EXPERIMENT NO. 1

AIM / TITLE: To study basic Commands in Linux Operating System.

This is a list of the most commonly used Linux commands and their usage.

1. Ls
   Syntax : `ls [options] [directory]`
   Usage : `ls` is used to list the contents of a directory. It can be used to list the contents of the current directory or any other directory.
2. Cd
   Syntax : `cd [directory]`
   Usage : `cd` is used to change the current working directory. It can be used to change the current working directory to the home directory or any other directory.
3. Grep
   Syntax : `grep [options] pattern [file]`
   Usage : `grep` is used to search for a pattern in a file. It can be used to search for a pattern in a file or in the output of a command.
4. su/ sudo
   Syntax : `su [options] [username]` or `sudo [options] command`
   Usage : `su` is used to switch to another user. It can be used to switch to the root user or any other user.
   `sudo` is used to execute a command as another user. It can be used to execute a command as the root user or any other user.
5. Pwd
   Syntax : `pwd`
   Usage: `pwd` is used to print the current working directory.
6. Mv
   Syntax : `mv [options] source destination`
   Usage : `mv` is used to move or rename a file or directory. It can be used to move or rename a file or directory in the same directory or in a different directory.
7. Cp
   Syntax : `cp [options] source destination`
   Usage : `cp` is used to copy a file or directory. It can be used to copy a file or directory in the same directory or in a different directory.
8. Rm
   Syntax : `rm [options] file`
   Usage : `rm` is used to remove a file or directory. It can be used to remove a file or directory in the same directory or in a different directory.
9. Mkdir
   Syntax : `mkdir [options] directory`
   Usage : `mkdir` is used to create a directory.
10. Rmdir
    Syntax : `rmdir [options] directory`
    Usage : `rmdir` is used to remove a directory.
11. Chmod
    Syntax : `chmod [options] permissions file`
    Usage : `chmod` is used to change the permissions of a file or directory.
12. Cat
    Syntax : `cat [options] file1 file2`
    Usage : `cat` is used to concatenate files and print on the standard output.

13. Echo
    Syntax : `echo [options] string`
    Usage : `echo` is used to display a line of text/string.
14. Wc
    Syntax : `wc [options] file`
    Usage : `wc` is used to count the number of lines, words, and bytes in a file.
15. Man
    Syntax : `man [options] command`
    Usage : `man` is used to display the user manual of any command that we can run on the terminal.
16. History
    Syntax : `history [options]`
    Usage : `history` is used to print the history of commands.
17. Clear
    Syntax : `clear`
    Usage : `clear` is used to clear the terminal screen.
18. Apt-get
    Syntax : `apt-get [options]`
    Usage : `apt-get` is used to install, remove, and update software packages on Debian based systems.
19. Ping
    Syntax : `ping [options] hostname`
    Usage : `ping` is used to test the reachability of a host on an Internet Protocol (IP) network.
20. Find
    Syntax : `find [options] path`
    Usage : `find` is used to search for files in a directory hierarchy.

EXPERIMENT NO. 2

AIM / TITLE: To develop a program to implement Thread and Process.

Processes: A process is an instance of a program in execution. Each process has its own memory space and resources, such as CPU time, file handles, and so on. Processes are isolated from each other; one process cannot directly access the memory of another process. Communication between processes (Inter-Process Communication, IPC) usually requires special mechanisms like pipes, message queues, shared memory, etc.

Threads: A thread is a smaller unit of a process that can be scheduled for execution. Multiple threads can exist within the same process, sharing the same memory space. Threads can communicate with each other more easily than processes since they share the same memory space. Each thread has its own stack, program counter, and local variables, but shares global variables and heap memory with other threads in the same process.

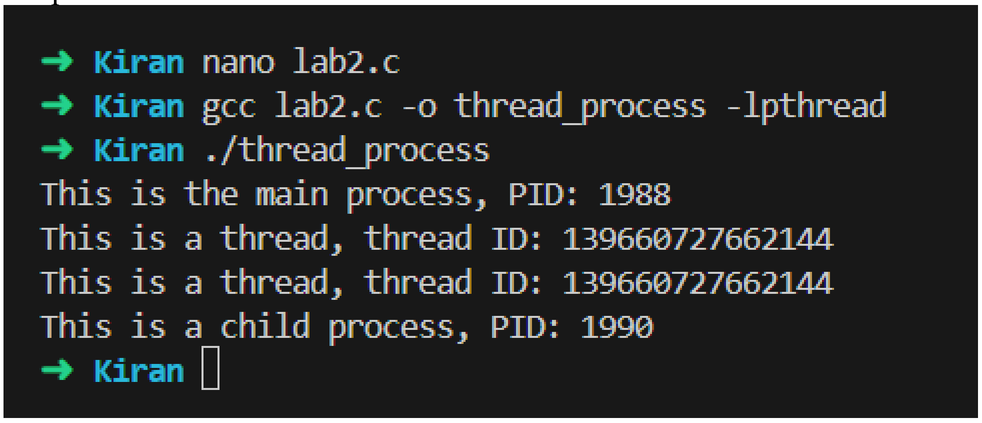
Differences between Threads and Processes:

- **Memory:** Threads share the same memory within a process, while processes have their own separate memory spaces.
- **Creation:** Creating threads is generally faster than creating processes because threads are lighter weight.
- **Communication:** Threads can communicate with each other more easily than processes.
- **Isolation:** Processes are more isolated, making them more secure but harder to communicate with.

C Program: Processes creation and termination in LINUX

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
void* thread_function(void* arg) {
   printf("This is a thread, thread ID: %ld\n", pthread_self());
   return NULL;
}
int main() {
   pthread_t thread;
   pthread_create(&thread, NULL, thread_function, NULL);
   printf("This is the main process, PID: %d\n", getpid());
   if (fork() == 0) {
      printf("This is a child process, PID: %d\n", getpid());
   }
   pthread_join(thread, NULL); // Wait for the thread to finish
   return 0;
}
```

Output:



```
➜ Kiran nano lab2.c
➜ Kiran gcc lab2.c -o thread_process -lpthread
➜ Kiran ./thread_process
This is the main process, PID: 1988
This is a thread, thread ID: 139660727662144
This is a thread, thread ID: 139660727662144
This is a child process, PID: 1990
➜ Kiran
```

EXPERIMENT NO. 3

AIM / TITLE: Write a C program to implementation of producer-consumer problem using Semaphores.

The Producer-Consumer problem is a classical example of a multi-process synchronization problem. The problem describes two types of processes, the producer and the consumer, sharing a common, fixed-size buffer used as a queue. The producer's job is to generate data and place it into the buffer, while the consumer's job is to consume the data from the buffer. The challenge is to make sure that the producer does not try to add data into the buffer if it's full, and the consumer does not try to remove data from an empty buffer.

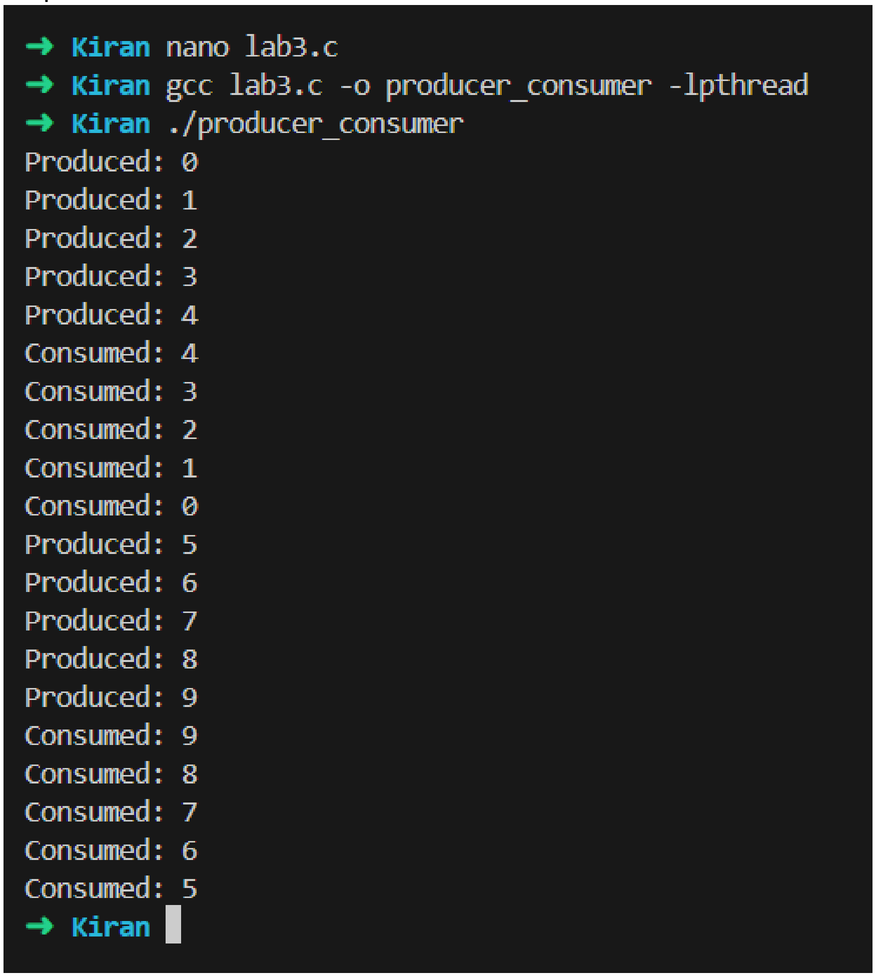
Semaphores are used to solve this problem by ensuring mutual exclusion and proper synchronization between the producer and consumer processes.

Source Code

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

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int count = 0;

sem_t empty, full, mutex;

void* producer(void* arg) {
   for (int i = 0; i < 10; i++) {
      sem_wait(&empty);
      sem_wait(&mutex);

      buffer[count++] = i;
      printf("Produced: %d\n", i);

      sem_post(&mutex);
      sem_post(&full);
   }
}

void* consumer(void* arg) {
   for (int i = 0; i < 10; i++) {
      sem_wait(&full);
      sem_wait(&mutex);

      int item = buffer[--count];
      printf("Consumed: %d\n", item);

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

```
int main() {
    pthread_t prod, cons;

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

    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

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

    return 0;
}
```

Output:



```
➜ Kiran nano lab3.c
➜ Kiran gcc lab3.c -o producer_consumer -lpthread
➜ Kiran ./producer_consumer
Produced: 0
Produced: 1
Produced: 2
Produced: 3
Produced: 4
Consumed: 4
Consumed: 3
Consumed: 2
Consumed: 1
Consumed: 0
Produced: 5
Produced: 6
Produced: 7
Produced: 8
Produced: 9
Consumed: 9
Consumed: 8
Consumed: 7
Consumed: 6
Consumed: 5
➜ Kiran
```

EXPERIMENT NO. 4

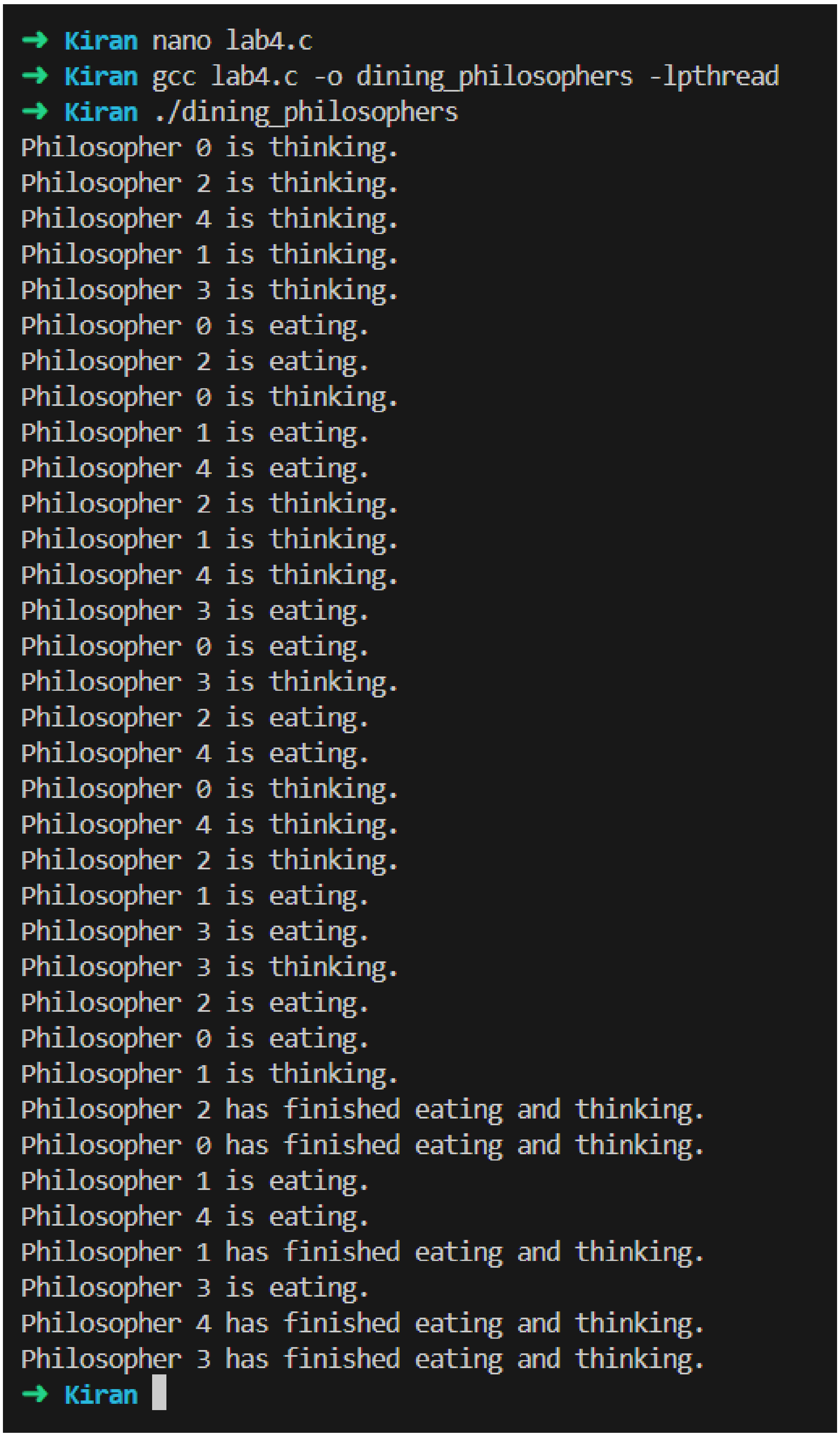
AIM / TITLE: Write a C program to Simulate the concept of Dining-philosophers problem.

The Dining Philosophers problem is a classic synchronization problem in computer science that involves philosophers who do either of two activities: thinking or eating. Each philosopher needs two forks (or chopsticks) to eat, but there's a limited number of forks, so they must share them. The challenge is to ensure that no philosopher starves and that no deadlock occurs.

Source Code:

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define N 5
#define CYCLES 3

sem_t forks[N];
void* philosopher(void* num) {
int id = *(int*)num;
for (int i = 0; i < CYCLES; i++) {
printf("Philosopher %d is thinking.\n", id);
sleep(1);  // Thinking for a while
sem_wait(&forks[id]);
sem_wait(&forks[(id + 1) % N]);
printf("Philosopher %d is eating.\n", id);
sleep(1);  // Eating for a while
sem_post(&forks[id]);
sem_post(&forks[(id + 1) % N]);
}
printf("Philosopher %d has finished eating and thinking.\n", id);
return NULL;
}
int main()
{
pthread_t philosophers[N];
int ids[N];
for (int i = 0; i < N; i++) {
sem_init(&forks[i], 0, 1);
ids[i] = i;
}
for (int i = 0; i < N; i++) {
      pthread_create(&philosophers[i], NULL, philosopher, &ids[i]);
   }
for (int i = 0; i < N; i++) {
 pthread_join(philosophers[i], NULL);
}
for (int i = 0; i < N; i++) {
sem_destroy(&forks[i]);
 }
return 0;}
```

Output:



```
➜ Kiran nano lab4.c
➜ Kiran gcc lab4.c -o dining_philosophers -lpthread
➜ Kiran ./dining_philosophers
Philosopher 0 is thinking.
Philosopher 2 is thinking.
Philosopher 4 is thinking.
Philosopher 1 is thinking.
Philosopher 3 is thinking.
Philosopher 0 is eating.
Philosopher 2 is eating.
Philosopher 0 is thinking.
Philosopher 1 is eating.
Philosopher 4 is eating.
Philosopher 2 is thinking.
Philosopher 1 is thinking.
Philosopher 4 is thinking.
Philosopher 3 is eating.
Philosopher 0 is eating.
Philosopher 3 is thinking.
Philosopher 2 is eating.
Philosopher 4 is eating.
Philosopher 0 is thinking.
Philosopher 4 is thinking.
Philosopher 2 is thinking.
Philosopher 1 is eating.
Philosopher 3 is eating.
Philosopher 3 is thinking.
Philosopher 2 is eating.
Philosopher 0 is eating.
Philosopher 1 is thinking.
Philosopher 2 has finished eating and thinking.
Philosopher 0 has finished eating and thinking.
Philosopher 1 is eating.
Philosopher 4 is eating.
Philosopher 1 has finished eating and thinking.
Philosopher 3 is eating.
Philosopher 4 has finished eating and thinking.
Philosopher 3 has finished eating and thinking.
➜ Kiran
```

EXPERIMENT NO. 5

AIM / TITLE: Write a C program to simulate the following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time.

a) FCFS **b) SJF** **c) Priority**

CPU scheduling is a critical aspect of operating systems that determines the order in which processes are executed by the CPU. Non-preemptive scheduling algorithms are a type where, once a process starts execution, it runs to completion without being interrupted. This contrasts with preemptive scheduling, where a process can be interrupted and replaced by another process.

In this experiment, we focus on three key non-preemptive scheduling algorithms: First-Come, First-Served (FCFS), Shortest Job First (SJF), and Priority Scheduling. These algorithms are widely studied for their simplicity and are fundamental in understanding how operating systems manage process execution.

