms

THEORY

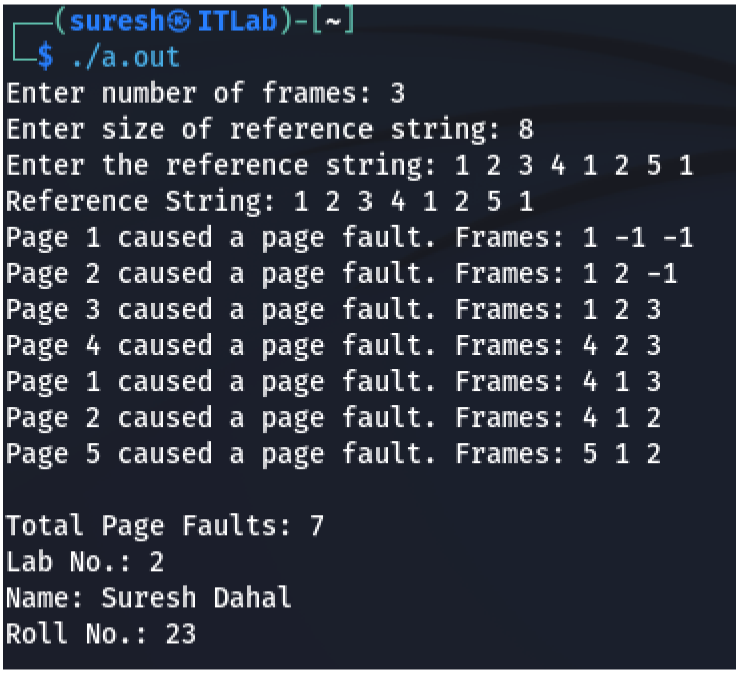
Page replacement algorithms decide which pages to keep in memory and which to swap out during a page fault. Common algorithms include FIFO, Optimal, LRU, Second Chance, and LFU. Each algorithm aims to minimize page faults and optimize memory usage.

a. **WAP in C to simulate FIFO Page Replacement Algorithm**

Program

```
#include <stdio.h>

void FIFO(int frameCount, int referenceString[], int size) {
    int frames[frameCount];
    int pageFaults = 0;
    int index = 0;
    int isPageInMemory;

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

    printf("Reference String: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", referenceString[i]);
    }
    printf("\n");

    for (int i = 0; i < size; i++) {
        isPageInMemory = 0;

        // Check if the page is already in memory
        for (int j = 0; j < frameCount; j++) {
            if (frames[j] == referenceString[i]) {
                isPageInMemory = 1;
                break;
            }
        }

        // If page is not in memory, replace the oldest page (FIFO)
        if (!isPageInMemory) {
            frames[index] = referenceString[i];
            index = (index + 1) % frameCount;
            pageFaults++;
            printf("Page %d caused a page fault. Frames: ", referenceString[i]);
            for (int j = 0; j < frameCount; j++) {
                printf("%d ", frames[j]);
            }
            printf("\n");
        }
    }

    printf("\nTotal Page Faults: %d\n", pageFaults);
    printf("Lab No.: 2\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No.: 23\n");
}

int main() {
    int frameCount, size;

    printf("Enter number of frames: ");
    scanf("%d", &frameCount);

    printf("Enter size of reference string: ");
    scanf("%d", &size);

    int referenceString[size];

    printf("Enter the reference string: ");
    for (int i = 0; i < size; i++) {
        scanf("%d", &referenceString[i]);
    }

    FIFO(frameCount, referenceString, size);

    return 0;
}
```

Output



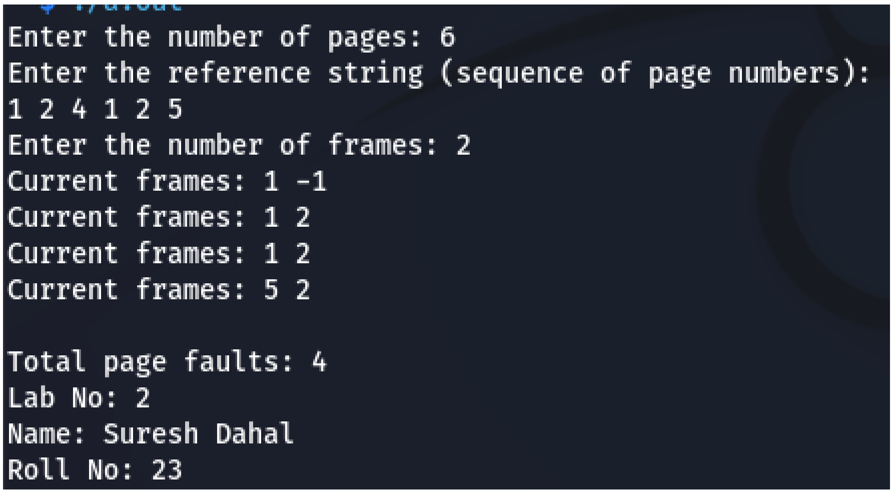
```
┌──(suresh㉿ITLab)-[~]
└─$ ./a.out
Enter number of frames: 3
Enter size of reference string: 8
Enter the reference string: 1 2 3 4 1 2 5 1
Reference String: 1 2 3 4 1 2 5 1
Page 1 caused a page fault. Frames: 1 -1 -1
Page 2 caused a page fault. Frames: 1 2 -1
Page 3 caused a page fault. Frames: 1 2 3
Page 4 caused a page fault. Frames: 4 2 3
Page 1 caused a page fault. Frames: 4 1 3
Page 2 caused a page fault. Frames: 4 1 2
Page 5 caused a page fault. Frames: 5 1 2

Total Page Faults: 7
Lab No.: 2
Name: Suresh Dahal
Roll No.: 23
```

b. WAP in C to simulate Optimal Page Replacement Algorithm

Program

```
#include <stdio.h>

#define MAX_PAGES 10
#define MAX_FRAMES 10

int frames[MAX_FRAMES];
int referenceString[MAX_PAGES];

int findOptimalPageToReplace(int currentIndex, int frames[], int referenceString[], int
n) {
    int farthest = currentIndex;
    int pageToReplace = -1;

    for (int i = 0; i < n; i++) {
        int page = frames[i];
        int j;
        for (j = currentIndex; j < n; j++) {
            if (referenceString[j] == page) {
                break;
            }
        }

        if (j == n) {
            return i;  // If the page is not found, replace it
        }

        if (j > farthest) {
            farthest = j;
            pageToReplace = i;
        }
    }

    return pageToReplace;
}

int main() {
    int numPages, numFrames;

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

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

    printf("Enter the number of frames: ");
    scanf("%d", &numFrames);

    // Initialize frames with -1 (empty slots)
    for (int i = 0; i < numFrames; i++) {
        frames[i] = -1;
    }

    int pageFaults = 0;

    // Simulate the optimal page replacement algorithm
    for (int i = 0; i < numPages; i++) {
        int page = referenceString[i];
        int pageFound = 0;

        // Check if page is already in one of the frames
        for (int j = 0; j < numFrames; j++) {
            if (frames[j] == page) {
                pageFound = 1;
                break;
            }
        }

        // If page is not found, it's a page fault
        if (!pageFound) {
            pageFaults++;
            int pageToReplace = findOptimalPageToReplace(i + 1, frames,
referenceString, numPages);

            // Replace the page
            frames[pageToReplace] = page;

            // Print the current frames
            printf("Current frames: ");
            for (int j = 0; j < numFrames; j++) {
                printf("%d ", frames[j]);
            }
            printf("\n");
        }
    }

    printf("\nTotal page faults: %d\n", pageFaults);

    printf("Lab No: 2\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No: 23\n");

    return 0;
}
```

Output



```
$ ./a.out
Enter the number of pages: 6
Enter the reference string (sequence of page numbers):
1 2 4 1 2 5
Enter the number of frames: 2
Current frames: 1 -1
Current frames: 1 2
Current frames: 1 2
Current frames: 5 2

Total page faults: 4
Lab No: 2
Name: Suresh Dahal
Roll No: 23
```

c. WAP in C to simulate LRU Page Replacement Algorithm

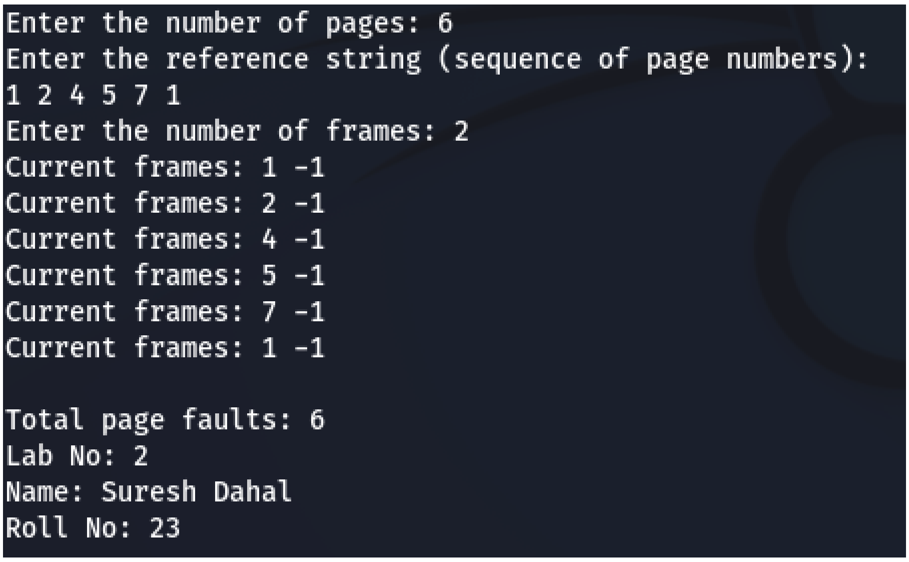
Program

```
#include <stdio.h>

#define MAX_PAGES 10
#define MAX_FRAMES 10

int frames[MAX_FRAMES];
int referenceString[MAX_PAGES];

int isPageInFrames(int page, int frames[], int numFrames) {
    for (int i = 0; i < numFrames; i++) {
        if (frames[i] == page) {
            return 1; // Page is in frames
        }
    }
    return 0; // Page is not in frames
}

int getLRUPage(int numFrames, int frames[], int recent[], int numPages, int
currentIndex) {
    int lruIndex = 0;
    for (int i = 1; i < numFrames; i++) {
        if (recent[frames[i]] < recent[frames[lruIndex]]) {
            lruIndex = i;
        }
    }
    return lruIndex;
}

int main() {
    int numPages, numFrames;

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

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

```
printf("Enter the number of frames: ");
scanf("%d", &numFrames);

// Initialize frames with -1 (empty slots)
for (int i = 0; i < numFrames; i++) {
    frames[i] = -1;
}

// Array to keep track of last usage time of pages
int recent[MAX_PAGES] = {0};
int pageFaults = 0;

// Simulate the LRU page replacement algorithm
for (int i = 0; i < numPages; i++) {
    int page = referenceString[i];
    int pageFound = 0;

    // Check if page is already in one of the frames
    if (isPageInFrames(page, frames, numFrames)) {
        pageFound = 1;
    }

    // If page is not found, it's a page fault
    if (!pageFound) {
        pageFaults++;
        int replaceIndex = getLRUPage(numFrames, frames, recent, numPages, i);

        // Replace the LRU page
        frames[replaceIndex] = page;
    }

    // Update the recent usage time of the page
    recent[page] = i;

    // Print the current frames after each page fault
    printf("Current frames: ");
    for (int j = 0; j < numFrames; j++) {
        printf("%d ", frames[j]);
    }
    printf("\n");
}
printf("\nTotal page faults: %d\n", pageFaults);

printf("Lab No: 2\n");
printf("Name: Suresh Dahal\n");
printf("Roll No: 23\n");
```

```
    return 0;
}
```

Output



```
Enter the number of pages: 6
Enter the reference string (sequence of page numbers):
1 2 4 5 7 1
Enter the number of frames: 2
Current frames: 1 -1
Current frames: 2 -1
Current frames: 4 -1
Current frames: 5 -1
Current frames: 7 -1
Current frames: 1 -1