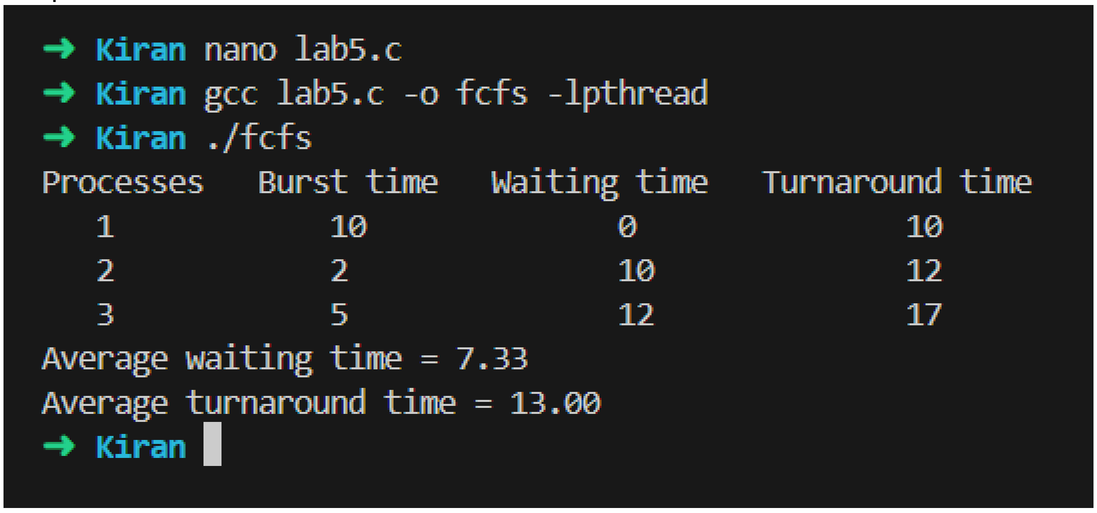
1. First-Come, First-Served (FCFS) Scheduling

FCFS is the simplest scheduling algorithm. The process that arrives first in the ready queue is executed first. This approach is easy to implement but may lead to the "convoy effect," where short processes wait for a long process to finish, leading to suboptimal CPU utilization.

```
#include <stdio.h>
void findWaitingTime(int processes[], int n, int bt[], int wt[]) {
   wt[0] = 0;
   for (int i = 1; i < n; i++) {
      wt[i] = bt[i-1] + wt[i-1];
   }
}

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 wt[n], tat[n];
   int total_wt = 0, total_tat = 0;

   findWaitingTime(processes, n, bt, wt);
   findTurnaroundTime(processes, n, bt, wt, tat);

   printf("Processes   Burst time   Waiting time   Turnaround time\n");
   for (int i = 0; i < n; i++) {
      total_wt += wt[i];
```

```
        total_tat += tat[i];
        printf("   %d ", (i + 1));
        printf("\t\t%d ", bt[i]);
        printf("\t\t%d", wt[i]);
        printf("\t\t%d\n", tat[i]);
    }

    printf("Average waiting time = %.2f\n", (float)total_wt / n);
    printf("Average turnaround time = %.2f\n", (float)total_tat / n);
}

int main() {
    int processes[] = {1, 2, 3};
    int n = sizeof processes / sizeof processes[0];
    int burst_time[] = {10, 2, 5};

    findAverageTime(processes, n, burst_time);
    return 0;
}
```

Output:



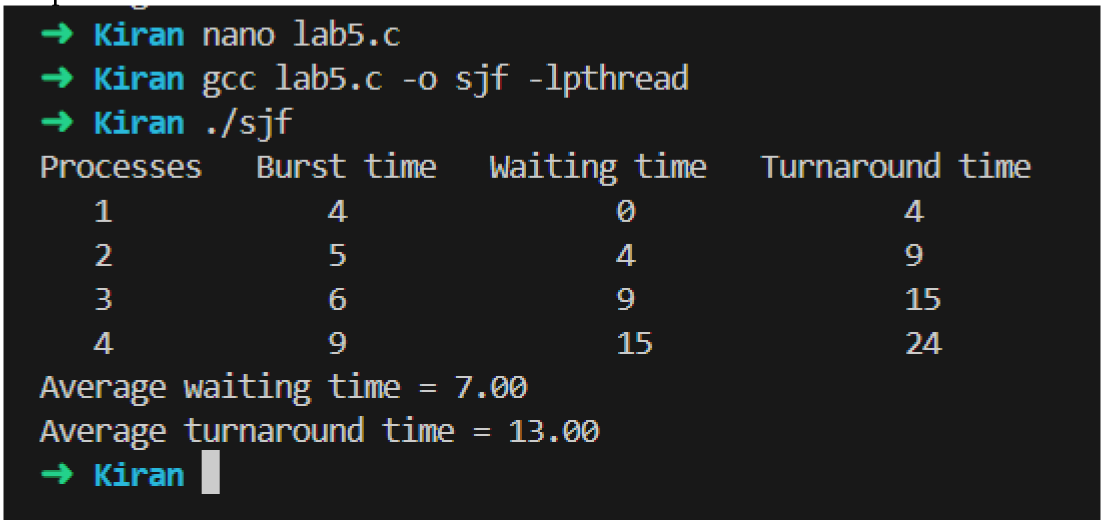
```
➜ Kiran nano lab5.c
➜ Kiran gcc lab5.c -o fcfs -lpthread
➜ Kiran ./fcfs
Processes   Burst time   Waiting time   Turnaround time
   1            10            0              10
   2            2             10             12
   3            5             12             17
Average waiting time = 7.33
Average turnaround time = 13.00
➜ Kiran
```

2. Shortest Job First (SJF) Scheduling

SJF is a scheduling algorithm where the process with the shortest burst time is selected for execution next. This approach minimizes the average waiting time but requires knowledge of the burst time of all processes, which may not always be possible.

```
#include <stdio.h>
// Function to sort processes by burst time
void sortByBurstTime(int n, int processes[], int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (bt[i] > bt[j]) {
                // Swap burst time
                int temp = bt[i];
                bt[i] = bt[j];
                bt[j] = temp;// Swap process number
temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }
}
void findWaitingTime(int processes[], int n, int bt[], int wt[]) {
wt[0] = 0;  // First process has no waiting time
for (int i = 1; i < n; i++) {
wt[i] = bt[i - 1] + wt[i - 1];
}
}
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 wt[n], tat[n];
    int total_wt = 0, total_tat = 0;

    sortByBurstTime(n, processes, bt, wt, tat);  // Sort processes by burst time
    findWaitingTime(processes, n, bt, wt);
    findTurnaroundTime(processes, n, bt, wt, tat);

    printf("Processes   Burst time   Waiting time   Turnaround time\n");
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf("   %d ", processes[i]);
        printf("\t\t%d ", bt[i]);
        printf("\t\t%d", wt[i]);
        printf("\t\t%d\n", tat[i]);
    }
    printf("Average waiting time = %.2f\n", (float)total_wt / n);
    printf("Average turnaround time = %.2f\n", (float)total_tat / n);
}

int main() {
    int processes[] = {1, 2, 3, 4};
    int n = sizeof(processes) / sizeof(processes[0]);

    int burst_time[] = {4, 5, 6, 9};  // Burst time of processes

    findAverageTime(processes, n, burst_time);

    return 0;
}
```

Output:



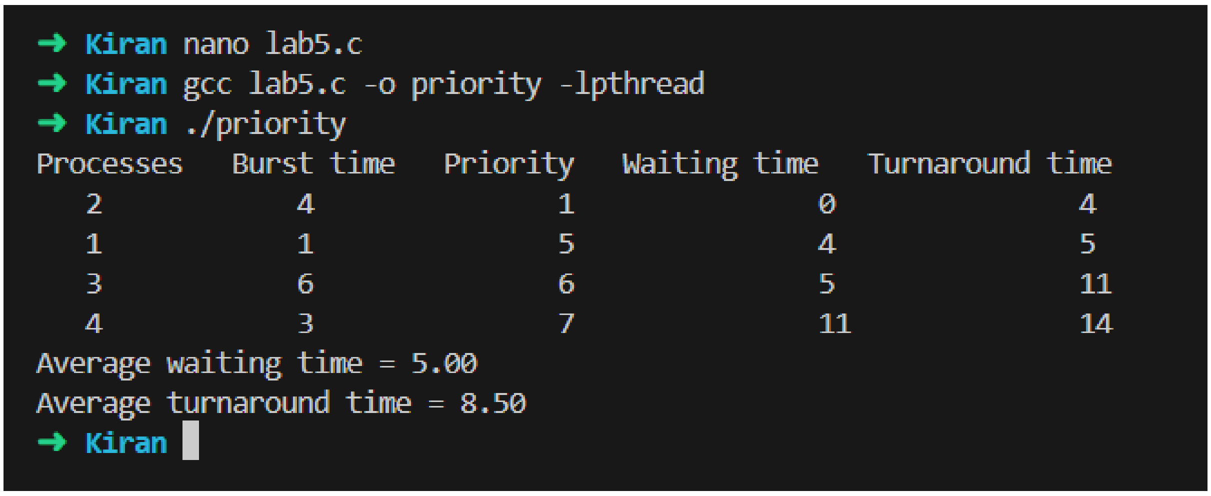
```
➜ Kiran nano lab5.c
➜ Kiran gcc lab5.c -o sjf -lpthread
➜ Kiran ./sjf
Processes    Burst time    Waiting time    Turnaround time
   1            4               0                4
   2            5               4                9
   3            6               9                15
   4            9               15               24
Average waiting time = 7.00
Average turnaround time = 13.00
➜ Kiran
```

3. Priority Scheduling

In Priority Scheduling, each process is assigned a priority, and the process with the highest priority (typically the lowest number) is selected for execution. If two processes have the same priority, FCFS order is used.

```
#include <stdio.h>

// Function to sort processes by priority
void sortByPriority(int n, int processes[], int bt[], int pr[], int wt[], int tat[]) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (pr[i] > pr[j]) {
                // Swap priority
                int temp = pr[i];
                pr[i] = pr[j];
                pr[j] = temp;

                // Swap burst time
                temp = bt[i];
                bt[i] = bt[j];
                bt[j] = temp;

                // Swap process number
                temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }
}

void findWaitingTime(int processes[], int n, int bt[], int wt[]) {
    wt[0] = 0;  // First process has no waiting time

    for (int i = 1; i < n; i++) {
        wt[i] = bt[i - 1] + wt[i - 1];
    }
}

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 pr[]) {
    int wt[n], tat[n];
    int total_wt = 0, total_tat = 0;

    sortByPriority(n, processes, bt, pr, wt, tat);  // Sort processes by priority
    findWaitingTime(processes, n, bt, wt);
    findTurnaroundTime(processes, n, bt, wt, tat);

    printf("Processes   Burst time   Priority   Waiting time   Turnaround time\n");
    for (int i = 0; i < n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf("   %d ", processes[i]);
        printf("\t\t%d ", bt[i]);
        printf("\t\t%d", pr[i]);
        printf("\t\t%d", wt[i]);
        printf("\t\t%d\n", tat[i]);
    }

    printf("Average waiting time = %.2f\n", (float)total_wt / n);
    printf("Average turnaround time = %.2f\n", (float)total_tat / n);
}

int main() {
    int processes[] = {1, 2, 3, 4};
    int n = sizeof(processes) / sizeof(processes[0]);

    int burst_time[] = {1, 4, 6, 3};  // Burst time of processes
    int priority[] = {5, 1, 6, 7};     // Priority of processes

    findAverageTime(processes, n, burst_time, priority);

    return 0;
}
```

Output:



```
➜ Kiran nano lab5.c
➜ Kiran gcc lab5.c -o priority -lpthread
➜ Kiran ./priority
Processes   Burst time   Priority   Waiting time   Turnaround time
   2            4            1             0              4
   1            1            5             4              5
   3            6            6             5              11
   4            3            7             11             14
Average waiting time = 5.00
Average turnaround time = 8.50
➜ Kiran
```

EXPERIMENT NO. 6

Aim/Title: Write a C program to simulate the following preemptive CPU scheduling algorithms to find turnaround time and waiting time.
a) Round Robin b) Priority

a) Round Robin (RR) Scheduling

Round Robin is a preemptive CPU scheduling algorithm where each process is assigned a fixed time slice, known as a quantum. The processes are scheduled in a cyclic order, and if a process's burst time exceeds the quantum, it is preempted and added to the end of the queue. The process will resume when its turn comes again. This algorithm is commonly used in time-sharing systems.

- **Advantages**: Fairness, as each process gets an equal share of the CPU.
- **Disadvantages**: High context-switching overhead, especially with a small quantum.
- **Key Terms**:
    - **Turnaround Time**: The total time taken from the submission of a process to its completion.
    - **Waiting Time**: The total time a process spends waiting in the ready queue.

Algorithm
Input: List of processes with burst times, and the time quantum.

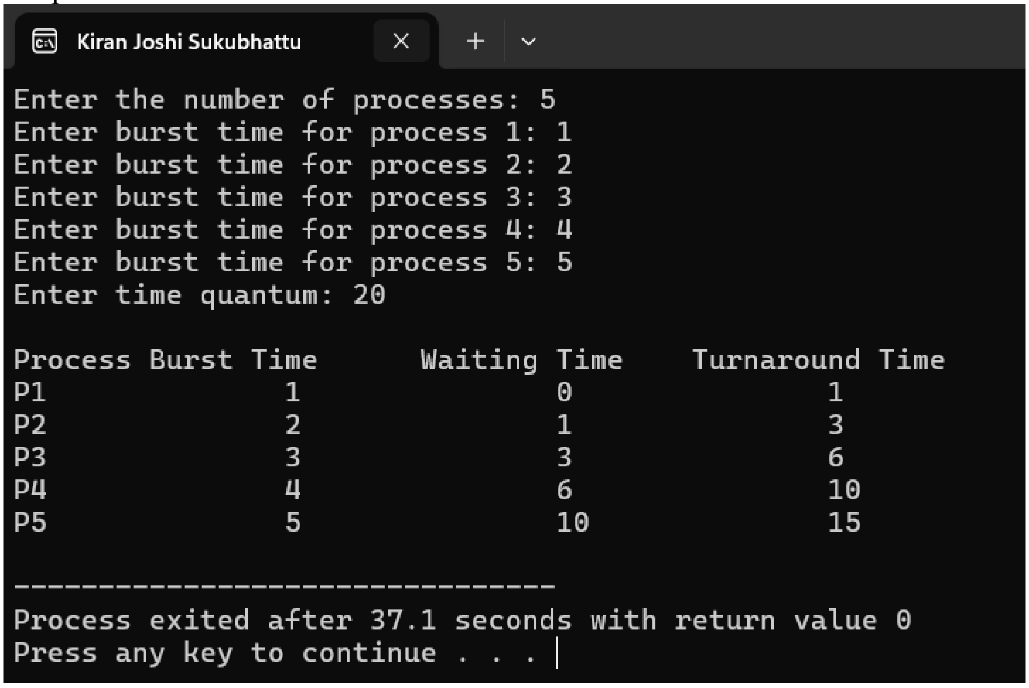
```
1. Initialize the current time t = 0.
2. While there are uncompleted processes:
     • For each process P in the list:
          • If P has remaining burst time greater than 0:
               • If P's remaining burst time is greater than the quantum:
                    ▪ Increment t by the quantum.
                    ▪ Subtract the quantum from P's remaining burst time.
               • Else:
                    ▪ Increment t by P's remaining burst time.
                    ▪ Set P's remaining burst time to 0.
                    ▪ Calculate P's waiting time as t - burst_time[P].
                    ▪ Calculate P's turnaround time as t.

3. Output the waiting time and turnaround time for each process.
```

Time Complexity: O(n*q), where n is the number of processes and q is the number of quanta.

Source Code:

```
#include <stdio.h>

struct Process {
   int pid;
   int burst_time;
   int remaining_time;
   int waiting_time;
   int turnaround_time;
};

void roundRobin(struct Process proc[], int n, int quantum) {
    int t = 0;
    int completed = 0;

    while (completed != n) {
        for (int i = 0; i < n; i++) {
            if (proc[i].remaining_time > 0) {
                if (proc[i].remaining_time > quantum) {
                    t += quantum;
                    proc[i].remaining_time -= quantum;
                } else {
                    t += proc[i].remaining_time;
                    proc[i].waiting_time = t - proc[i].burst_time;
                    proc[i].remaining_time = 0;
                    proc[i].turnaround_time = t;
                    completed++;
                }
            }
        }
    }
}


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

    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("Enter burst time for process %d: ", proc[i].pid);
        scanf("%d", &proc[i].burst_time);
        proc[i].remaining_time = proc[i].burst_time;
        proc[i].waiting_time = 0;
    }
    printf("Enter time quantum: ");
    scanf("%d", &quantum);

    roundRobin(proc, n, quantum);

    printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t\t%d\t\t%d\t\t%d\n", proc[i].pid, proc[i].burst_time, proc[i].waiting_time,
proc[i].turnaround_time);
    }

    return 0;
}
```

Output:



```
Enter the number of processes: 5
Enter burst time for process 1: 1
Enter burst time for process 2: 2
Enter burst time for process 3: 3
Enter burst time for process 4: 4
Enter burst time for process 5: 5
Enter time quantum: 20

Process Burst Time      Waiting Time    Turnaround Time
P1              1               0               1
P2              2               1               3
P3              3               3               6
P4              4               6               10
P5              5               10              15

--------------------------------
Process exited after 37.1 seconds with return value 0
Press any key to continue . . .
```

b) Priority Scheduling

Priority Scheduling is a preemptive CPU scheduling algorithm where each process is assigned a priority. The CPU is allocated to the process with the highest priority (usually the lowest numerical value). If two processes have the same priority, they are scheduled based on their order in the ready queue. Lower-priority processes are preempted by higher-priority ones.

- **Advantages**: Important processes can be executed first.
- **Disadvantages**: Can lead to starvation of low-priority processes.
- **Key Terms**:
  - **Priority**: A value assigned to a process that determines its importance relative to other processes.
  - **Turnaround Time**: The total time taken from the submission of a process to its completion.
  - **Waiting Time**: The total time a process spends waiting in the ready queue.

Algorithm

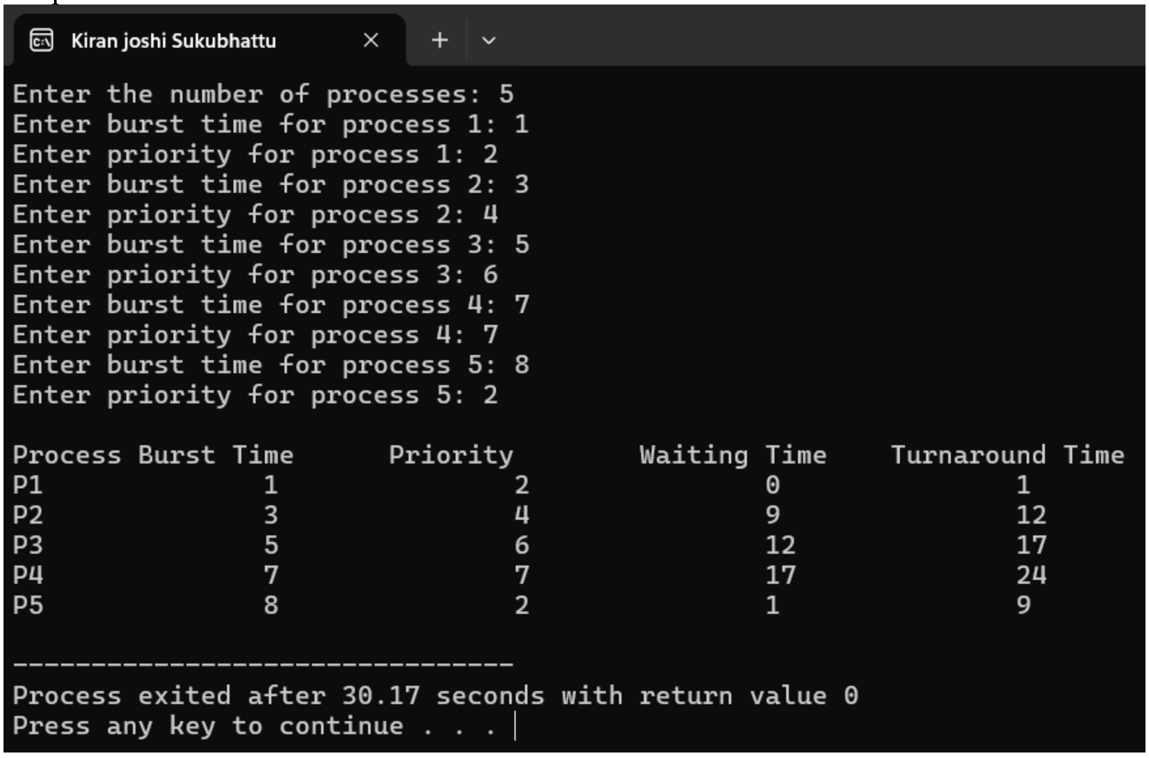
Input: List of processes with burst times and priorities.

```
1. Initialize the current time t = 0.
2. While there are uncompleted processes:
    1. Find the process P with the highest priority (lowest priority number) among
       those that have remaining burst time.
    2. Preempt the current process (if any) and allocate the CPU to P.
    3. Decrement P's remaining burst time by 1 unit.
    4. Increment t by 1.
    5. If P's remaining burst time is 0:
        ▪ Calculate P's turnaround time as t.
        ▪ Calculate P's waiting time as t - burst_time[P].
3. Output the waiting time and turnaround time for each process.
```

Time Complexity: O(n^2), where n is the number of processes.