Total page faults: 6
Lab No: 2
Name: Suresh Dahal
Roll No: 23
```

d. **WAP in C to simulate Second Chance Page Replacement Algorithm**

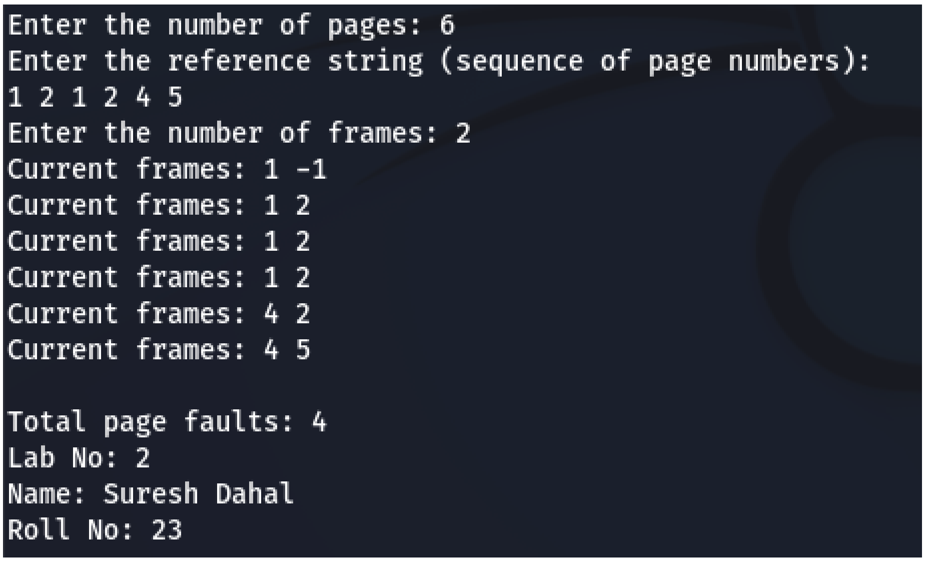
Program

```
#include <stdio.h>

#define MAX_PAGES 10
#define MAX_FRAMES 10

int frames[MAX_FRAMES];
int referenceString[MAX_PAGES];

int main() {
    int numPages, numFrames;
    int referenceBits[MAX_FRAMES] = {0}; // To track reference bits (0 or 1)

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

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

    printf("Enter the number of frames: ");
    scanf("%d", &numFrames);
```

```
// Initialize frames with -1 (empty slots)
for (int i = 0; i < numFrames; i++) {
    frames[i] = -1;
}

int pageFaults = 0;
int pointer = 0; // To keep track of the next frame to replace

// Simulate the Second Chance page replacement algorithm
for (int i = 0; i < numPages; i++) {
    int page = referenceString[i];
    int pageFound = 0;

    // Check if page is already in one of the frames
    for (int j = 0; j < numFrames; j++) {
        if (frames[j] == page) {
            pageFound = 1; // Page hit
            referenceBits[j] = 1; // Set reference bit
            break;
        }
    }

    // If page is not found, it's a page fault
    if (!pageFound) {
        pageFaults++;

        // Find an empty slot or a page to replace using the second chance mechanism
        while (referenceBits[pointer] == 1) {
            referenceBits[pointer] = 0; // Reset the reference bit to 0
            pointer = (pointer + 1) % numFrames; // Move the pointer to the next frame
        }

        // Replace the page
        frames[pointer] = page;
        referenceBits[pointer] = 1; // Set reference bit to 1 for the newly loaded page
        pointer = (pointer + 1) % numFrames; // Move the pointer to the next frame
    }

    // Print the current frames after each page fault
    printf("Current frames: ");
    for (int j = 0; j < numFrames; j++) {
        printf("%d ", frames[j]);
    }
    printf("\n");
}
```

```
    printf("\nTotal page faults: %d\n", pageFaults);

    printf("Lab No: 2\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No: 23\n");

    return 0;
}
```

Output



```
Enter the number of pages: 6
Enter the reference string (sequence of page numbers):
1 2 1 2 4 5
Enter the number of frames: 2
Current frames: 1 -1
Current frames: 1 2
Current frames: 1 2
Current frames: 1 2
Current frames: 4 2
Current frames: 4 5

Total page faults: 4
Lab No: 2
Name: Suresh Dahal
Roll No: 23
```

e. **WAP in C to simulate LFU Page Replacement Algorithm**

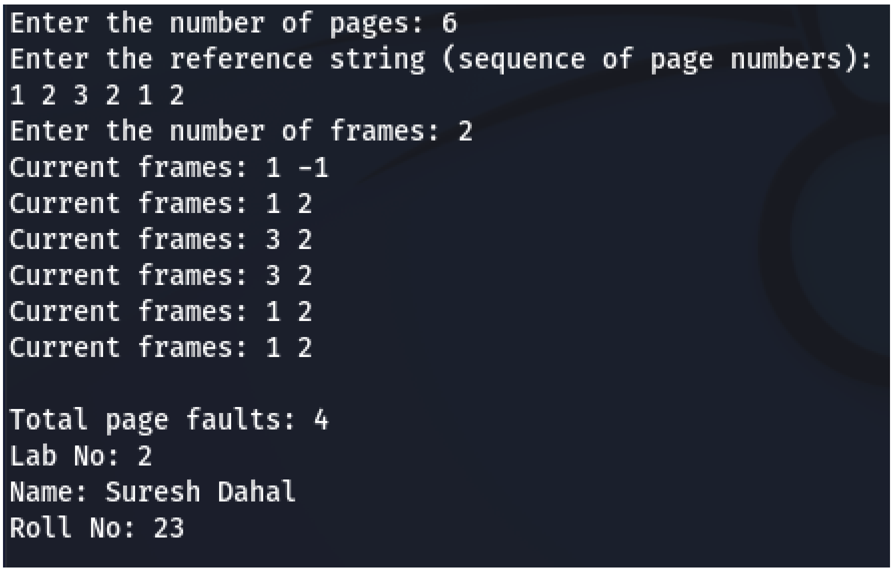
Program