Source Code:
```
#include <stdio.h>

struct Process {
    int pid;
    int burst_time;
    int priority;
    int remaining_time;
    int waiting_time;
    int turnaround_time;
};

void priorityScheduling(struct Process proc[], int n) {
    int t = 0, completed = 0, highest_priority = 0, min_priority;

    while (completed != n) {
        min_priority = 9999;
        int idx = -1;

        for (int i = 0; i < n; i++) {
            if (proc[i].remaining_time > 0 && proc[i].priority < min_priority) {
                min_priority = proc[i].priority;
                idx = i;
            }
        }
        if (idx != -1) {
            proc[idx].remaining_time--;
            t++;

            if (proc[idx].remaining_time == 0) {
                proc[idx].turnaround_time = t;
                proc[idx].waiting_time = t - proc[idx].burst_time;
                completed++;
            }
        }
    }
}


int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    struct Process proc[n];
    for (int i = 0; i < n; i++) {
        proc[i].pid = i + 1;
        printf("Enter burst time for process %d: ", proc[i].pid);
        scanf("%d", &proc[i].burst_time);
        printf("Enter priority for process %d: ", proc[i].pid);
        scanf("%d", &proc[i].priority);
        proc[i].remaining_time = proc[i].burst_time;
        proc[i].waiting_time = 0;
    }
    priorityScheduling(proc, n);
    printf("\nProcess\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\n", proc[i].pid, proc[i].burst_time, proc[i].priority,
proc[i].waiting_time, proc[i].turnaround_time);
    }

    return 0;
}
```

Output:



```
Enter the number of processes: 5
Enter burst time for process 1: 1
Enter priority for process 1: 2
Enter burst time for process 2: 3
Enter priority for process 2: 4
Enter burst time for process 3: 5
Enter priority for process 3: 6
Enter burst time for process 4: 7
Enter priority for process 4: 7
Enter burst time for process 5: 8
Enter priority for process 5: 2

Process Burst Time      Priority        Waiting Time    Turnaround Time
P1              1               2               0               1
P2              3               4               9               12
P3              5               6               12              17
P4              7               7               17              24
P5              8               2               1               9

--------------------------------
Process exited after 30.17 seconds with return value 0
Press any key to continue . . .
```

Experiment No.7

Aim/Title: Program to Simulate Bankers Algorithm for Dead Lock Avoidance.

Theory:

The Banker's Algorithm is a resource allocation and deadlock avoidance algorithm that tests for the safety of resource allocation to processes. It simulates resource allocation for a given number of processes and ensures that no deadlock will occur if the requested resources are allocated. The algorithm is named after a banker who allocates resources to customers (processes) in such a way that no customer leaves the bank in a state where they cannot fulfill their maximum needs. The banker only allocates resources if they can guarantee that even after allocation, at least one sequence of processes exists that can finish their execution without running into a deadlock.

Banker's Algorithm

Steps:

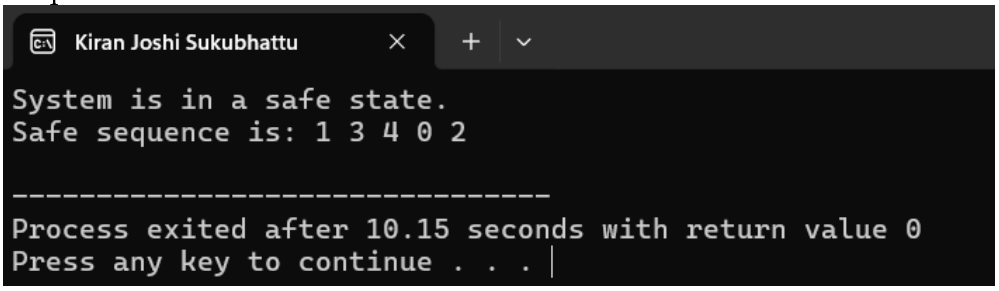
1. **Safety Algorithm**:
   - Initialize a work vector Work as a copy of Available.
   - Initialize a finish vector Finish of size n (number of processes) to false.
   - Find an index i such that:
     - Finish[i] == false
     - Need[i] <= Work
   - If such an index is found, update Work = Work + Allocation[i], and set Finish[i] = true.
   - Repeat until all processes are finished or no such index i exists.
   - If all processes can be finished, the system is in a safe state.
2. **Resource-Request Algorithm**:
   - If Request[i] <= Need[i], proceed; otherwise, raise an error condition.
   - If Request[i] <= Available, proceed; otherwise, the process must wait.
   - Pretend to allocate the requested resources by updating the state as:
     - Available = Available - Request[i]
     - Allocation[i] = Allocation[i] + Request[i]
     - Need[i] = Need[i] - Request[i]
   - Call the safety algorithm to check if the system is still in a safe state.
   - If safe, proceed with the allocation; if not, rollback the changes.

Source Code:

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

#define P 5 // Number of processes
#define R 3 // Number of resource types

void calculateNeed(int need[P][R], int max[P][R], int allocation[P][R]) {
    for (int i = 0; i < P; i++)
        for (int j = 0; j < R; j++)
            need[i][j] = max[i][j] - allocation[i][j];
}

bool isSafe(int processes[], int avail[], int max[][R], int allocation[][R]) {
    int need[P][R];
    calculateNeed(need, max, allocation);

    bool finish[P] = {false};
    int safeSeq[P];
    int work[R];

    for (int i = 0; i < R; i++)
        work[i] = avail[i];

    int count = 0;
    while (count < P) {
        bool found = false;
        for (int p = 0; p < P; p++) {
            if (finish[p] == false) {
                int j;
                for (j = 0; j < R; j++)
                    if (need[p][j] > work[j])
                        break;

                if (j == R) {
                    for (int k = 0 ; k < R ; k++)
                        work[k] += allocation[p][k];

                    safeSeq[count++] = p;
                    finish[p] = true;
                    found = true;
                }
            }
        }

        if (found == false) {
            printf("System is not in a safe state\n");
            return false;
        }
    }

    printf("System is in a safe state.\nSafe sequence is: ");
    for (int i = 0; i < P; i++)
        printf("%d ", safeSeq[i]);
    printf("\n");

    return true;
}

int main() {
    int processes[] = {0, 1, 2, 3, 4};

    int avail[] = {3, 3, 2}; // Available instances of resources

    int max[P][R] = { {7, 5, 3}, // Max demand of each process
```

```
                  {3, 2, 2},
                  {9, 0, 2},
                  {2, 2, 2},
                  {4, 3, 3} };

    int allocation[P][R] = { {0, 1, 0}, // Resources currently allocated to each process
                          {2, 0, 0},
                          {3, 0, 2},
                          {2, 1, 1},
                          {0, 0, 2} };

    isSafe(processes, avail, max, allocation);

    return 0;
}
```

Output:



```
System is in a safe state.
Safe sequence is: 1 3 4 0 2

--------------------------------
Process exited after 10.15 seconds with return value 0
Press any key to continue . . .
```

Experiment no. 8

Aim/Title: Program to Simulate Bankers Algorithm for Dead Lock Prevention.

Theory: Banker's Algorithm for Deadlock Prevention

The Banker's Algorithm is designed primarily for deadlock avoidance. However, it can also be adapted to help in deadlock prevention by ensuring that certain conditions that can lead to deadlock (such as circular wait) are avoided. In deadlock prevention, the system ensures that resources are allocated in such a way that it prevents the occurrence of a deadlock by careful resource allocation and ensuring that no unsafe state is entered.

To prevent deadlock, the Banker's Algorithm can be modified to enforce certain rules:

1. **Resource Allocation**: Allocate resources only if the system will remain in a safe state after the allocation.
2. **Safe Sequence**: Ensure that after allocation, at least one safe sequence exists for the remaining processes.

Banker's Algorithm for Deadlock Prevention

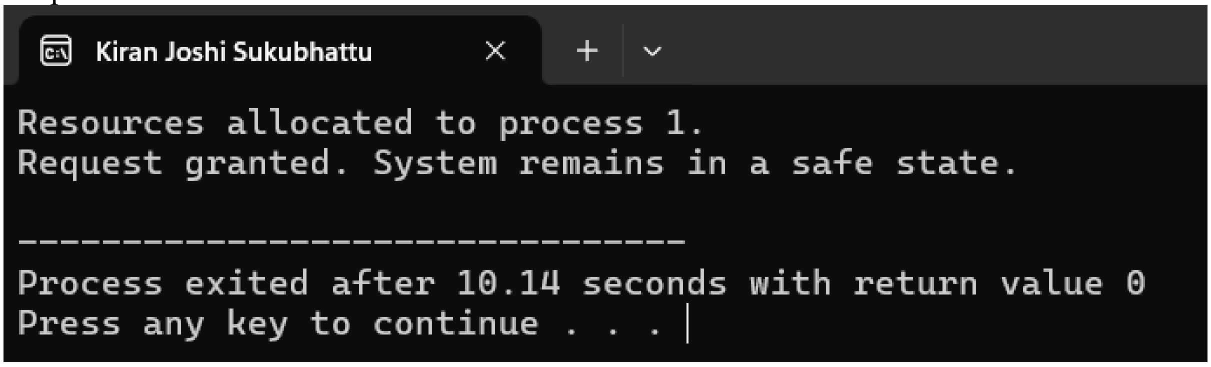
Steps:

1. **Calculate Need**: Compute the remaining resource needs of each process.
2. **Check Safety**: For each resource request, simulate the allocation, and check if the system remains in a safe state.
3. **Resource Request Handling**:
    - If the request can be granted without leading to an unsafe state, allocate the resources.
    - If granting the request would lead to an unsafe state, deny the request to prevent potential deadlock.

Source Code:

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

#define P 5 // Number of processes
#define R 3 // Number of resource types

void calculateNeed(int need[P][R], int max[P][R], int allocation[P][R]) {
    for (int i = 0; i < P; i++)
        for (int j = 0; j < R; j++)
            need[i][j] = max[i][j] - allocation[i][j];
}

bool isSafe(int processes[], int avail[], int max[][R], int allocation[][R]) {
    int need[P][R];
    calculateNeed(need, max, allocation);
    bool finish[P] = {false};
    int safeSeq[P];
    int work[R];

    for (int i = 0; i < R; i++)
        work[i] = avail[i];

    int count = 0;
    while (count < P) {
        bool found = false;
        for (int p = 0; p < P; p++) {
            if (finish[p] == false) {
                int j;
                for (j = 0; j < R; j++)
                    if (need[p][j] > work[j])
                        break;

                if (j == R) {
                    for (int k = 0 ; k < R ; k++)
                        work[k] += allocation[p][k];

                    safeSeq[count++] = p;
                    finish[p] = true;
                    found = true;
                }
            }
        }

        if (found == false) {
            return false;
        }
    }

    return true;
}

bool requestResources(int processID, int request[], int avail[], int allocation[][R], int max[][R]) {
    int need[P][R];
    calculateNeed(need, max, allocation);

    for (int i = 0; i < R; i++) {
        if (request[i] > need[processID][i]) {
            printf("Error: Process has exceeded its maximum claim.\n");
            return false;
        }

        if (request[i] > avail[i]) {
            printf("Process %d must wait, resources not available.\n", processID);
            return false;
        }
    }

    for (int i = 0; i < R; i++) {
        avail[i] -= request[i];
        allocation[processID][i] += request[i];
        need[processID][i] -= request[i];
    }

    int processes[P] = {0, 1, 2, 3, 4}; // Explicitly declare the processes array

    if (isSafe(processes, avail, max, allocation)) {
        printf("Resources allocated to process %d.\n", processID);
        return true;
    } else {
        for (int i = 0; i < R; i++) {
            avail[i] += request[i];
            allocation[processID][i] -= request[i];
            need[processID][i] += request[i];
        }
        printf("Cannot allocate resources to process %d. System would be in an unsafe state.\n",
processID);
        return false;
    }
}

int main() {
    int avail[] = {3, 3, 2}; // Available instances of resources

    int max[P][R] = { {7, 5, 3}, // Max demand of each process
                      {3, 2, 2},
                      {9, 0, 2},
                      {2, 2, 2},
                      {4, 3, 3} };

    int allocation[P][R] = { {0, 1, 0}, // Resources currently allocated to each process
                             {2, 0, 0},
                             {3, 0, 2},
                             {2, 1, 1},
                             {0, 0, 2} };

    int processID = 1; // Example: Process 1 requesting resources
    int request[R] = {1, 0, 2}; // Example: Process 1 requesting these resources

    if (requestResources(processID, request, avail, allocation, max)) {
        printf("Request granted. System remains in a safe state.\n");
    } else {
        printf("Request denied to prevent deadlock.\n");
    }
    return 0;
}
```

Output:



```
Resources allocated to process 1.
Request granted. System remains in a safe state.

--------------------------------
Process exited after 10.14 seconds with return value 0
Press any key to continue . . .
```

Experiment no. 9

Aim/Title: Program to Simulate the MVT and MFT memory management techniques.

Theory: MVT and MFT Memory Management Techniques

Memory management is a critical aspect of operating systems, determining how memory is allocated to processes. Two popular memory management techniques are **MVT (Multiprogramming with a Variable number of Tasks)** and **MFT (Multiprogramming with a Fixed number of Tasks)**.

1. MVT (Multiprogramming with a Variable number of Tasks)

- **Concept**: In MVT, the memory is divided into variable-sized partitions. Processes are allocated exactly as much memory as they need, which leads to better memory utilization but can cause external fragmentation.
- **Advantages**:
  - Flexible memory allocation.
  - Minimizes internal fragmentation.
- **Disadvantages**:
  - External fragmentation can occur.
  - More complex memory management is needed.

2. MFT (Multiprogramming with a Fixed number of Tasks)

- **Concept**: In MFT, the memory is divided into fixed-sized partitions. Each process is allocated a partition, and multiple processes can be loaded into memory simultaneously if the total size does not exceed the partition size.
- **Advantages**:
  - Simple to implement.
  - No external fragmentation.
- **Disadvantages**:
  - Leads to internal fragmentation if processes do not fully utilize their partition.
  - Less flexible compared to MVT.

Algorithm

MVT Algorithm:

1. **Initialize memory** with a total size.
2. **Input process sizes** and dynamically allocate memory to each process.
3. **Check if memory is available** for the new process:
   - If available, allocate memory.
   - If not available, wait until sufficient memory is freed.
4. **Deallocate memory** when processes are completed.

MFT Algorithm:

1. **Initialize memory** by dividing it into fixed-size partitions.
2. **Input process sizes** and allocate a suitable partition to each process.
3. **Check if partition size is sufficient** for the process:
   - If a suitable partition is available, allocate it.
   - If no partition is available, the process must wait.
4. **Deallocate the partition** when processes are completed, making it available for new processes.

Source Code:

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

#define MAX_PROCESSES 10
```

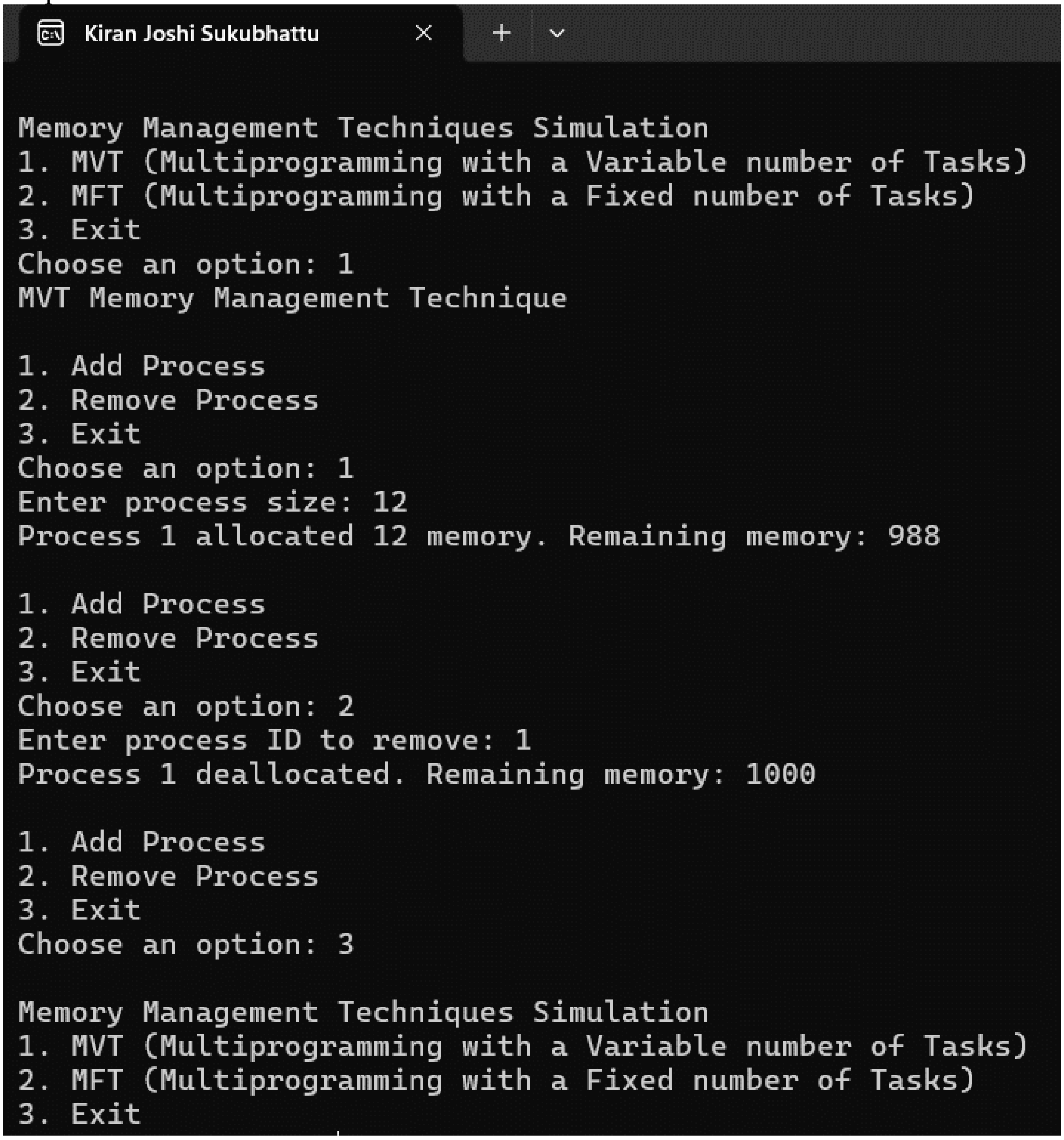
```
#define MEMORY_SIZE 1000  // Total memory size for MVT

void mvt() {
   int memory = MEMORY_SIZE;
   int processSize[MAX_PROCESSES];
   int allocated[MAX_PROCESSES] = {0};
   int processCount = 0;

   printf("MVT Memory Management Technique\n");
   while (1) {
      int choice;
      printf("\n1. Add Process\n2. Remove Process\n3. Exit\nChoose an option: ");
      scanf("%d", &choice);

      if (choice == 1) {
         if (processCount >= MAX_PROCESSES) {
            printf("Cannot add more processes. Maximum process limit reached.\n");
            continue;
         }

         printf("Enter process size: ");
         scanf("%d", &processSize[processCount]);

         if (processSize[processCount] <= memory) {
            memory -= processSize[processCount];
            allocated[processCount] = 1;
            printf("Process %d allocated %d memory. Remaining memory: %d\n", processCount
+ 1, processSize[processCount], memory);
            processCount++;
         } else {
            printf("Not enough memory to allocate process %d. Remaining memory: %d\n",
processCount + 1, memory);
         }
      } else if (choice == 2) {
         int processID;
         printf("Enter process ID to remove: ");
         scanf("%d", &processID);

         if (processID > 0 && processID <= processCount && allocated[processID - 1]) {
            memory += processSize[processID - 1];
            allocated[processID - 1] = 0;
            printf("Process %d deallocated. Remaining memory: %d\n", processID, memory);
         } else {
            printf("Invalid process ID.\n");
         }
      } else if (choice == 3) {
         break;
      } else {
         printf("Invalid choice.\n");
      }
```