```
#include <stdio.h>

#define MAX_PAGES 10
#define MAX_FRAMES 10

int frames[MAX_FRAMES];
int referenceString[MAX_PAGES];

int main() {
    int numPages, numFrames;
    int frequency[MAX_FRAMES];  // To track the frequency of page accesses
    int pageFaults = 0;

    printf("Enter the number of pages: ");
```

```
scanf("%d", &numPages);

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

printf("Enter the number of frames: ");
scanf("%d", &numFrames);

// Initialize frames with -1 (empty slots) and frequency to 0
for (int i = 0; i < numFrames; i++) {
    frames[i] = -1;
    frequency[i] = 0;
}

// Simulate LFU page replacement algorithm
for (int i = 0; i < numPages; i++) {
    int page = referenceString[i];
    int pageFound = 0;

    // Check if the page is already in one of the frames
    for (int j = 0; j < numFrames; j++) {
        if (frames[j] == page) {
            pageFound = 1;  // Page hit
            frequency[j]++; // Increment frequency of the page
            break;
        }
    }

    // If page is not found, it's a page fault
    if (!pageFound) {
        pageFaults++;

        // Find the least frequently used page to replace
        int minFrequency = frequency[0];
        int minIndex = 0;

        // Find the frame with the least frequency
        for (int j = 1; j < numFrames; j++) {
            if (frequency[j] < minFrequency) {
                minFrequency = frequency[j];
                minIndex = j;
            }
        }
```

```
            // Replace the least frequently used page
            frames[minIndex] = page;
            frequency[minIndex] = 1;  // Reset frequency of the newly loaded page
        }

        // Print the current frames after each page fault
        printf("Current frames: ");
        for (int j = 0; j < numFrames; j++) {
            printf("%d ", frames[j]);
        }
        printf("\n");
    }

    printf("\nTotal page faults: %d\n", pageFaults);

    printf("Lab No: 2\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No: 23\n");

    return 0;
}
```

Output



```
Enter the number of pages: 6
Enter the reference string (sequence of page numbers):
1 2 3 2 1 2
Enter the number of frames: 2
Current frames: 1 -1
Current frames: 1 2
Current frames: 3 2
Current frames: 3 2
Current frames: 1 2
Current frames: 1 2

Total page faults: 4
Lab No: 2
Name: Suresh Dahal
Roll No: 23
```

TITLE: Simulation of disk scheduling algorithms

THEORY

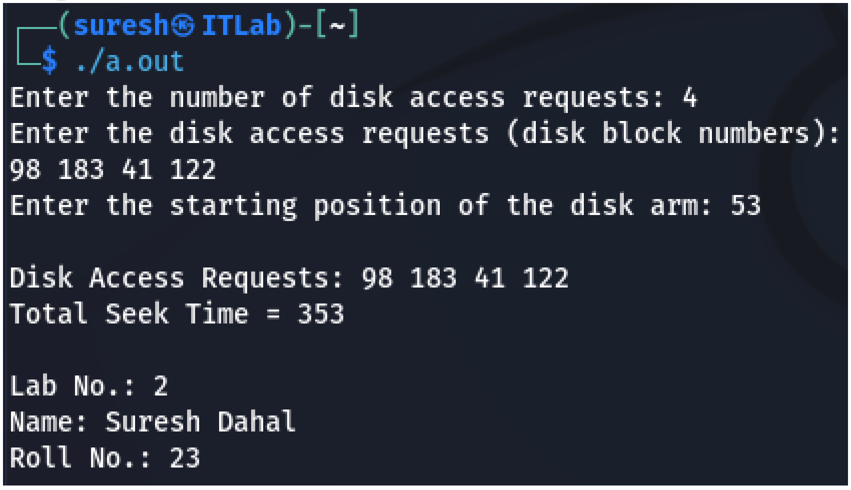
Disk scheduling algorithms manage the order of disk I/O requests to minimize seek time and improve efficiency. Common algorithms include **FCFS**, **SSTF**, **SCAN**, **C-SCAN**, and **LOOK**, each optimizing disk access in different ways based on request patterns.

a. **WAP to simulate FCFS Disk Scheduling Algorithm**

Program

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

#define MAX_REQUESTS 100

// Function to calculate the total seek time
int calculateSeekTime(int requests[], int numRequests, int start) {
    int seekTime = 0;
    int current = start;

    for (int i = 0; i < numRequests; i++) {
        seekTime += abs(current - requests[i]);
        current = requests[i];
    }

    return seekTime;
}

int main() {
    int numRequests, start, seekTime;

    // Input the number of requests
    printf("Enter the number of disk access requests: ");
    scanf("%d", &numRequests);

    int requests[numRequests];

    // Input the disk access requests
    printf("Enter the disk access requests (disk block numbers):\n");
    for (int i = 0; i < numRequests; i++) {
        scanf("%d", &requests[i]);
    }
    // Input the starting position of the disk arm
    printf("Enter the starting position of the disk arm: ");
    scanf("%d", &start);

    // Calculate the total seek time for FCFS
    seekTime = calculateSeekTime(requests, numRequests, start);

    // Display the result
    printf("\nDisk Access Requests: ");
    for (int i = 0; i < numRequests; i++) {
        printf("%d ", requests[i]);
    }

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

    // Printing lab details
    printf("\nLab No.: 2\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No.: 23\n");

    return 0;
}
```

Output



```
┌──(suresh㉿ITLab)-[~]
└─$ ./a.out
Enter the number of disk access requests: 4
Enter the disk access requests (disk block numbers):
98 183 41 122
Enter the starting position of the disk arm: 53

Disk Access Requests: 98 183 41 122
Total Seek Time = 353

Lab No.: 2
Name: Suresh Dahal
Roll No.: 23
```