```
    }
}

void mft() {
    int partitionSize;
    int processSize[MAX_PROCESSES];
    int allocated[MAX_PROCESSES] = {0};
    int partitions[MAX_PROCESSES];
    int processCount = 0;
    int partitionCount;

    printf("\nMFT Memory Management Technique\n");

    printf("Enter partition size: ");
    scanf("%d", &partitionSize);

    printf("Enter number of partitions: ");
    scanf("%d", &partitionCount);

    for (int i = 0; i < partitionCount; i++) {
        partitions[i] = partitionSize;
    }

    while (1) {
        int choice;
        printf("\n1. Add Process\n2. Remove Process\n3. Exit\nChoose an option: ");
        scanf("%d", &choice);

        if (choice == 1) {
            if (processCount >= MAX_PROCESSES) {
                printf("Cannot add more processes. Maximum process limit reached.\n");
                continue;
            }

            printf("Enter process size: ");
            scanf("%d", &processSize[processCount]);

            bool allocatedFlag = false;
            for (int i = 0; i < partitionCount; i++) {
                if (partitions[i] >= processSize[processCount] && allocated[i] == 0) {
                    allocated[i] = 1;
                    partitions[i] -= processSize[processCount];
                    printf("Process %d allocated %d memory in partition %d.\n", processCount + 1,
processSize[processCount], i + 1);
                    allocatedFlag = true;
                    break;
                }
            }

            if (!allocatedFlag) {
```

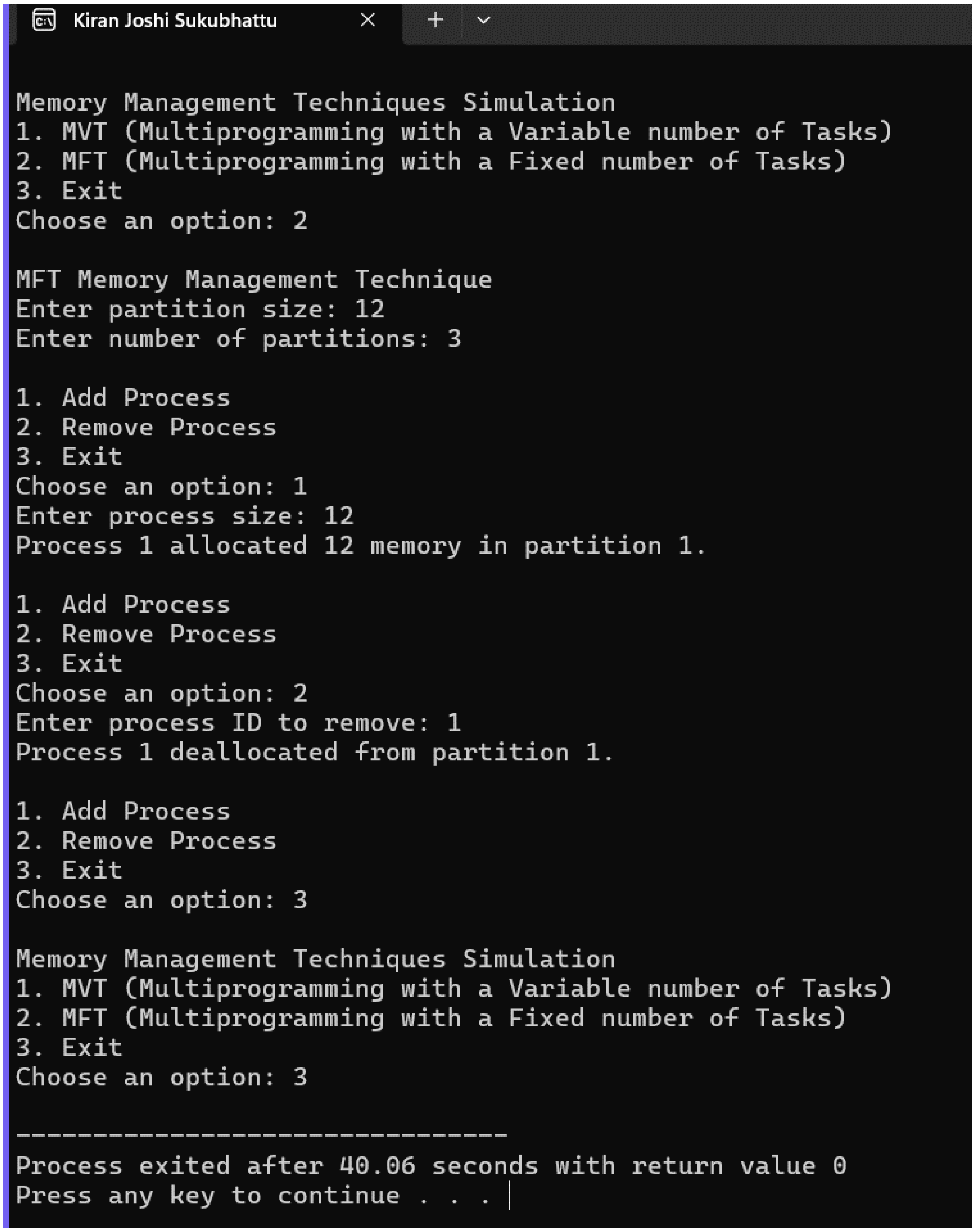
```
            printf("No suitable partition available for process %d.\n", processCount + 1);
        } else {
            processCount++;
        }
    } else if (choice == 2) {
        int processID;
        printf("Enter process ID to remove: ");
        scanf("%d", &processID);

        if (processID > 0 && processID <= processCount && allocated[processID - 1]) {
            for (int i = 0; i < partitionCount; i++) {
                if (allocated[i] == 1 && processSize[processID - 1] == partitionSize -
partitions[i]) {
                    partitions[i] = partitionSize;
                    allocated[i] = 0;
                    printf("Process %d deallocated from partition %d.\n", processID, i + 1);
                    break;
                }
            }
        } else {
            printf("Invalid process ID.\n");
        }
    } else if (choice == 3) {
        break;
    } else {
        printf("Invalid choice.\n");
    }
  }
}

int main() {
  int choice;

  while (1) {
    printf("\nMemory Management Techniques Simulation\n");
    printf("1. MVT (Multiprogramming with a Variable number of Tasks)\n");
    printf("2. MFT (Multiprogramming with a Fixed number of Tasks)\n");
    printf("3. Exit\nChoose an option: ");
    scanf("%d", &choice);

    if (choice == 1) {
        mvt();
    } else if (choice == 2) {
        mft();
    } else if (choice == 3) {
        break;
    } else {
        printf("Invalid choice.\n");
    }
  }
```

```
    return 0;
}
```

Output:



```
Memory Management Techniques Simulation
1. MVT (Multiprogramming with a Variable number of Tasks)
2. MFT (Multiprogramming with a Fixed number of Tasks)
3. Exit
Choose an option: 1
MVT Memory Management Technique

1. Add Process
2. Remove Process
3. Exit
Choose an option: 1
Enter process size: 12
Process 1 allocated 12 memory. Remaining memory: 988

1. Add Process
2. Remove Process
3. Exit
Choose an option: 2
Enter process ID to remove: 1
Process 1 deallocated. Remaining memory: 1000

1. Add Process
2. Remove Process
3. Exit
Choose an option: 3

Memory Management Techniques Simulation
1. MVT (Multiprogramming with a Variable number of Tasks)
2. MFT (Multiprogramming with a Fixed number of Tasks)
3. Exit
```



```
Memory Management Techniques Simulation
1. MVT (Multiprogramming with a Variable number of Tasks)
2. MFT (Multiprogramming with a Fixed number of Tasks)
3. Exit
Choose an option: 2

MFT Memory Management Technique
Enter partition size: 12
Enter number of partitions: 3

1. Add Process
2. Remove Process
3. Exit
Choose an option: 1
Enter process size: 12
Process 1 allocated 12 memory in partition 1.

1. Add Process
2. Remove Process
3. Exit
Choose an option: 2
Enter process ID to remove: 1
Process 1 deallocated from partition 1.

1. Add Process
2. Remove Process
3. Exit
Choose an option: 3

Memory Management Techniques Simulation
1. MVT (Multiprogramming with a Variable number of Tasks)
2. MFT (Multiprogramming with a Fixed number of Tasks)
3. Exit
Choose an option: 3

--------------------------------
Process exited after 40.06 seconds with return value 0
Press any key to continue . . .
```

Experiment No. 10

Aim/Title: Simulate paging technique of memory management.

Theory: Paging in Memory Management

Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory, thus minimizing fragmentation issues. It divides the process's memory into fixed-size blocks called **pages** and divides the physical memory into fixed-size blocks called **frames**. Each page of a process is loaded into a frame of the physical memory.

- **Page**: A fixed-size block of logical memory.
- **Frame**: A fixed-size block of physical memory.

Key Concepts:

- **Page Table**: A data structure used to map logical addresses to physical addresses.
- **Page Number**: The part of the logical address that identifies which page in the logical address space the address belongs to.
- **Frame Number**: The part of the physical address that identifies which frame in the physical memory the page resides in.
- **Page Offset**: The displacement within the page/frame.

Paging Algorithm

Steps:

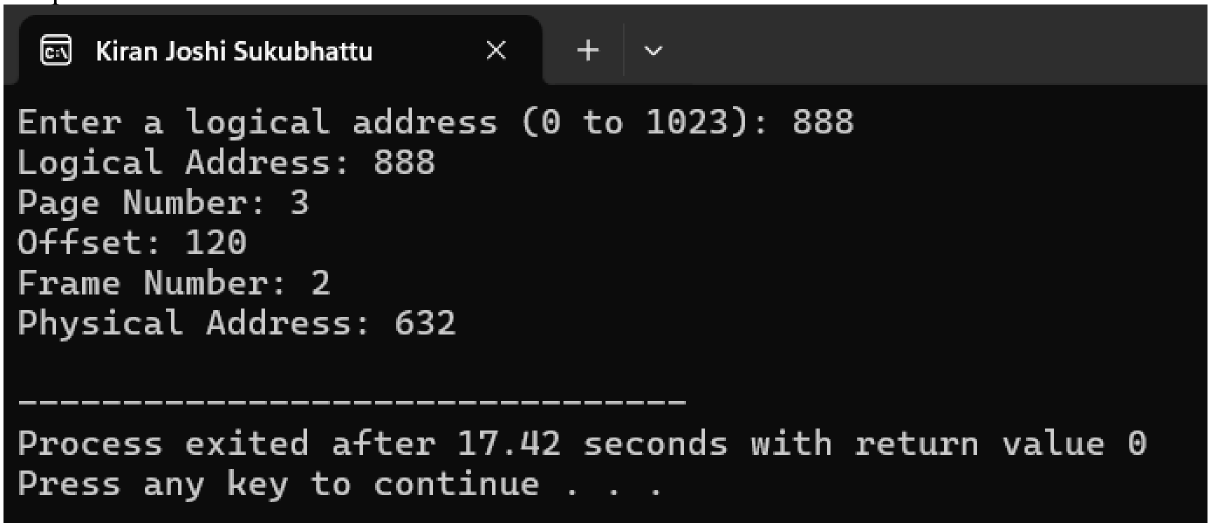
1. **Divide the Process**: Split the process's logical memory into pages.
2. **Divide Physical Memory**: Split the physical memory into frames.
3. **Map Pages to Frames**: Use the page table to map each page to a specific frame.
4. **Address Translation**: When a process generates a logical address, the operating system divides this address into a page number and a page offset.
5. **Look Up the Page Table**: The page number is used to find the corresponding frame number in the page table.
6. **Calculate Physical Address**: The physical address is then computed using the frame number and page offset.
7.

Source Code:

```
#include <stdio.h>

#define FRAME_SIZE 256  // Frame size
#define MEMORY_SIZE 1024 // Total memory size
#define PAGE_SIZE 256 // Page size

int main() {
   int logical_address, page_number, offset, frame_number, physical_address;
   int page_table[4]; // Assume we have 4 pages, hence 4 entries in the page table
   int frames[MEMORY_SIZE / FRAME_SIZE]; // Memory divided into frames

   // Initialize the page table with some frame numbers
   page_table[0] = 3; // Page 0 is stored in Frame 3
   page_table[1] = 1; // Page 1 is stored in Frame 1
   page_table[2] = 4; // Page 2 is stored in Frame 4
   page_table[3] = 2; // Page 3 is stored in Frame 2

   printf("Enter a logical address (0 to %d): ", MEMORY_SIZE - 1);
   scanf("%d", &logical_address);
```

```
    // Calculate the page number and offset
    page_number = logical_address / PAGE_SIZE;
    offset = logical_address % PAGE_SIZE;

    // Get the frame number from the page table
    frame_number = page_table[page_number];

    // Calculate the physical address
    physical_address = frame_number * FRAME_SIZE + offset;

    // Display the results
    printf("Logical Address: %d\n", logical_address);
    printf("Page Number: %d\n", page_number);
    printf("Offset: %d\n", offset);
    printf("Frame Number: %d\n", frame_number);
    printf("Physical Address: %d\n", physical_address);

    return 0;
}
```

Output:



```
Enter a logical address (0 to 1023): 888
Logical Address: 888
Page Number: 3
Offset: 120
Frame Number: 2
Physical Address: 632

--------------------------------
Process exited after 17.42 seconds with return value 0
Press any key to continue . . .
```

EXPERIMENT NO. 11

Aim/Theory: Write a C program to simulate the FIRST-FIT contiguous memory allocation technique.

Theory

The FIRST-FIT memory allocation technique is used to manage memory by allocating the first available block that is large enough to satisfy the request. It is a simple approach that searches from the beginning of the memory list and allocates the first block that is sufficiently large.

Algorithm: FIRST-FIT Memory Allocation

1 Initialization:

- Define a list of memory blocks, where each block has a size and a status (allocated or free).
- Define a list of processes, where each process has a size and a status (allocated or not allocated).

2 Input:

- Read the number of memory blocks and their sizes.
- Read the number of processes and their sizes.

3 Allocate Memory:

- For each process in the list:

A) Initialize a flag allocated to false.

B) Iterate over each memory block:

- If the current memory block is free and its size is greater than or equal to the size of the process:
- Allocate the memory block to the process.
- Set the allocated flag to true.
- Mark the memory block as allocated.
- Exit the loop and move to the next process.

C) If the allocated flag is still false after checking all blocks, mark the process as not allocated.

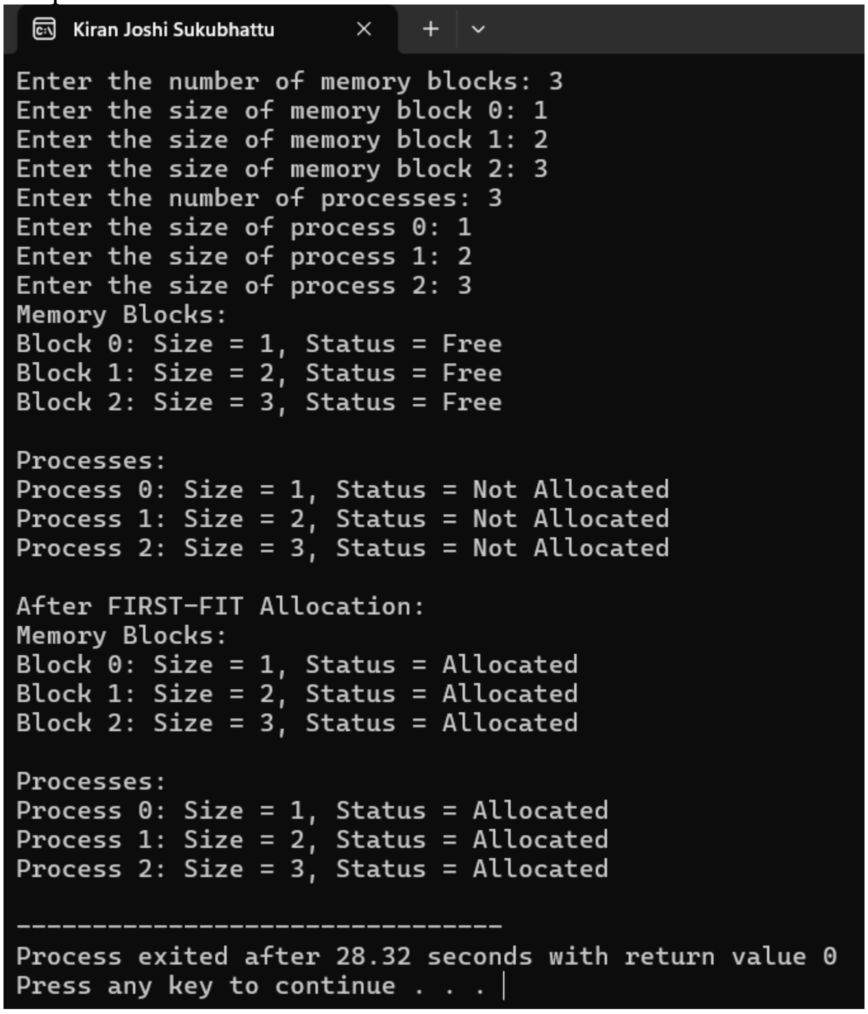
4 Output:

- Display the status of memory blocks (allocated or free).
- Display the status of processes (allocated or not allocated).

Source Code:

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

typedef struct {
   int size;
   int allocated;
} MemoryBlock;

typedef struct {
    int size;
    int allocated;
} Process;

void displayMemory(MemoryBlock blocks[], int numBlocks) {
    printf("Memory Blocks:\n");
    for (int i = 0; i < numBlocks; i++) {
        if (blocks[i].allocated) {
            printf("Block %d: Size = %d, Status = Allocated\n", i, blocks[i].size);
        } else {
            printf("Block %d: Size = %d, Status = Free\n", i, blocks[i].size);
        }
    }
}
void displayProcesses(Process processes[], int numProcesses) {
    printf("\nProcesses:\n");
    for (int i = 0; i < numProcesses; i++) {
        if (processes[i].allocated) {
            printf("Process %d: Size = %d, Status = Allocated\n", i, processes[i].size);
        } else {
            printf("Process %d: Size = %d, Status = Not Allocated\n", i, processes[i].size);
        }
    }
}
void firstFit(MemoryBlock blocks[], int numBlocks, Process processes[], int numProcesses) {
    for (int i = 0; i < numProcesses; i++) {
        int allocated = 0;
        for (int j = 0; j < numBlocks; j++) {
            if (!blocks[j].allocated && blocks[j].size >= processes[i].size) {
                blocks[j].allocated = 1;
                processes[i].allocated = 1;
                allocated = 1;
                break;
            }
        }

        if (!allocated) {
            processes[i].allocated = 0;
        }
    }
}

int main() {
    int numBlocks, numProcesses;
    printf("Enter the number of memory blocks: ");
    scanf("%d", &numBlocks);
    MemoryBlock blocks[MAX_BLOCKS];
    for (int i = 0; i < numBlocks; i++) {
        printf("Enter the size of memory block %d: ", i);
        scanf("%d", &blocks[i]);
        blocks[i].allocated = 0; // Mark all blocks as free initially
    }
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);
    Process processes[MAX_PROCESSES];
    for (int i = 0; i < numProcesses; i++) {
        printf("Enter the size of process %d: ", i);
        scanf("%d", &processes[i]);
        processes[i].allocated = 0; // Mark all processes as not allocated initially
    } // Display initial state
    displayMemory(blocks, numBlocks);
    displayProcesses(processes, numProcesses);
    // Apply FIRST-FIT allocation
    firstFit(blocks, numBlocks, processes, numProcesses);
    // Display final state
    printf("\nAfter FIRST-FIT Allocation:\n");
    displayMemory(blocks, numBlocks);
    displayProcesses(processes, numProcesses);
    return 0;
}
```

Output:



```
Enter the number of memory blocks: 3
Enter the size of memory block 0: 1
Enter the size of memory block 1: 2
Enter the size of memory block 2: 3
Enter the number of processes: 3
Enter the size of process 0: 1
Enter the size of process 1: 2
Enter the size of process 2: 3
Memory Blocks:
Block 0: Size = 1, Status = Free
Block 1: Size = 2, Status = Free
Block 2: Size = 3, Status = Free

Processes:
Process 0: Size = 1, Status = Not Allocated
Process 1: Size = 2, Status = Not Allocated
Process 2: Size = 3, Status = Not Allocated

After FIRST-FIT Allocation:
Memory Blocks:
Block 0: Size = 1, Status = Allocated
Block 1: Size = 2, Status = Allocated
Block 2: Size = 3, Status = Allocated

Processes:
Process 0: Size = 1, Status = Allocated
Process 1: Size = 2, Status = Allocated
Process 2: Size = 3, Status = Allocated

--------------------------------
Process exited after 28.32 seconds with return value 0
Press any key to continue . . .
```

EXPERIMENT NO. 12

AIM / TITLE: Write a C program to simulate the BEST FIT contiguous memory allocation technique.

Theory

The BEST-FIT contiguous memory allocation technique allocates the smallest available memory block that is sufficient to accommodate a process. This strategy aims to reduce memory wastage by choosing the block that leaves the smallest leftover space after allocation. The main idea is to minimize fragmentation by choosing the best fit.

Algorithm: BEST-FIT Memory Allocation

1 **Initialization:**

- Define a list of memory blocks, where each block has a size and a status (allocated or free).
- Define a list of processes, where each process has a size and a status (allocated or not allocated).

2 **Input:**

- Read the number of memory blocks and their sizes.
- Read the number of processes and their sizes.

3 **Allocate Memory:**

- For each process in the list:

A) Initialize a variable to keep track of the best fit (bestFitIndex), and set it to -1.

B) Iterate over each memory block:

- If the current memory block is free and its size is greater than or equal to the size of the process:
- If bestFitIndex is -1 or the current block leaves less leftover space than the previously found block:
- Update bestFitIndex to the index of the current block.

C) After checking all blocks, if bestFitIndex is not -1:

- Allocate the memory block at bestFitIndex to the process.
- Mark the memory block as allocated.
- Mark the process as allocated.

D) If bestFitIndex is -1 after checking all blocks, mark the process as not allocated.

4 **Output:**

- Display the status of memory blocks (allocated or free).
- Display the status of processes (allocated or not allocated).

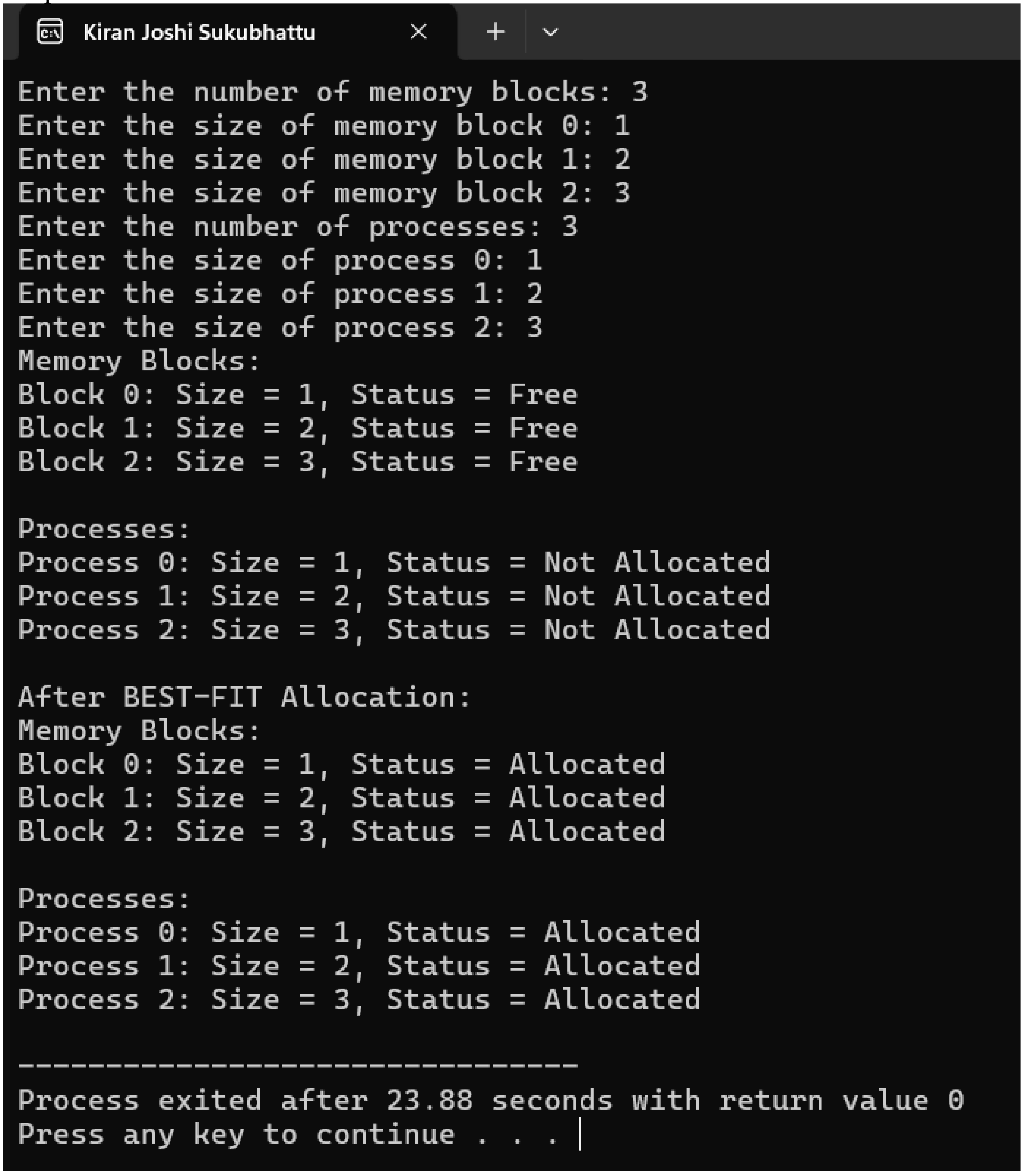
Source Code:

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

typedef struct {
   int size;
   int allocated;
} MemoryBlock;
```

```
typedef struct {
    int size;
    int allocated;
} Process;

void displayMemory(MemoryBlock blocks[], int numBlocks) {
    printf("Memory Blocks:\n");
    for (int i = 0; i < numBlocks; i++) {
        if (blocks[i].allocated) {
            printf("Block %d: Size = %d, Status = Allocated\n", i, blocks[i].size);
        } else {
            printf("Block %d: Size = %d, Status = Free\n", i, blocks[i].size);
        }
    }
}

void displayProcesses(Process processes[], int numProcesses) {
    printf("\nProcesses:\n");
    for (int i = 0; i < numProcesses; i++) {
        if (processes[i].allocated) {
            printf("Process %d: Size = %d, Status = Allocated\n", i, processes[i].size);
        } else {
            printf("Process %d: Size = %d, Status = Not Allocated\n", i, processes[i].size);
        }
    }
}

void bestFit(MemoryBlock blocks[], int numBlocks, Process processes[], int numProcesses) {
    for (int i = 0; i < numProcesses; i++) {
        int bestFitIndex = -1;
        int minSizeLeftover = __INT_MAX__; // Use a large number to ensure proper
comparison
        for (int j = 0; j < numBlocks; j++) {
            if (!blocks[j].allocated && blocks[j].size >= processes[i].size) {
                int sizeLeftover = blocks[j].size - processes[i].size;
                if (sizeLeftover < minSizeLeftover) {
                    minSizeLeftover = sizeLeftover;
                    bestFitIndex = j;
                }}}
        if (bestFitIndex != -1) {
            blocks[bestFitIndex].allocated = 1;
            processes[i].allocated = 1;
        } else {
            processes[i].allocated = 0;
        }
    }
}
```

```
int main() {
   int numBlocks, numProcesses;
   printf("Enter the number of memory blocks: ");
   scanf("%d", &numBlocks);
   MemoryBlock blocks[MAX_BLOCKS];
   for (int i = 0; i < numBlocks; i++) {
      printf("Enter the size of memory block %d: ", i);
      scanf("%d", &blocks[i].size);
      blocks[i].allocated = 0; // Mark all blocks as free initially
   }
   printf("Enter the number of processes: ");
   scanf("%d", &numProcesses);

   Process processes[MAX_PROCESSES];
   for (int i = 0; i < numProcesses; i++) {
      printf("Enter the size of process %d: ", i);
      scanf("%d", &processes[i].size);
      processes[i].allocated = 0; // Mark all processes as not allocated initially
   }
   displayMemory(blocks, numBlocks);
   displayProcesses(processes, numProcesses);
   bestFit(blocks, numBlocks, processes, numProcesses);
   printf("\nAfter BEST-FIT Allocation:\n");
   displayMemory(blocks, numBlocks);
   displayProcesses(processes, numProcesses);
   return 0;
}
```

Output:



```
Enter the number of memory blocks: 3
Enter the size of memory block 0: 1
Enter the size of memory block 1: 2
Enter the size of memory block 2: 3
Enter the number of processes: 3
Enter the size of process 0: 1
Enter the size of process 1: 2
Enter the size of process 2: 3
Memory Blocks:
Block 0: Size = 1, Status = Free
Block 1: Size = 2, Status = Free
Block 2: Size = 3, Status = Free

Processes:
Process 0: Size = 1, Status = Not Allocated
Process 1: Size = 2, Status = Not Allocated
Process 2: Size = 3, Status = Not Allocated

After BEST-FIT Allocation:
Memory Blocks:
Block 0: Size = 1, Status = Allocated
Block 1: Size = 2, Status = Allocated
Block 2: Size = 3, Status = Allocated

Processes:
Process 0: Size = 1, Status = Allocated
Process 1: Size = 2, Status = Allocated
Process 2: Size = 3, Status = Allocated

--------------------------------
Process exited after 23.88 seconds with return value 0
Press any key to continue . . .
```

Experiment No. 13:

AIM / TITLE: Write a C program to simulate the WORST-FIT contiguous memory allocation technique.

Theory

The WORST-FIT contiguous memory allocation technique allocates the largest available memory block to a process. This approach aims to reduce fragmentation by leaving the largest possible chunks of memory available. The idea is to use the block that has the most leftover space after allocation, hoping that it will be easier to allocate subsequent processes in the remaining larger portions of memory.

Algorithm: WORST-FIT Memory Allocation

1 **Initialization:**
- Define a list of memory blocks, where each block has a size and a status (allocated or free).
- Define a list of processes, where each process has a size and a status (allocated or not allocated).

2 **Input:**
- Read the number of memory blocks and their sizes.
- Read the number of processes and their sizes.

3 **Allocate Memory:**
- For each process in the list:

A) Initialize a variable to keep track of the worst fit (worstFitIndex), and set it to -1.

B)Iterate over each memory block:
- If the current memory block is free and its size is greater than or equal to the size of the process:
- If worstFitIndex is -1 or the current block leaves more leftover space than the previously found block:
- Update worstFitIndex to the index of the current block.

C) After checking all blocks, if worstFitIndex is not -1:
- Allocate the memory block at worstFitIndex to the process.
- Mark the memory block as allocated.
- Mark the process as allocated.

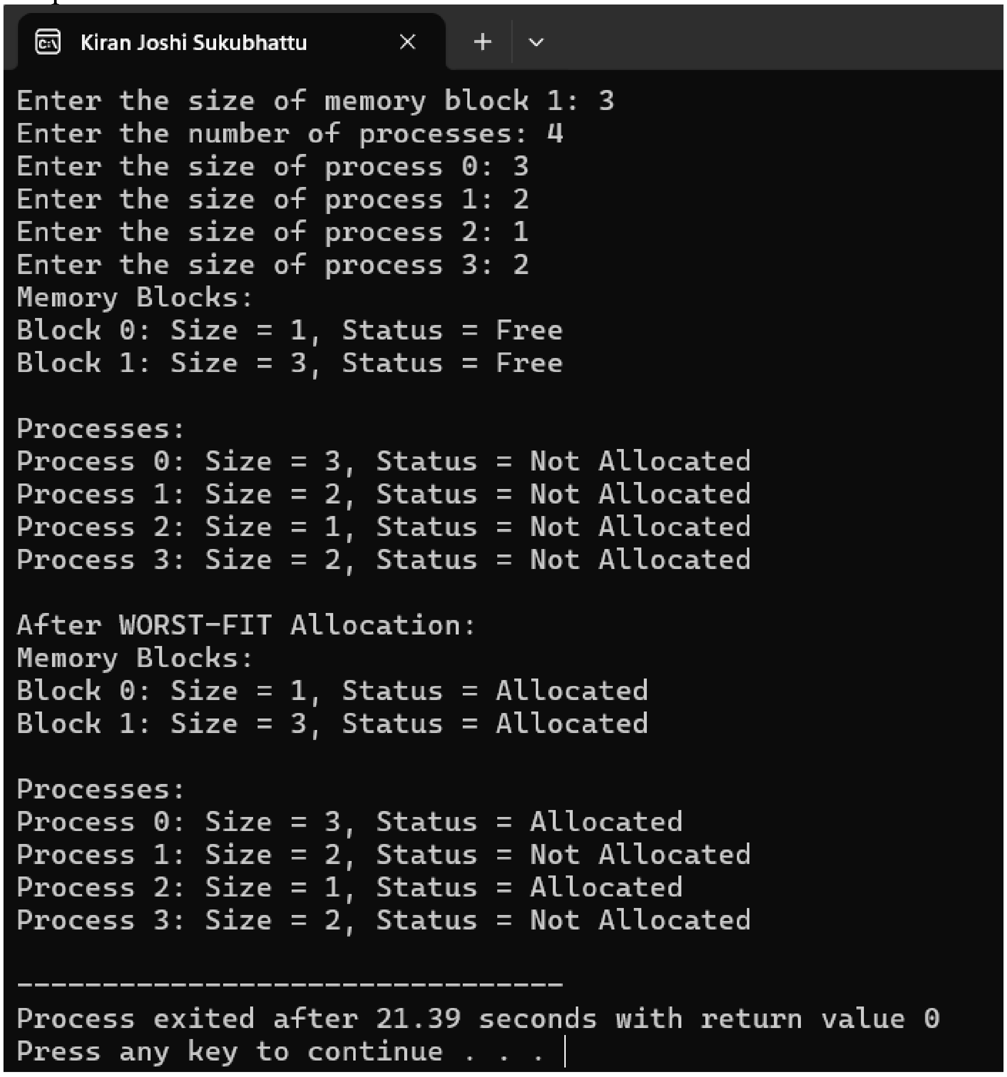
D) If worstFitIndex is -1 after checking all blocks, mark the process as not allocated.

Output:
- Display the status of memory blocks (allocated or free).
- Display the status of processes (allocated or not allocated).

Source Code:
```
#include <stdio.h>
#include <stdlib.h>
#define MAX_BLOCKS 100
#define MAX_PROCESSES 100

typedef struct {
   int size;
   int allocated;
} MemoryBlock;
```

```
typedef struct {
    int size;
    int allocated;
} Process;

void displayMemory(MemoryBlock blocks[], int numBlocks) {
    printf("Memory Blocks:\n");
    for (int i = 0; i < numBlocks; i++) {
        if (blocks[i].allocated) {
            printf("Block %d: Size = %d, Status = Allocated\n", i, blocks[i].size);
        } else {
            printf("Block %d: Size = %d, Status = Free\n", i, blocks[i].size);
        }
    }
}
void displayProcesses(Process processes[], int numProcesses) {
    printf("\nProcesses:\n");
    for (int i = 0; i < numProcesses; i++) {
        if (processes[i].allocated) {
            printf("Process %d: Size = %d, Status = Allocated\n", i, processes[i].size);
        } else {
            printf("Process %d: Size = %d, Status = Not Allocated\n", i, processes[i].size);
        }}}
void worstFit(MemoryBlock blocks[], int numBlocks, Process processes[], int numProcesses)
{
    for (int i = 0; i < numProcesses; i++) {
        int worstFitIndex = -1;
        int maxSizeLeftover = -1; // Use a small number to ensure proper comparison
        for (int j = 0; j < numBlocks; j++) {
            if (!blocks[j].allocated && blocks[j].size >= processes[i].size) {
                int sizeLeftover = blocks[j].size - processes[i].size;
                if (sizeLeftover > maxSizeLeftover) {
                    maxSizeLeftover = sizeLeftover;
                    worstFitIndex = j;
                }}}
        if (worstFitIndex != -1) {
            blocks[worstFitIndex].allocated = 1;
            processes[i].allocated = 1;
        } else {
            processes[i].allocated = 0;
        }}}
int main() {
    int numBlocks, numProcesses;
    printf("Enter the number of memory blocks: ");
    scanf("%d", &numBlocks);
    MemoryBlock blocks[MAX_BLOCKS];
    for (int i = 0; i < numBlocks; i++) {
        printf("Enter the size of memory block %d: ", i);
        scanf("%d", &blocks[i].size);
        blocks[i].allocated = 0; // Mark all blocks as free initially
    }
    printf("Enter the number of processes: ");
    scanf("%d", &numProcesses);
    Process processes[MAX_PROCESSES];
    for (int i = 0; i < numProcesses; i++) {
        printf("Enter the size of process %d: ", i);
        scanf("%d", &processes[i].size);
        processes[i].allocated = 0; // Mark all processes as not allocated initially
    }
    displayMemory(blocks, numBlocks);
    displayProcesses(processes, numProcesses);
    worstFit(blocks, numBlocks, processes, numProcesses);
    printf("\nAfter WORST-FIT Allocation:\n");
    displayMemory(blocks, numBlocks);
    displayProcesses(processes, numProcesses);
    return 0;
}
```

Output:



```
Enter the size of memory block 1: 3
Enter the number of processes: 4
Enter the size of process 0: 3
Enter the size of process 1: 2
Enter the size of process 2: 1
Enter the size of process 3: 2
Memory Blocks:
Block 0: Size = 1, Status = Free
Block 1: Size = 3, Status = Free

Processes:
Process 0: Size = 3, Status = Not Allocated
Process 1: Size = 2, Status = Not Allocated
Process 2: Size = 1, Status = Not Allocated
Process 3: Size = 2, Status = Not Allocated

After WORST-FIT Allocation:
Memory Blocks:
Block 0: Size = 1, Status = Allocated
Block 1: Size = 3, Status = Allocated

Processes:
Process 0: Size = 3, Status = Allocated
Process 1: Size = 2, Status = Not Allocated
Process 2: Size = 1, Status = Allocated
Process 3: Size = 2, Status = Not Allocated