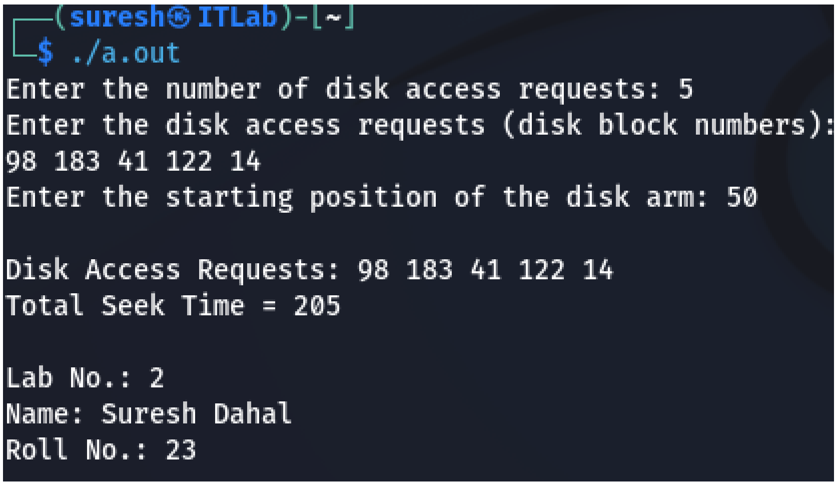
b. WAP to simulate SSTF Disk Scheduling Algorithm

Program

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

```
#define MAX_REQUESTS 100

// Function to calculate the total seek time
int calculateSeekTime(int requests[], int numRequests, int start) {
    int seekTime = 0;
    int current = start;
    int completed[numRequests];
    for (int i = 0; i < numRequests; i++) {
        completed[i] = 0; // Initialize the completed array to 0 (not completed)
    }

    int remaining = numRequests;
    while (remaining > 0) {
        int minDistance = 999999; // Set a large value for the minimum distance
        int closestRequest = -1;

        // Find the closest request to the current position
        for (int i = 0; i < numRequests; i++) {
            if (!completed[i]) {
                int distance = abs(current - requests[i]);
                if (distance < minDistance) {
                    minDistance = distance;
                    closestRequest = i;
                }
            }
        }

        // Update the seek time and the current position
        seekTime += minDistance;
        current = requests[closestRequest];
        completed[closestRequest] = 1; // Mark the request as completed
        remaining--; // Decrease the remaining requests count
    }

    return seekTime;
}

int main() {
    int numRequests, start, seekTime;

    // Input the number of requests
    printf("Enter the number of disk access requests: ");
    scanf("%d", &numRequests);

    int requests[numRequests];
    // Input the disk access requests
    printf("Enter the disk access requests (disk block numbers):\n");
    for (int i = 0; i < numRequests; i++) {
        scanf("%d", &requests[i]);
    }

    // Input the starting position of the disk arm
    printf("Enter the starting position of the disk arm: ");
    scanf("%d", &start);

    // Calculate the total seek time for SSTF
    seekTime = calculateSeekTime(requests, numRequests, start);

    // Display the result
    printf("\nDisk Access Requests: ");
    for (int i = 0; i < numRequests; i++) {
        printf("%d ", requests[i]);
    }

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

    // Printing lab details
    printf("\nLab No.: 2\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No.: 23\n");

    return 0;
}
```

Output



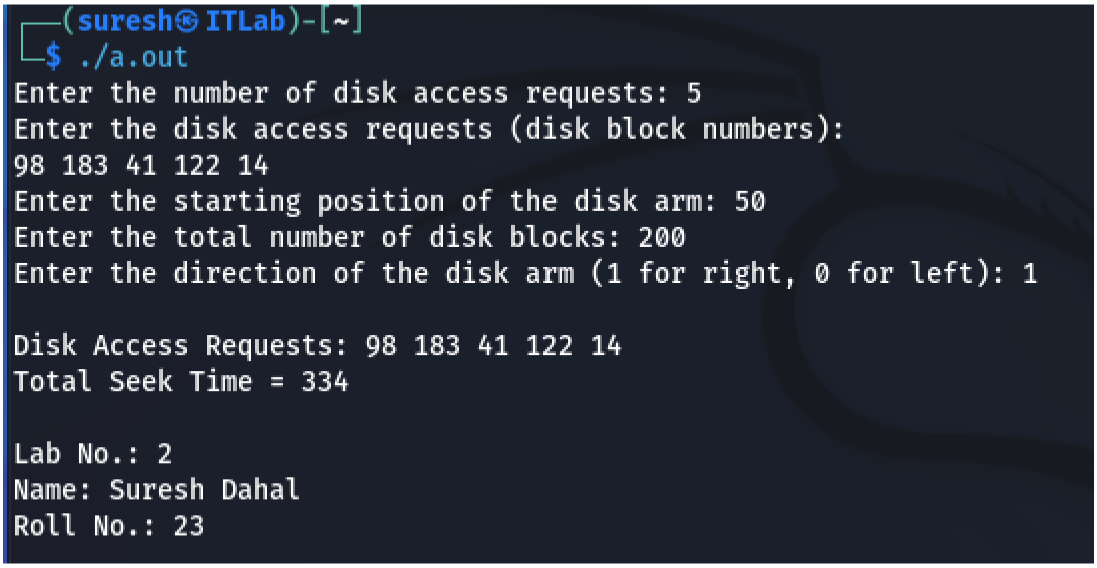
```
┌──(suresh㉿ITLab)-[~]
└─$ ./a.out
Enter the number of disk access requests: 5
Enter the disk access requests (disk block numbers):
98 183 41 122 14
Enter the starting position of the disk arm: 50

Disk Access Requests: 98 183 41 122 14
Total Seek Time = 205

Lab No.: 2
Name: Suresh Dahal
Roll No.: 23
```

c. WAP to simulate SCAN Disk Scheduling Algorithm

Program

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

#define MAX_REQUESTS 100

// Function to calculate the total seek time
int calculateSeekTime(int requests[], int numRequests, int start, int diskSize, int direction) {
    int seekTime = 0;
    int current = start;

    // Sort the requests to process them in order
    int sortedRequests[numRequests];
    for (int i = 0; i < numRequests; i++) {
        sortedRequests[i] = requests[i];
    }

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

    // Calculate the seek time
    if (direction == 1) {
        // Moving to the right (ascending order)
        for (int i = 0; i < numRequests; i++) {
            if (sortedRequests[i] >= current) {
                seekTime += abs(current - sortedRequests[i]);
                current = sortedRequests[i];
            }
        }
        // Reverse direction
        seekTime += abs(current - (diskSize - 1));
        current = diskSize - 1;

        for (int i = numRequests - 1; i >= 0; i--) {
            if (sortedRequests[i] <= current) {
                seekTime += abs(current - sortedRequests[i]);
                current = sortedRequests[i];
            }
        }
    } else {
        // Moving to the left (descending order)
        for (int i = numRequests - 1; i >= 0; i--) {
            if (sortedRequests[i] <= current) {
                seekTime += abs(current - sortedRequests[i]);
                current = sortedRequests[i];
            }
        }

        // Reverse direction
        seekTime += abs(current - 0);
        current = 0;

        for (int i = 0; i < numRequests; i++) {
            if (sortedRequests[i] >= current) {
                seekTime += abs(current - sortedRequests[i]);
                current = sortedRequests[i];
            }
        }
    }

    return seekTime;
}

int main() {
    int numRequests, start, diskSize, direction, seekTime;

    // Input the number of requests
    printf("Enter the number of disk access requests: ");
    scanf("%d", &numRequests);

    int requests[numRequests];

    // Input the disk access requests
    printf("Enter the disk access requests (disk block numbers):\n");
    for (int i = 0; i < numRequests; i++) {
        scanf("%d", &requests[i]);
    }

    // Input the starting position of the disk arm and disk size
    printf("Enter the starting position of the disk arm: ");
    scanf("%d", &start);
    printf("Enter the total number of disk blocks: ");
    scanf("%d", &diskSize);

    // Input the direction of the arm (1 for right, 0 for left)
    printf("Enter the direction of the disk arm (1 for right, 0 for left): ");
    scanf("%d", &direction);

    // Calculate the total seek time for SCAN
    seekTime = calculateSeekTime(requests, numRequests, start, diskSize, direction);

    // Display the result
    printf("\nDisk Access Requests: ");
    for (int i = 0; i < numRequests; i++) {
        printf("%d ", requests[i]);
    }

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

    // Printing lab details
    printf("\nLab No.: 2\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No.: 23\n");

    return 0;
}
```

Output



```
┌──(suresh㉿ITLab)-[~]
└─$ ./a.out
Enter the number of disk access requests: 5
Enter the disk access requests (disk block numbers):
98 183 41 122 14
Enter the starting position of the disk arm: 50
Enter the total number of disk blocks: 200
Enter the direction of the disk arm (1 for right, 0 for left): 1

Disk Access Requests: 98 183 41 122 14
Total Seek Time = 334

Lab No.: 2
Name: Suresh Dahal
Roll No.: 23
```

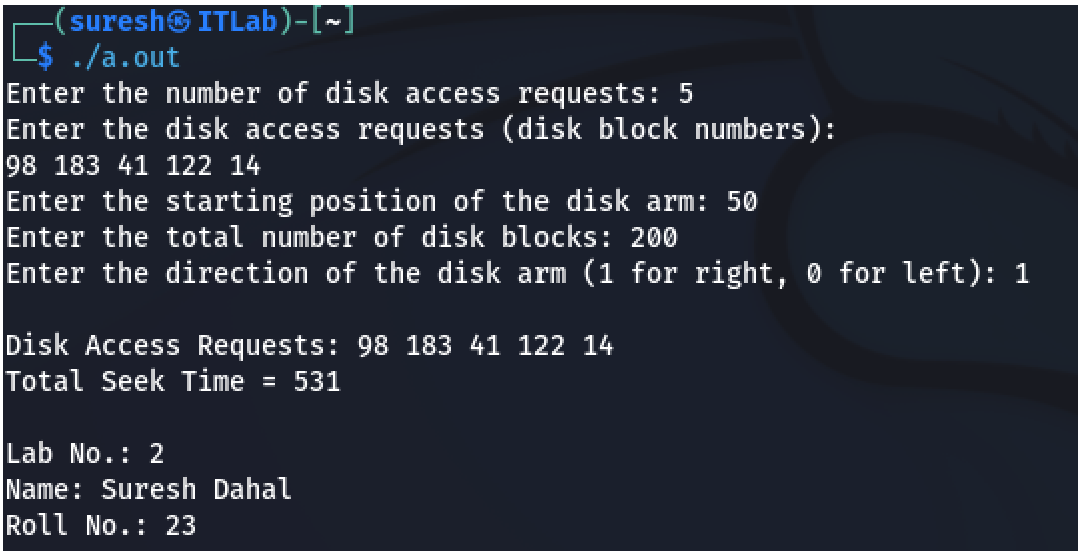
d. WAP to simulate C-SCAN Disk Scheduling Algorithm

Program

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

#define MAX_REQUESTS 100

// Function to calculate the total seek time
int calculateSeekTime(int requests[], int numRequests, int start, int diskSize, int direction) {
    int seekTime = 0;
    int current = start;

    // Sort the requests to process them in order
    int sortedRequests[numRequests];
    for (int i = 0; i < numRequests; i++) {
        sortedRequests[i] = requests[i];
    }

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

    // Calculate the seek time for C-SCAN
    if (direction == 1) {
        // Moving to the right (ascending order)
        for (int i = 0; i < numRequests; i++) {
            if (sortedRequests[i] >= current) {
                seekTime += abs(current - sortedRequests[i]);
                current = sortedRequests[i];
            }
        }

        // Move to the end of the disk
        seekTime += abs(current - (diskSize - 1));
        current = diskSize - 1;

        // Jump to the beginning of the disk and service the requests
        seekTime += abs(current - 0);
        current = 0;

        // Continue servicing the requests from the beginning
        for (int i = 0; i < numRequests; i++) {
            if (sortedRequests[i] >= current) {
                seekTime += abs(current - sortedRequests[i]);
                current = sortedRequests[i];
            }
        }
    } else {
        // Moving to the left (descending order)
        for (int i = numRequests - 1; i >= 0; i--) {
            if (sortedRequests[i] <= current) {
                seekTime += abs(current - sortedRequests[i]);
                current = sortedRequests[i];
            }
        }

        // Move to the beginning of the disk
        seekTime += abs(current - 0);
        current = 0;

        // Jump to the end of the disk and service the requests
        seekTime += abs(current - (diskSize - 1));
        current = diskSize - 1;

        // Continue servicing the requests from the end
```

```
        for (int i = numRequests - 1; i >= 0; i--) {
            if (sortedRequests[i] <= current) {
                seekTime += abs(current - sortedRequests[i]);
                current = sortedRequests[i];
            }
        }
    }

    return seekTime;
}

int main() {
    int numRequests, start, diskSize, direction, seekTime;

    // Input the number of requests
    printf("Enter the number of disk access requests: ");
    scanf("%d", &numRequests);

    int requests[numRequests];

    // Input the disk access requests
    printf("Enter the disk access requests (disk block numbers):\n");
    for (int i = 0; i < numRequests; i++) {
        scanf("%d", &requests[i]);
    }

    // Input the starting position of the disk arm and disk size
    printf("Enter the starting position of the disk arm: ");
    scanf("%d", &start);
    printf("Enter the total number of disk blocks: ");
    scanf("%d", &diskSize);

    // Input the direction of the arm (1 for right, 0 for left)
    printf("Enter the direction of the disk arm (1 for right, 0 for left): ");
    scanf("%d", &direction);

    // Calculate the total seek time for C-SCAN
    seekTime = calculateSeekTime(requests, numRequests, start, diskSize, direction);

    // Display the result
    printf("\nDisk Access Requests: ");
    for (int i = 0; i < numRequests; i++) {
        printf("%d ", requests[i]);
    }

    printf("\nTotal Seek Time = %d\n", seekTime);
    // Printing lab details
    printf("\nLab No.: 2\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No.: 23\n");

    return 0;
}
```