--------------------------------
Process exited after 21.39 seconds with return value 0
Press any key to continue . . .
```

EXPERIMENT NO. 14

AIM / TITLE: To simulate the FIFO page replacement algorithm.

Theory

The FIFO (First-In, First-Out) page replacement algorithm is a simple and intuitive method used in operating systems to manage the contents of a page table. When a page fault occurs (i.e., a page that is needed is not currently in memory), FIFO replaces the oldest page in memory with the new page. The oldest page is the one that has been in memory the longest.

Algorithm: FIFO Page Replacement

1. **Initialization:**
   - Define a page frame list (FIFO queue) that stores the current pages in memory.
   - Define the page reference string (a sequence of page requests).

2. **Input:**
   - Read the number of page frames (capacity of the memory).
   - Read the page reference string (sequence of pages to be accessed).

3. **Page Replacement:**
   - For each page in the reference string:

   A) Check if the page is already in the page frame list (memory).
   - If the page is present, no page fault occurs.
   - If the page is not present (page fault), proceed to step 2.

   B) If there is space available in the page frame list, add the new page to the list.

   C) If there is no space available, remove the oldest page from the list (the page that has been in memory the longest) and add the new page to the list.

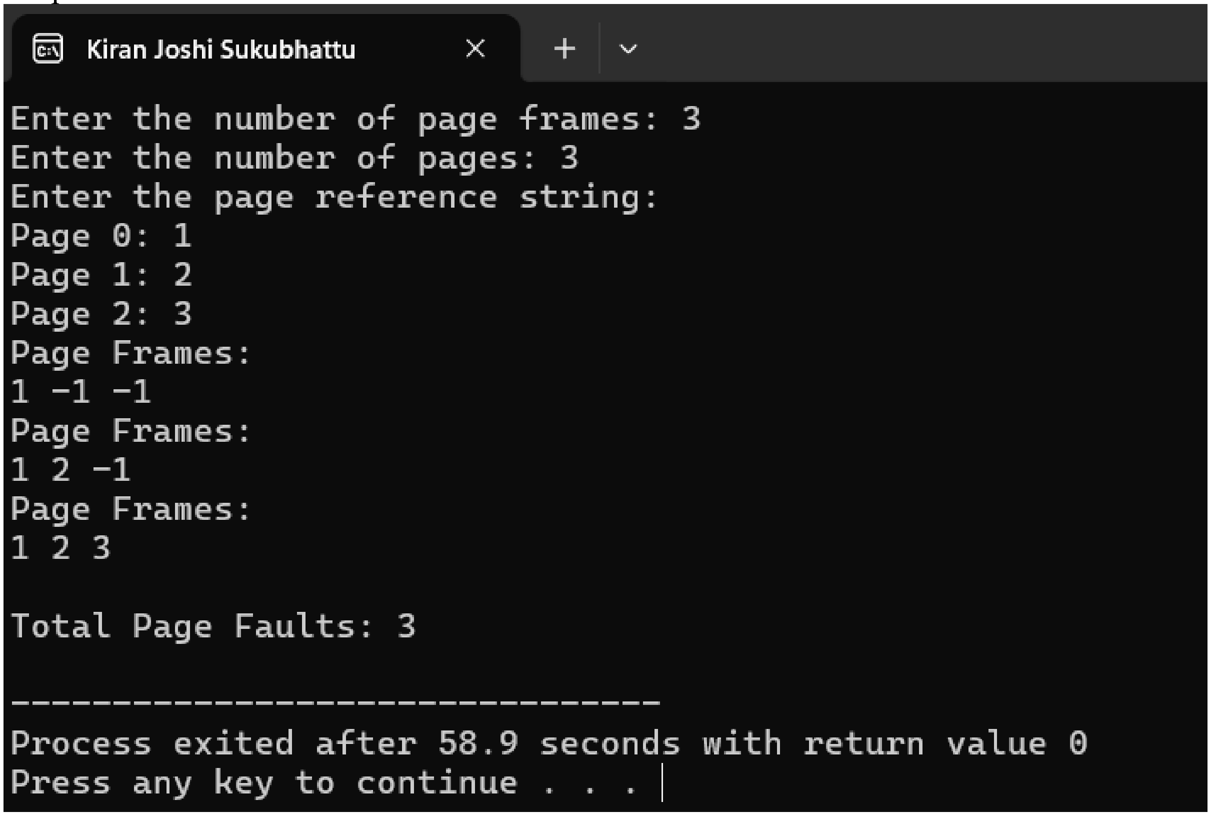
4. **Output:**
   - Display the number of page faults.
   - Display the final contents of the page frame list.

Source Code

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_FRAMES 100
#define MAX_PAGES 100

void displayFrames(int frames[], int numFrames) {
   printf("Page Frames:\n");
   for (int i = 0; i < numFrames; i++) {
      printf("%d ", frames[i]);
   }
   printf("\n");
}

int isPageInFrames(int frames[], int numFrames, int page) {
   for (int i = 0; i < numFrames; i++) {
      if (frames[i] == page) {
         return 1;
      }
```

```
    }
    return 0;
}

void fifoPageReplacement(int pageReference[], int numPages, int numFrames) {
    int frames[MAX_FRAMES];
    int front = 0; // index to track the oldest page
    int pageFaults = 0;

    for (int i = 0; i < numFrames; i++) {
        frames[i] = -1;
    }
    for (int i = 0; i < numPages; i++) {
        int currentPage = pageReference[i];

        if (!isPageInFrames(frames, numFrames, currentPage)) {
            pageFaults++;
            if (isPageInFrames(frames, numFrames, -1)) {
                for (int j = 0; j < numFrames; j++) {
                    if (frames[j] == -1) {
                        frames[j] = currentPage;
                        break;
                    }
                }
            } else {
                // Replace the oldest page
                frames[front] = currentPage;
                front = (front + 1) % numFrames;
            }
        }
        // Display current state of frames
        displayFrames(frames, numFrames);
    }
    printf("\nTotal Page Faults: %d\n", pageFaults);
}


int main() {
    int numFrames, numPages;
    // Input number of page frames
    printf("Enter the number of page frames: ");
    scanf("%d", &numFrames);
    // Input number of pages
    printf("Enter the number of pages: ");
    scanf("%d", &numPages);
    int pageReference[MAX_PAGES];
    printf("Enter the page reference string:\n");
    for (int i = 0; i < numPages; i++) {
        printf("Page %d: ", i);
        scanf("%d", &pageReference[i]);
    }
    // Apply FIFO page replacement
    fifoPageReplacement(pageReference, numPages, numFrames);
    return 0;
}
```

Output:



```
Enter the number of page frames: 3
Enter the number of pages: 3
Enter the page reference string:
Page 0: 1
Page 1: 2
Page 2: 3
Page Frames:
1 -1 -1
Page Frames:
1 2 -1
Page Frames:
1 2 3

Total Page Faults: 3

--------------------------------
Process exited after 58.9 seconds with return value 0
Press any key to continue . . .
```

EXPERIMENT NO. 15

AIM / TITLE: To simulate LRU page replacement algorithm.

Theory

The LRU (Least Recently Used) page replacement algorithm replaces the page that has not been used for the longest period of time when a page fault occurs. This method assumes that pages that have been used recently are more likely to be used again soon, while pages that haven't been used recently are less likely to be used.

Algorithm: LRU Page Replacement

1 **Initialization:**

- Define a page frame list to store the current pages in memory.
- Define a page reference string (a sequence of page requests).
- Use a data structure (like a list or an array) to keep track of the order in which pages are accessed.

2 **Input:**

- Read the number of page frames (capacity of the memory).
- Read the page reference string (sequence of pages to be accessed).

3 **Page Replacement:**

- For each page in the reference string:

A) Check if the page is already in the page frame list (memory).

- If the page is present, update the access order to show that this page was recently used.
- If the page is not present (page fault), proceed to step 2.

B) If there is space available in the page frame list, add the new page to the list and update the access order.

C) If there is no space available, remove the least recently used page from the list (the page that has not been accessed for the longest time) and add the new page to the list.

4 **Output:**

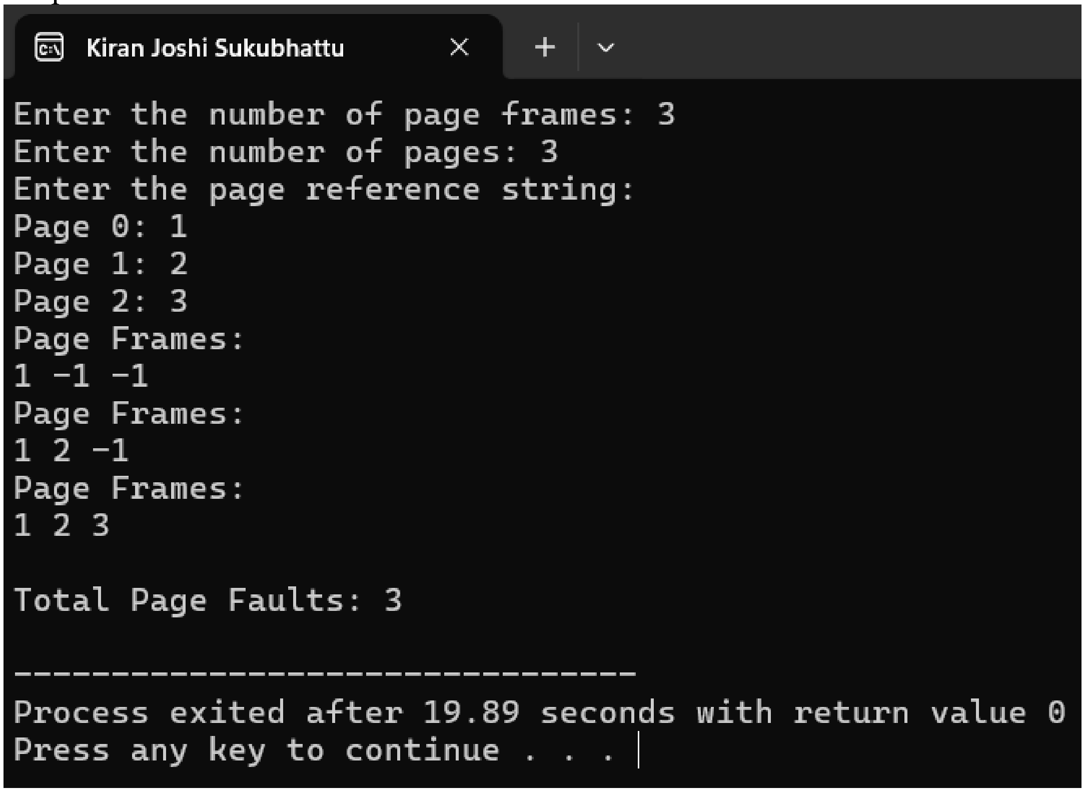
- Display the number of page faults.
- Display the final contents of the page frame list.

Source Code

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_FRAMES 100
#define MAX_PAGES 100

typedef struct {
   int page;
   int lastUsed;
} PageFrame;

void displayFrames(PageFrame frames[], int numFrames) {
   printf("Page Frames:\n");
```

```
    for (int i = 0; i < numFrames; i++) {
        printf("%d ", frames[i].page);
    }
    printf("\n");
}
int findPageIndex(PageFrame frames[], int numFrames, int page) {
    for (int i = 0; i < numFrames; i++) {
        if (frames[i].page == page) {
            return i;
        }
    }
    return -1;
}
int findLRUIndex(PageFrame frames[], int numFrames, int currentTime) {
    int lruIndex = 0;
    int minTime = frames[0].lastUsed;
    for (int i = 1; i < numFrames; i++) {
        if (frames[i].lastUsed < minTime) {
            minTime = frames[i].lastUsed;
            lruIndex = i;
        }
    }
    return lruIndex;
}
void lruPageReplacement(int pageReference[], int numPages, int numFrames) {
    PageFrame frames[MAX_FRAMES];
    int pageFaults = 0;
    int currentTime = 0;
    // Initialize page frames
    for (int i = 0; i < numFrames; i++) {
        frames[i].page = -1;
        frames[i].lastUsed = -1;
    }

    for (int i = 0; i < numPages; i++) {
        int currentPage = pageReference[i];
        int pageIndex = findPageIndex(frames, numFrames, currentPage);
        // Check if the page is already in frames
        if (pageIndex == -1) {
            // Page fault occurs
            pageFaults++;
            // Find the index of the least recently used page
            int lruIndex = findLRUIndex(frames, numFrames, currentTime);
            // Replace the LRU page with the new page
            frames[lruIndex].page = currentPage;
            frames[lruIndex].lastUsed = currentTime;
        } else {
            // Page is already in frames, update the last used time
            frames[pageIndex].lastUsed = currentTime;
        }
        // Update the current time
        currentTime++;
        // Display current state of frames
        displayFrames(frames, numFrames);
    }
    printf("\nTotal Page Faults: %d\n", pageFaults);
}

int main() {
    int numFrames, numPages;
    // Input number of page frames
    printf("Enter the number of page frames: ");
    scanf("%d", &numFrames);
    // Input number of pages
    printf("Enter the number of pages: ");
    scanf("%d", &numPages);
    int pageReference[MAX_PAGES];
    printf("Enter the page reference string:\n");
    for (int i = 0; i < numPages; i++) {
        printf("Page %d: ", i);
        scanf("%d", &pageReference[i]);
    }
    // Apply LRU page replacement
    lruPageReplacement(pageReference, numPages, numFrames);
    return 0;
}
```

Output:



```
Enter the number of page frames: 3
Enter the number of pages: 3
Enter the page reference string:
Page 0: 1
Page 1: 2
Page 2: 3
Page Frames:
1 -1 -1
Page Frames:
1 2 -1
Page Frames:
1 2 3

Total Page Faults: 3

--------------------------------
Process exited after 19.89 seconds with return value 0
Press any key to continue . . .
```

EXPERIMENT NO. 16

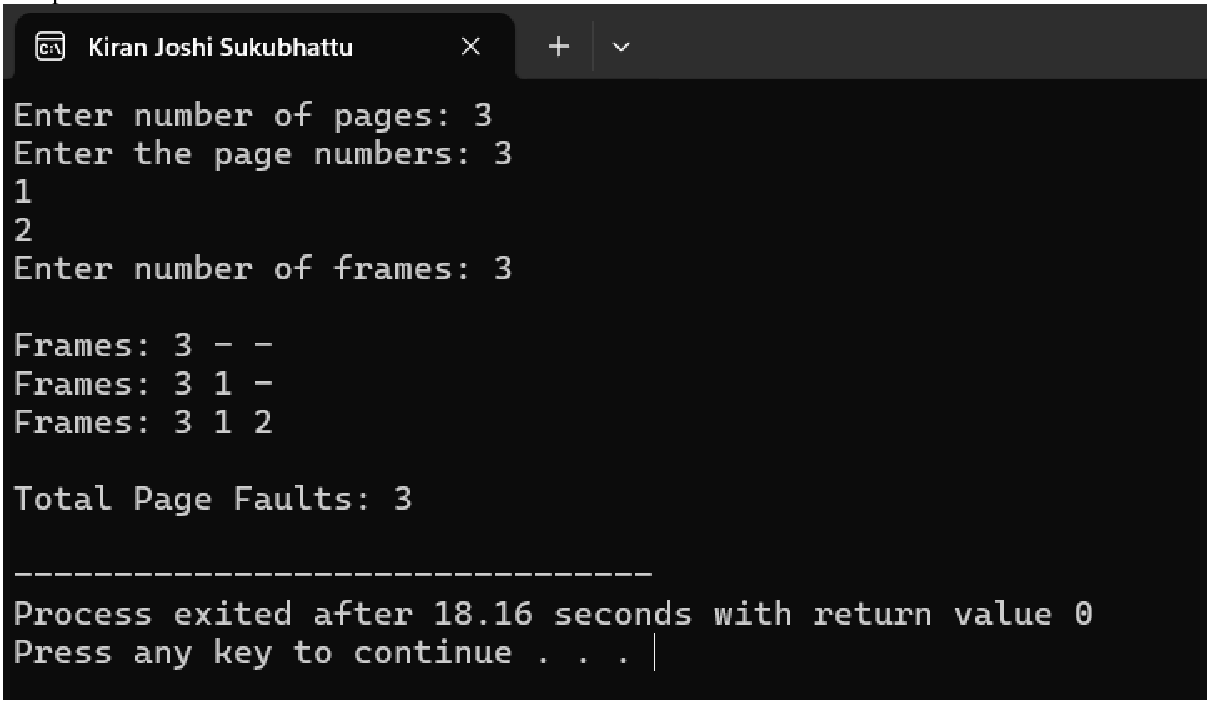
AIM / TITLE: Write a C program to simulate LFU page replacement algorithm.

The Least Frequently Used (LFU) page replacement algorithm is a method used in operating systems to manage the contents of a page table or cache. It tracks the number of times each page is accessed and replaces the page with the least number of accesses when a new page needs to be loaded, and there is no room for it in the memory.

Source Code:

```
#include <stdio.h>
#include <limits.h>
#define MAX_FRAMES 10
#define MAX_PAGES 100

int find_least_frequently_used(int freq[], int n) {
    int min = INT_MAX, min_index = -1;
    for (int i = 0; i < n; i++) {
        if (freq[i] < min) {
            min = freq[i];
            min_index = i;
        }
    }
    return min_index;
}
int main() {
    int frames[MAX_FRAMES], freq[MAX_FRAMES];
    int pages[MAX_PAGES];
    int n, f, page_faults = 0, found, pos;

    printf("Enter number of pages: ");
    scanf("%d", &n);
    printf("Enter the page numbers: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &pages[i]);
    }
    printf("Enter number of frames: ");
    scanf("%d", &f);
    for (int i = 0; i < f; i++) {
        frames[i] = -1;  // Initialize frames to -1 (empty)
        freq[i] = 0;     // Initialize frequency counts to 0
    }
    for (int i = 0; i < n; i++) {
        found = 0;
    for (int j = 0; j < f; j++) {
            if (frames[j] == pages[i]) {
                freq[j]++;
                found = 1;
                break;
            }
        }
        if (!found) {
            pos = find_least_frequently_used(freq, f);
            frames[pos] = pages[i];
            freq[pos] = 1;
            page_faults++;
        }
        printf("\nFrames: ");
        for (int j = 0; j < f; j++) {
            if (frames[j] != -1)
                printf("%d ", frames[j]);
            else
                printf("- ");
        }
    }
    printf("\n\nTotal Page Faults: %d\n", page_faults);
    return 0;
}
```

Output:



```
Enter number of pages: 3
Enter the page numbers: 3
1
2
Enter number of frames: 3