Output



```
┌──(suresh㉿ITLab)-[~]
└─$ ./a.out
Enter the number of disk access requests: 5
Enter the disk access requests (disk block numbers):
98 183 41 122 14
Enter the starting position of the disk arm: 50
Enter the total number of disk blocks: 200
Enter the direction of the disk arm (1 for right, 0 for left): 1

Disk Access Requests: 98 183 41 122 14
Total Seek Time = 531

Lab No.: 2
Name: Suresh Dahal
Roll No.: 23
```

e. WAP to simulate LOOK Disk Scheduling Algorithm

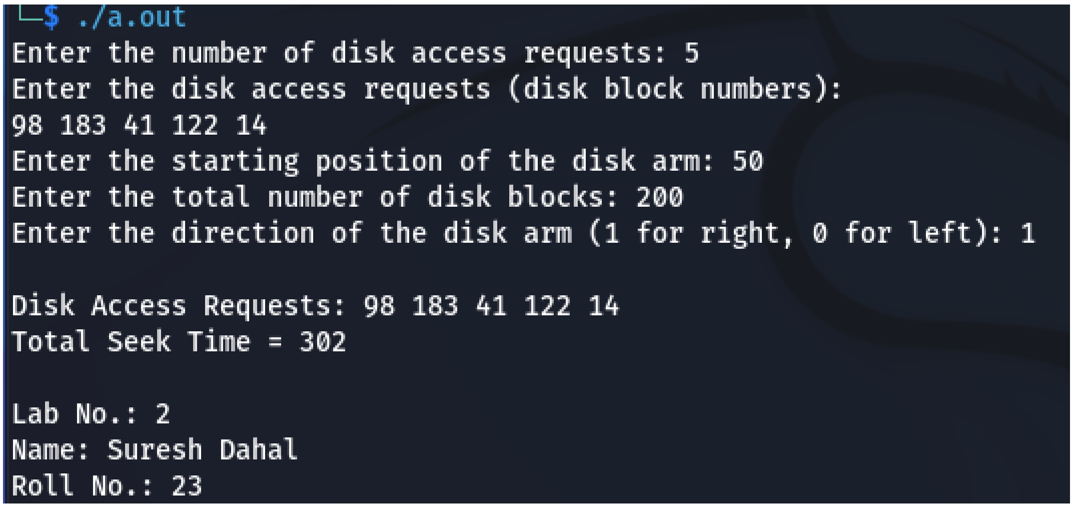
Program

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

#define MAX_REQUESTS 100

// Function to calculate the total seek time
```

```
int calculateSeekTime(int requests[], int numRequests, int start, int diskSize, int
direction) {
   int seekTime = 0;
   int current = start;

   // Sort the requests to process them in order
   int sortedRequests[numRequests];
   for (int i = 0; i < numRequests; i++) {
      sortedRequests[i] = requests[i];
   }

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

   // Calculate the seek time for LOOK
   if (direction == 1) {
      // Moving to the right (ascending order)
      for (int i = 0; i < numRequests; i++) {
         if (sortedRequests[i] >= current) {
            seekTime += abs(current - sortedRequests[i]);
            current = sortedRequests[i];
         }
      }

      // Reverse direction and move to the left
      for (int i = numRequests - 1; i >= 0; i--) {
         if (sortedRequests[i] < current) {
            seekTime += abs(current - sortedRequests[i]);
            current = sortedRequests[i];
         }
      }
   } else {
      // Moving to the left (descending order)
      for (int i = numRequests - 1; i >= 0; i--) {
         if (sortedRequests[i] <= current) {
            seekTime += abs(current - sortedRequests[i]);
            current = sortedRequests[i];
         }
```

```
        }

        // Reverse direction and move to the right
        for (int i = 0; i < numRequests; i++) {
            if (sortedRequests[i] > current) {
                seekTime += abs(current - sortedRequests[i]);
                current = sortedRequests[i];
            }
        }
    }

    return seekTime;
}

int main() {
    int numRequests, start, diskSize, direction, seekTime;

    // Input the number of requests
    printf("Enter the number of disk access requests: ");
    scanf("%d", &numRequests);

    int requests[numRequests];

    // Input the disk access requests
    printf("Enter the disk access requests (disk block numbers):\n");
    for (int i = 0; i < numRequests; i++) {
        scanf("%d", &requests[i]);
    }

    // Input the starting position of the disk arm and disk size
    printf("Enter the starting position of the disk arm: ");
    scanf("%d", &start);
    printf("Enter the total number of disk blocks: ");
    scanf("%d", &diskSize);

    // Input the direction of the arm (1 for right, 0 for left)
    printf("Enter the direction of the disk arm (1 for right, 0 for left): ");
    scanf("%d", &direction);

    // Calculate the total seek time for LOOK
    seekTime = calculateSeekTime(requests, numRequests, start, diskSize, direction);

    // Display the result
    printf("\nDisk Access Requests: ");
    for (int i = 0; i < numRequests; i++) {
        printf("%d ", requests[i]);
    }

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

    // Printing lab details
    printf("\nLab No.: 2\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No.: 23\n");

    return 0;
}
```

Output



```
└─$ ./a.out
Enter the number of disk access requests: 5
Enter the disk access requests (disk block numbers):
98 183 41 122 14
Enter the starting position of the disk arm: 50
Enter the total number of disk blocks: 200
Enter the direction of the disk arm (1 for right, 0 for left): 1

Disk Access Requests: 98 183 41 122 14
Total Seek Time = 302

Lab No.: 2
Name: Suresh Dahal
Roll No.: 23
```

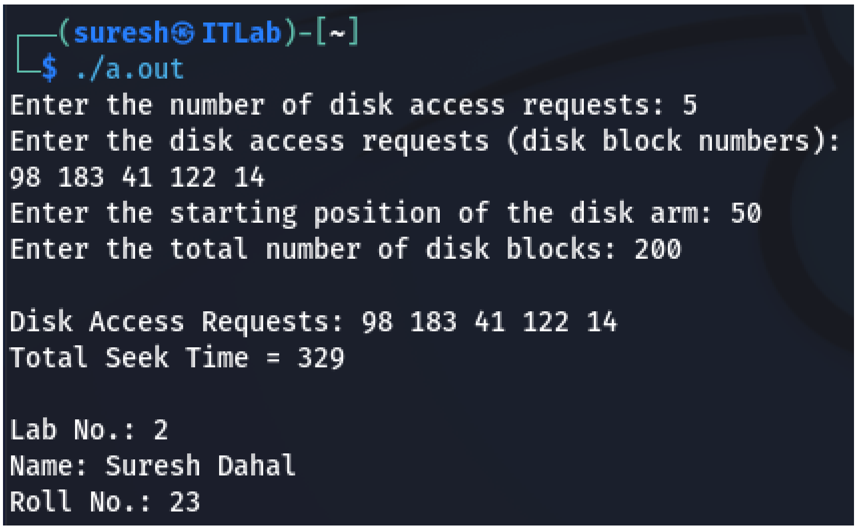
f. **WAP to simulate C-LOOK Disk Scheduling Algorithm**

Program

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

#define MAX_REQUESTS 100

// Function to calculate the total seek time using C-LOOK
int calculateSeekTime(int requests[], int numRequests, int start, int diskSize) {
    int seekTime = 0;
    int current = start;

    // Sort the requests in ascending order
    int sortedRequests[numRequests];
    for (int i = 0; i < numRequests; i++) {
        sortedRequests[i] = requests[i];
    }

    for (int i = 0; i < numRequests - 1; i++) {
        for (int j = 0; j < numRequests - i - 1; j++) {
            if (sortedRequests[j] > sortedRequests[j + 1]) {
                int temp = sortedRequests[j];
                sortedRequests[j] = sortedRequests[j + 1];
                sortedRequests[j + 1] = temp;
            }
        }
    }

    // Find the total seek time for C-LOOK
    int i;
    // Move towards the right (ascending order)
    for (i = 0; i < numRequests; i++) {
        if (sortedRequests[i] >= current) {
            break;
        }
    }

    // Travel to the last request in the current direction
    for (int j = i; j < numRequests; j++) {
        seekTime += abs(current - sortedRequests[j]);
        current = sortedRequests[j];
    }

    // Jump to the first request and travel back in the same direction
    seekTime += abs(current - sortedRequests[0]);
    current = sortedRequests[0];

    // Now travel from the first to the last request in ascending order
    for (int j = 1; j < i; j++) {
        seekTime += abs(current - sortedRequests[j]);
        current = sortedRequests[j];
    }

    return seekTime;
}
```

```
int main() {
    int numRequests, start, diskSize, seekTime;

    // Input the number of requests
    printf("Enter the number of disk access requests: ");
    scanf("%d", &numRequests);

    int requests[numRequests];

    // Input the disk access requests
    printf("Enter the disk access requests (disk block numbers):\n");
    for (int i = 0; i < numRequests; i++) {
        scanf("%d", &requests[i]);
    }

    // Input the starting position of the disk arm and disk size
    printf("Enter the starting position of the disk arm: ");
    scanf("%d", &start);
    printf("Enter the total number of disk blocks: ");
    scanf("%d", &diskSize);

    // Calculate the total seek time for C-LOOK
    seekTime = calculateSeekTime(requests, numRequests, start, diskSize);

    // Display the result
    printf("\nDisk Access Requests: ");
    for (int i = 0; i < numRequests; i++) {
        printf("%d ", requests[i]);
    }

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

    // Printing lab details
    printf("\nLab No.: 2\n");
    printf("Name: Suresh Dahal\n");
    printf("Roll No.: 23\n");

    return 0;
}
```

Output



```
┌──(suresh㉿ITLab)-[~]
└─$ ./a.out
Enter the number of disk access requests: 5
Enter the disk access requests (disk block numbers):
98 183 41 122 14
Enter the starting position of the disk arm: 50
Enter the total number of disk blocks: 200

Disk Access Requests: 98 183 41 122 14
Total Seek Time = 329

Lab No.: 2
Name: Suresh Dahal
Roll No.: 23
```