Frames: 3 - -
Frames: 3 1 -
Frames: 3 1 2

Total Page Faults: 3

--------------------------------
Process exited after 18.16 seconds with return value 0
Press any key to continue . . .
```

EXPERIMENT NO. 17

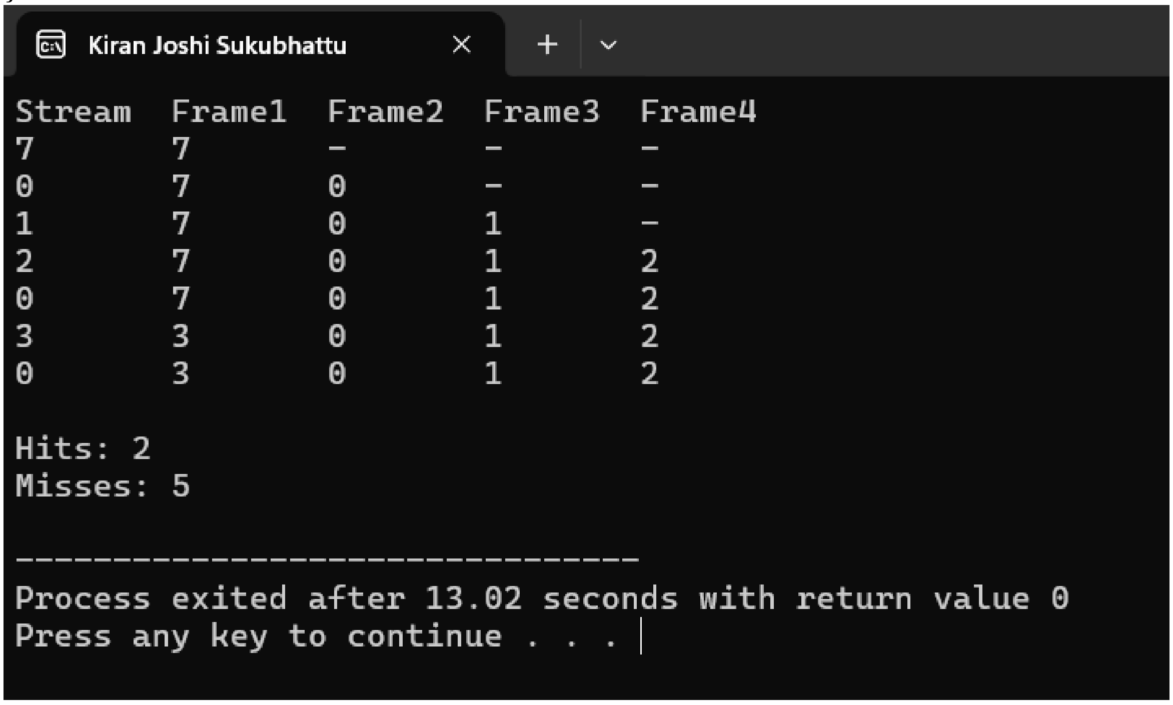
AIM / TITLE: Write a C program to simulate Optimal page replacement algorithm.

The Optimal Page Replacement algorithm is a theoretical algorithm that provides the lowest possible page fault rate for a given reference string and memory capacity. The basic idea is to replace the page that will not be used for the longest period of time in the future.

Source Code:

```
#include <stdio.h>
int search(int key, int frame_items[], int frame_occupied)
{
   for (int i = 0; i < frame_occupied; i++)
      if (frame_items[i] == key)
         return 1;
   return 0;
}

void printOuterStructure(int max_frames)
{
   printf("Stream ");
   for (int i = 0; i < max_frames; i++)
      printf("\t\tFrame%d ", i + 1);
}
void printCurrFrames(int item, int frame_items[], int frame_occupied, int
max_frames)
{
   printf("\n%d \t\t", item);
     for (int i = 0; i < max_frames; i++)
   {
      if (i < frame_occupied)
         printf("%d \t\t", frame_items[i]);
      else
         printf("- \t\t");
   }
}

int predict(int ref_str[], int frame_items[], int refStrLen, int index, int
frame_occupied)


{
   int result = -1, farthest = index;
   for (int i = 0; i < frame_occupied; i++)
   {
      int j;
      for (j = index; j < refStrLen; j++)
      {
         if (frame_items[i] == ref_str[j])
         {
                if (j > farthest)
                {
                    farthest = j;
                    result = i;
                }
                break;
            }

        if (j == refStrLen)
            return i;
    }
    return (result == -1) ? 0 : result;
}

void optimalPage(int ref_str[], int refStrLen, int frame_items[], int max_frames)
{
    int frame_occupied = 0;
    printOuterStructure(max_frames);
    int hits = 0;
    for (int i = 0; i < refStrLen; i++)
    {
        if (search(ref_str[i], frame_items, frame_occupied))
        {
            hits++;
            printCurrFrames(ref_str[i], frame_items, frame_occupied, max_frames);
            continue;
        }

        if (frame_occupied < max_frames)
        {
            frame_items[frame_occupied] = ref_str[i];
            frame_occupied++;
            printCurrFrames(ref_str[i], frame_items, frame_occupied, max_frames);
        }
        else
        {
            int pos = predict(ref_str, frame_items, refStrLen, i + 1, frame_occupied);
            frame_items[pos] = ref_str[i];
            printCurrFrames(ref_str[i], frame_items, frame_occupied, max_frames);
        }
    }
    printf("\n\nHits: %d\n", hits);
    printf("Misses: %d", refStrLen - hits);
}
```

```
int main(){

      int ref_str[] = {7, 0, 1, 2, 0, 3, 0};
    int refStrLen = sizeof(ref_str) / sizeof(ref_str[0]);
    int max_frames = 4;
    int frame_items[max_frames];

    optimalPage(ref_str, refStrLen, frame_items, max_frames);
    return 0;
}
```



Kiran Joshi Sukubhattu

```
Stream  Frame1  Frame2  Frame3  Frame4
7       7       -       -       -
0       7       0       -       -
1       7       0       1       -
2       7       0       1       2
0       7       0       1       2
3       3       0       1       2
0       3       0       1       2

Hits: 2
Misses: 5

--------------------------------
Process exited after 13.02 seconds with return value 0
Press any key to continue . . .
```

EXPERIMENT NO. 18

AIM / TITLE: Write a C program to simulate the following file organization techniques
a) Single level directory b) Two level directory c) Hierarchical

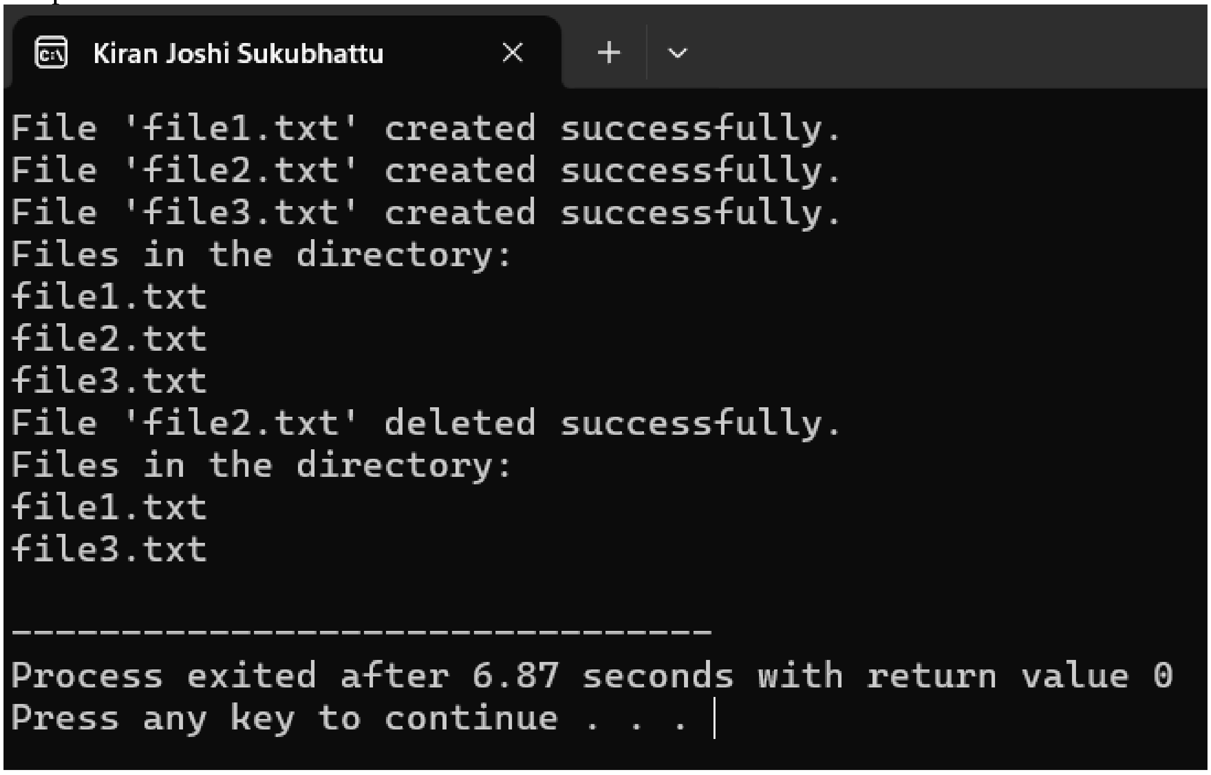
File organization refers to the way data is stored in files and how these files are managed within a file system. Proper file organization techniques are crucial for efficient data retrieval, storage management, and overall system performance.

a. Single Level Directory

A single-level directory structure is one of the simplest forms of file organization used in operating systems. In this structure, all files are stored in a single, flat directory. This directory is often referred to as the root directory, and all files are directly placed under this root without any subdirectories.

Source Code:

```
#include <stdio.h>
#include <string.h>
#define MAX_FILES 100
#define MAX_NAME_LEN 50
typedef struct {
    char name[MAX_NAME_LEN];
} File;
File directory[MAX_FILES];
int fileCount = 0;
void createFile(char filename[]) {
    if (fileCount >= MAX_FILES) {
        printf("Directory is full. Cannot create more files.\n");
        return;
    }
    for (int i = 0; i < fileCount; i++) {
        if (strcmp(directory[i].name, filename) == 0) {
            printf("File '%s' already exists.\n", filename);
            return;
        } }
    strcpy(directory[fileCount++].name, filename);
    printf("File '%s' created successfully.\n", filename);
}
void deleteFile(char filename[]) {
    // Search for the file in the directory
    for (int i = 0; i < fileCount; i++) {
        if (strcmp(directory[i].name, filename) == 0) {
            // Shift remaining files to fill the gap
            for (int j = i; j < fileCount - 1; j++) {
                directory[j] = directory[j + 1];
            }
            fileCount--;
            printf("File '%s' deleted successfully.\n", filename);
            return;
        }
    }
    printf("File '%s' not found.\n", filename);
}
void displayFiles() {
    if (fileCount == 0) {
        printf("No files in the directory.\n");
       return;
    }
    printf("Files in the directory:\n");
    for (int i = 0; i < fileCount; i++) {
        printf("%s\n", directory[i].name);
    }
}
int main() {
    // Automatically create some files
    createFile("file1.txt");
    createFile("file2.txt");
    createFile("file3.txt");
    // Display files after creation
    displayFiles();
    // Delete a specific file
    deleteFile("file2.txt");
    // Display files after deletion
    displayFiles();
    return 0;
}
```
Output:

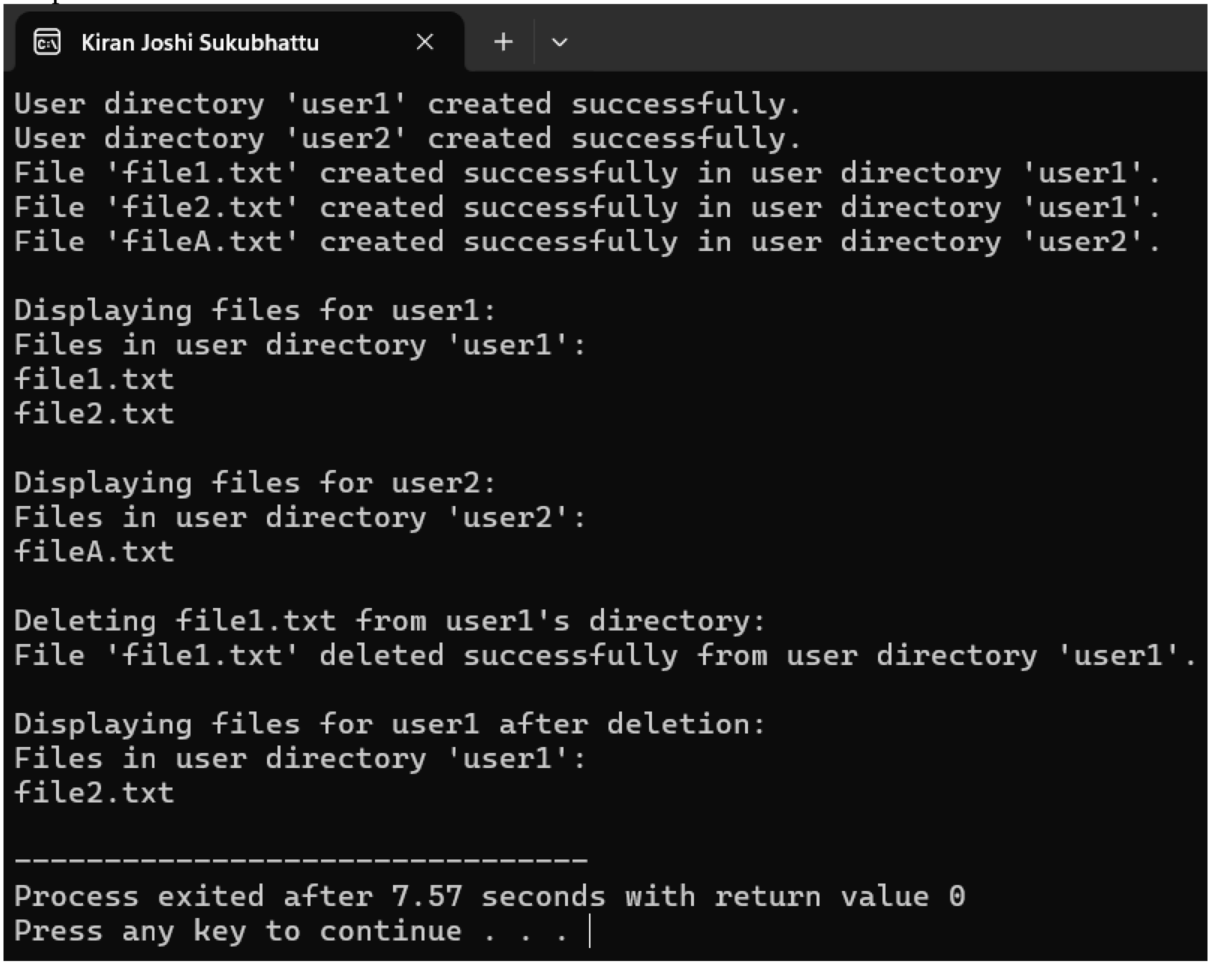


```
File 'file1.txt' created successfully.
File 'file2.txt' created successfully.
File 'file3.txt' created successfully.
Files in the directory:
file1.txt
file2.txt
file3.txt
File 'file2.txt' deleted successfully.
Files in the directory:
file1.txt
file3.txt

--------------------------------
Process exited after 6.87 seconds with return value 0
Press any key to continue . . .
```

b. Two Level Directory
The two-level directory structure is an enhancement over the single-level directory system, addressing some of its limitations by introducing a hierarchy. In this structure, there are separate directories for each user. Each user has their own directory (often called a user directory), and all files for a user are stored within this directory. This separation allows multiple users to have files with the same name without any conflict.

Source Code:
```
#include <stdio.h>
#include <string.h>

#define MAX_USERS 10
#define MAX_FILES 100
#define MAX_NAME_LEN 50

typedef struct {
    char fileName[MAX_NAME_LEN];
} File;

typedef struct {
    char userName[MAX_NAME_LEN];
    File files[MAX_FILES];
    int fileCount;
} UserDirectory;

UserDirectory userDirectories[MAX_USERS];
int userCount = 0;

void createUserDirectory(char userName[]) {
    if (userCount >= MAX_USERS) {
        printf("Maximum user limit reached. Cannot create more user directories.\n");
        return;
    }

    for (int i = 0; i < userCount; i++) {
        if (strcmp(userDirectories[i].userName, userName) == 0) {
            printf("User directory '%s' already exists.\n", userName);
            return;
        }
    }

    strcpy(userDirectories[userCount].userName, userName);
    userDirectories[userCount].fileCount = 0;
    userCount++;
    printf("User directory '%s' created successfully.\n", userName);
}

void createFile(char userName[], char fileName[]) {
    for (int i = 0; i < userCount; i++) {
        if (strcmp(userDirectories[i].userName, userName) == 0) {
            if (userDirectories[i].fileCount >= MAX_FILES) {
                printf("User '%s' directory is full. Cannot create more files.\n",
userName);
                return;
            }

            for (int j = 0; j < userDirectories[i].fileCount; j++) {
                if (strcmp(userDirectories[i].files[j].fileName, fileName) == 0) {
                    printf("File '%s' already exists in user directory '%s'.\n", fileName,
userName);
                    return;
                }
            }

            strcpy(userDirectories[i].files[userDirectories[i].fileCount++].fileName,
fileName);
            printf("File '%s' created successfully in user directory '%s'.\n", fileName,
userName);
            return;
        }
    }

    printf("User directory '%s' not found.\n", userName);
}

void deleteFile(char userName[], char fileName[]) {
    for (int i = 0; i < userCount; i++) {
        if (strcmp(userDirectories[i].userName, userName) == 0) {
            for (int j = 0; j < userDirectories[i].fileCount; j++) {
                if (strcmp(userDirectories[i].files[j].fileName, fileName) == 0) {
                    for (int k = j; k < userDirectories[i].fileCount - 1; k++) {
                        userDirectories[i].files[k] = userDirectories[i].files[k + 1];
                    }
                    userDirectories[i].fileCount--;
                    printf("File '%s' deleted successfully from user directory '%s'.\n",
fileName, userName);
                    return;
                }
            }

            printf("File '%s' not found in user directory '%s'.\n", fileName, userName);
            return;
        }
    }

    printf("User directory '%s' not found.\n", userName);
}

void displayFiles(char userName[]) {
    for (int i = 0; i < userCount; i++) {
        if (strcmp(userDirectories[i].userName, userName) == 0) {
            if (userDirectories[i].fileCount == 0) {
                printf("No files in user directory '%s'.\n", userName);
                return;
            }

            printf("Files in user directory '%s':\n", userName);
            for (int j = 0; j < userDirectories[i].fileCount; j++) {
                printf("%s\n", userDirectories[i].files[j].fileName);
            }
            return;
        }
    }

    printf("User directory '%s' not found.\n", userName);
}

int main() {
    createUserDirectory("user1");
    createUserDirectory("user2");

    createFile("user1", "file1.txt");
    createFile("user1", "file2.txt");
    createFile("user2", "fileA.txt");

    printf("\nDisplaying files for user1:\n");
    displayFiles("user1");

    printf("\nDisplaying files for user2:\n");
    displayFiles("user2");

    printf("\nDeleting file1.txt from user1's directory:\n");
    deleteFile("user1", "file1.txt");

    printf("\nDisplaying files for user1 after deletion:\n");
    displayFiles("user1");

    return 0;
}
```

Output:



```
User directory 'user1' created successfully.
User directory 'user2' created successfully.
File 'file1.txt' created successfully in user directory 'user1'.
File 'file2.txt' created successfully in user directory 'user1'.
File 'fileA.txt' created successfully in user directory 'user2'.

Displaying files for user1:
Files in user directory 'user1':
file1.txt
file2.txt

Displaying files for user2:
Files in user directory 'user2':
fileA.txt

Deleting file1.txt from user1's directory:
File 'file1.txt' deleted successfully from user directory 'user1'.

Displaying files for user1 after deletion:
Files in user directory 'user1':
file2.txt

--------------------------------
Process exited after 7.57 seconds with return value 0
Press any key to continue . . .
```

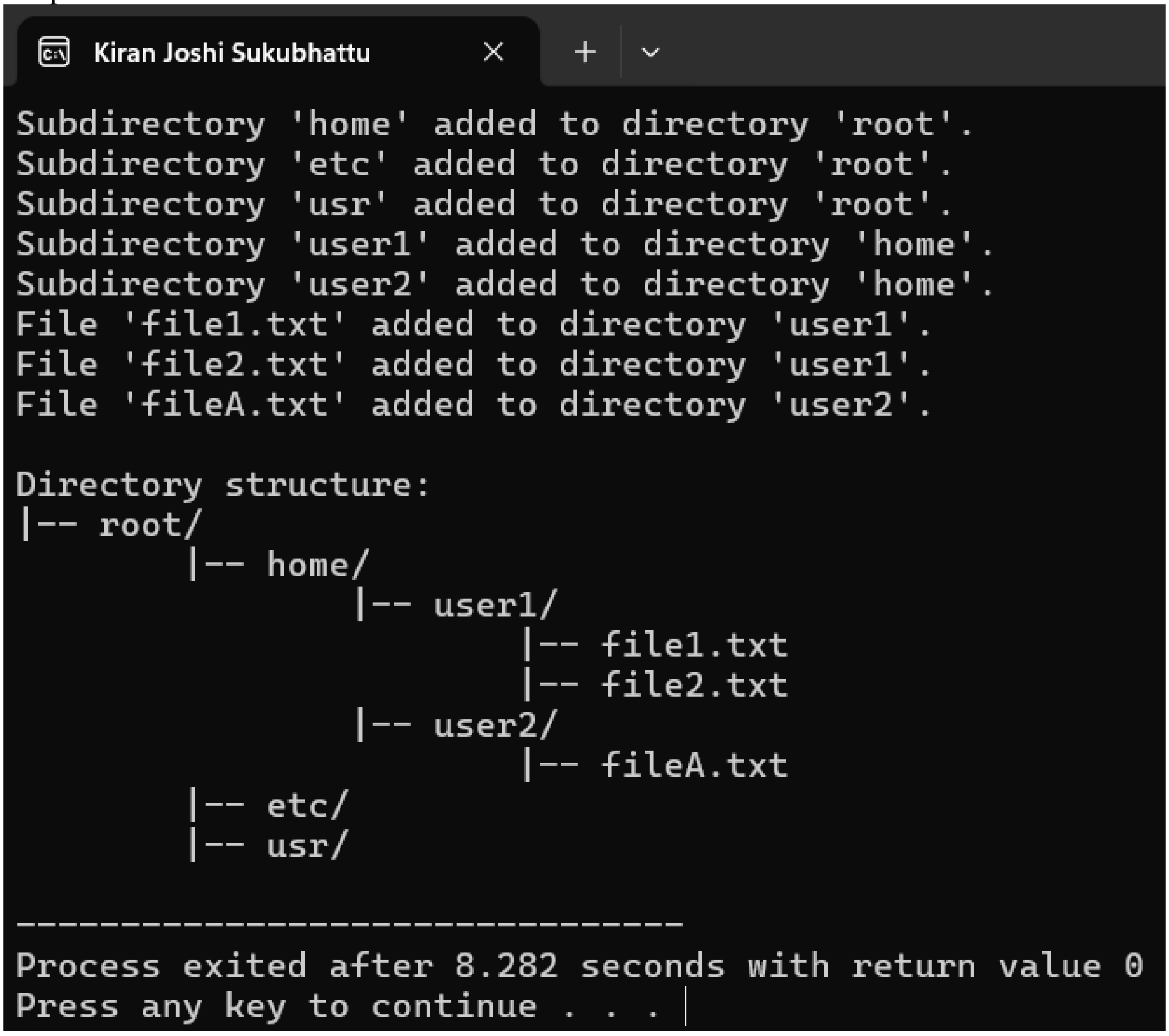
c.) Hierarchical

The hierarchical directory structure is an advanced and flexible method of organizing files in a computer system. It extends the concept of a two-level directory by allowing multiple levels of directories, creating a tree-like structure. This design supports a more organized, scalable, and efficient file management system, particularly in environments where files need to be categorized and accessed in a systematic manner.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_NAME_LEN 50
#define MAX_FILES 100
#define MAX_DIRS 10

typedef struct Directory {
    char dirName[MAX_NAME_LEN];
    struct Directory *subDirs[MAX_DIRS];
    char files[MAX_FILES][MAX_NAME_LEN];
    int fileCount;
    int dirCount;
} Directory;

Directory *createDirectory(char name[]) {
    Directory *newDir = (Directory *)malloc(sizeof(Directory));
    strcpy(newDir->dirName, name);
    newDir->fileCount = 0;
    newDir->dirCount = 0;
    return newDir;
}



void addFile(Directory *dir, char fileName[]) {
    if (dir->fileCount >= MAX_FILES) {
        printf("Directory '%s' is full. Cannot add more files.\n", dir->dirName);
        return;
    }
    strcpy(dir->files[dir->fileCount++], fileName);
    printf("File '%s' added to directory '%s'.\n", fileName, dir->dirName);
}

void addSubDirectory(Directory *parentDir, char subDirName[]) {
    if (parentDir->dirCount >= MAX_DIRS) {
        printf("Directory '%s' is full. Cannot add more subdirectories.\n",
parentDir->dirName);
        return;
    }
    parentDir->subDirs[parentDir->dirCount++] = createDirectory(subDirName);
    printf("Subdirectory '%s' added to directory '%s'.\n", subDirName,
parentDir->dirName);
}

Directory *findDirectory(Directory *root, char dirName[]) {
    if (strcmp(root->dirName, dirName) == 0) {
        return root;
    }
    for (int i = 0; i < root->dirCount; i++) {
        Directory *foundDir = findDirectory(root->subDirs[i], dirName);
        if (foundDir != NULL) {
            return foundDir;
        }
    }
    return NULL;
}


void displayDirectory(Directory *dir, int level) {
    for (int i = 0; i < level; i++) {
        printf("\t");
    }
    printf("|-- %s/\n", dir->dirName);

    for (int i = 0; i < dir->fileCount; i++) {
        for (int j = 0; j <= level; j++) {
            printf("\t");
        }
        printf("|-- %s\n", dir->files[i]);
    }

    for (int i = 0; i < dir->dirCount; i++) {
        displayDirectory(dir->subDirs[i], level + 1);
    }
}

int main() {
    Directory *root = createDirectory("root");

    addSubDirectory(root, "home");
    addSubDirectory(root, "etc");
    addSubDirectory(root, "usr");

    Directory *homeDir = findDirectory(root, "home");
    addSubDirectory(homeDir, "user1");
    addSubDirectory(homeDir, "user2");

    Directory *user1Dir = findDirectory(root, "user1");
    addFile(user1Dir, "file1.txt");
    addFile(user1Dir, "file2.txt");
    Directory *user2Dir = findDirectory(root, "user2");
    addFile(user2Dir, "fileA.txt");

    printf("\nDirectory structure:\n");
    displayDirectory(root, 0);

    return 0;
}
```

Output:



```
Subdirectory 'home' added to directory 'root'.
Subdirectory 'etc' added to directory 'root'.
Subdirectory 'usr' added to directory 'root'.
Subdirectory 'user1' added to directory 'home'.
Subdirectory 'user2' added to directory 'home'.
File 'file1.txt' added to directory 'user1'.
File 'file2.txt' added to directory 'user1'.
File 'fileA.txt' added to directory 'user2'.

Directory structure:
|-- root/
        |-- home/
                |-- user1/
                        |-- file1.txt
                        |-- file2.txt
                |-- user2/
                        |-- fileA.txt
        |-- etc/
        |-- usr/

--------------------------------
Process exited after 8.282 seconds with return value 0
Press any key to continue . . .
```

EXPERIMENT NO. 19

AIM / TITLE: Write a program to Simulate all file allocation strategies
a) Sequential b) Indexed c) Linked

File allocation strategies are methods used by operating systems to manage how files are stored on disk. These strategies determine how the blocks of a file are allocated, organized, and accessed.

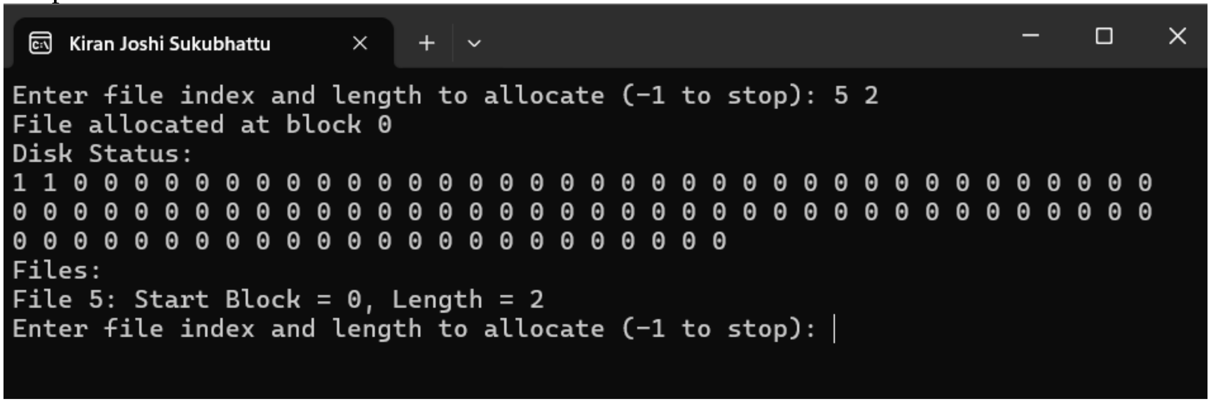
a. Sequential

Sequential (Contiguous) Allocation is one of the simplest file allocation methods. In this strategy, each file is stored in a sequence of contiguous disk blocks. This means that the blocks are physically next to each other on the disk.

Source Code:

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_BLOCKS 100
#define MAX_FILES 10

typedef struct {
    int startBlock;
    int length;
    bool allocated;
} File;

File files[MAX_FILES];
int disk[MAX_BLOCKS];

// Initialize the disk and file array
void initialize() {
    for (int i = 0; i < MAX_BLOCKS; i++) {
        disk[i] = 0; // 0 indicates free block
    }
    for (int i = 0; i < MAX_FILES; i++) {
        files[i].allocated = false;
    }
}

// Function to allocate a file using sequential allocation
int allocateFile(int fileIndex, int length) {
    int start = -1;

    // Find a contiguous sequence of free blocks of the required length
    for (int i = 0; i <= MAX_BLOCKS - length; i++) {
        bool found = true;
        for (int j = i; j < i + length; j++) {
            if (disk[j] != 0) {
                found = false;
                break;
            }
        }
        if (found) {
            start = i;
            break;
        }
    }

    // If enough contiguous blocks were found, allocate them
    if (start != -1) {
        for (int i = start; i < start + length; i++) {
            disk[i] = 1; // Mark block as allocated
        }
        files[fileIndex].startBlock = start;
        files[fileIndex].length = length;
        files[fileIndex].allocated = true;
    }

    return start;
}

// Function to display the status of the disk
void displayDiskStatus() {
    printf("Disk Status:\n");
    for (int i = 0; i < MAX_BLOCKS; i++) {
        printf("%d ", disk[i]);
    }
    printf("\n");
}

// Function to display the files and their allocation details
void displayFiles() {
    printf("Files:\n");
    for (int i = 0; i < MAX_FILES; i++) {
        if (files[i].allocated) {
            printf("File %d: Start Block = %d, Length = %d\n", i, files[i].startBlock,
files[i].length);
        }
    }
}

int main() {
    initialize();

    int fileIndex, length;

    // Simulate file allocation
    while (1) {
        printf("Enter file index and length to allocate (-1 to stop): ");
        scanf("%d", &fileIndex);
        if (fileIndex == -1) break;
        scanf("%d", &length);

        int start = allocateFile(fileIndex, length);
        if (start != -1) {
            printf("File allocated at block %d\n", start);
        } else {
            printf("Not enough contiguous space available for file %d\n", fileIndex);
        }

        displayDiskStatus();
        displayFiles();
    }

    return 0;
}
```

Output:



```
Enter file index and length to allocate (-1 to stop): 5 2
File allocated at block 0
Disk Status:
1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Files:
File 5: Start Block = 0, Length = 2
Enter file index and length to allocate (-1 to stop):
```

b. Indexed

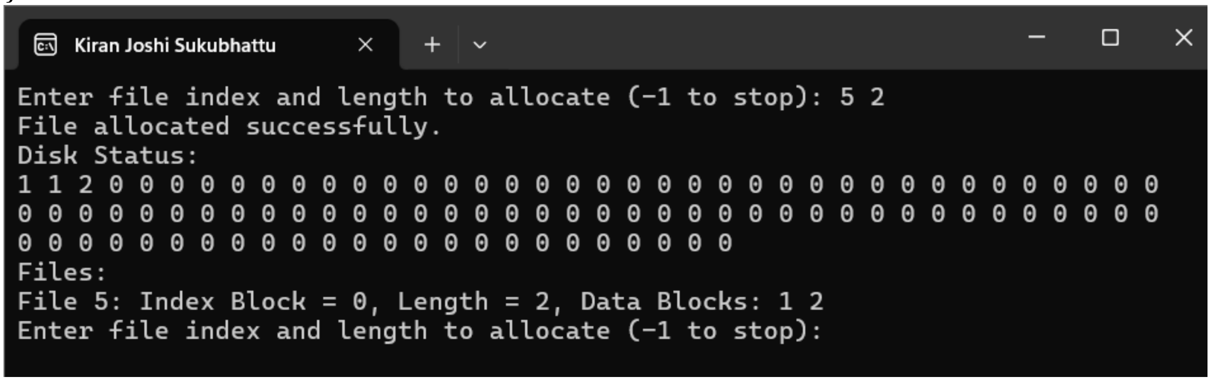
Indexed Allocation is a file allocation method in which each file has an index block that contains pointers to the actual blocks on the disk where the file's data is stored. Unlike sequential allocation, the blocks of a file do not need to be contiguous, which helps to avoid fragmentation issues.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_BLOCKS 100
#define MAX_FILES 10
#define INDEX_BLOCK_SIZE 10  // Number of pointers in an index block

typedef struct {
    int indexBlock;
    int length;
    int allocated;
} File;

File files[MAX_FILES];
```

```
int disk[MAX_BLOCKS];

// Initialize the disk and file array
void initialize() {
    for (int i = 0; i < MAX_BLOCKS; i++) {
        disk[i] = 0;  // 0 indicates free block
    }
    for (int i = 0; i < MAX_FILES; i++) {
        files[i].allocated = 0;  // 0 indicates file is not allocated
    }
}

// Function to find the next free block on the disk
int findFreeBlock() {
    for (int i = 0; i < MAX_BLOCKS; i++) {
        if (disk[i] == 0) {
            return i;
        }
    }
    return -1;
}

// Function to allocate a file using indexed allocation
int allocateFile(int fileIndex, int length) {
    int indexBlock = findFreeBlock();

    if (indexBlock == -1) {
        printf("No free blocks available for the index block.\n");
        return -1;
    }

    disk[indexBlock] = 1;  // Mark the index block as allocated
    files[fileIndex].indexBlock = indexBlock;
    files[fileIndex].length = length;
    files[fileIndex].allocated = 1;

    int blocksAllocated = 0;
    for (int i = 0; i < INDEX_BLOCK_SIZE && blocksAllocated < length; i++) {
        int dataBlock = findFreeBlock();
        if (dataBlock == -1) {
            printf("No free blocks available for data.\n");
            return blocksAllocated;  // Return the number of blocks successfully
allocated
        }
        disk[dataBlock] = 1;  // Mark the data block as allocated
        disk[indexBlock + i + 1] = dataBlock;  // Store the pointer in the index block
        blocksAllocated++;
    }
    return blocksAllocated;
}
```

```c
// Function to display the status of the disk
void displayDiskStatus() {
    printf("Disk Status:\n");
    for (int i = 0; i < MAX_BLOCKS; i++) {
        printf("%d ", disk[i]);
    }
    printf("\n");
}
// Function to display the files and their allocation details
void displayFiles() {
    printf("Files:\n");
    for (int i = 0; i < MAX_FILES; i++) {
        if (files[i].allocated) {
            printf("File %d: Index Block = %d, Length = %d, Data Blocks: ", i,
files[i].indexBlock, files[i].length);
            for (int j = 0; j < INDEX_BLOCK_SIZE && j < files[i].length; j++) {
                printf("%d ", disk[files[i].indexBlock + j + 1]);
            }
            printf("\n");
        }}}
int main() {
    initialize();
    int fileIndex, length;
    // Simulate file allocation
    while (1) {
        printf("Enter file index and length to allocate (-1 to stop): ");
        scanf("%d", &fileIndex);
        if (fileIndex == -1) break;
        scanf("%d", &length);
        int allocated = allocateFile(fileIndex, length);
        if (allocated == length) {
            printf("File allocated successfully.\n");
        } else {
            printf("Only %d out of %d blocks could be allocated.\n", allocated, length);
        }
        displayDiskStatus();
        displayFiles();
    }
    return 0;
}
```

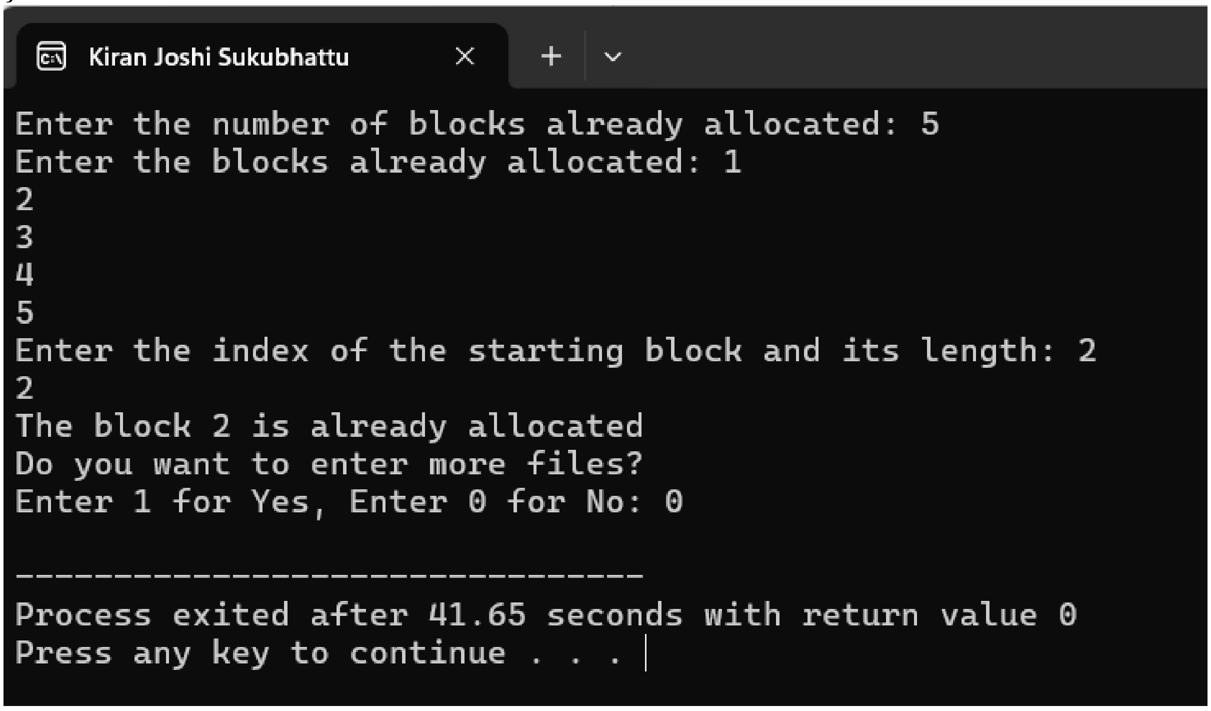


```
Enter file index and length to allocate (-1 to stop): 5 2
File allocated successfully.
Disk Status:
1 1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Files:
File 5: Index Block = 0, Length = 2, Data Blocks: 1 2
Enter file index and length to allocate (-1 to stop):
```

C. Linked
Linked Allocation is a file allocation method where each file is stored as a linked list of disk blocks. Each block contains data and a pointer to the next block. Unlike sequential allocation, the blocks of a file do not need to be contiguous on the disk, which helps in avoiding external fragmentation.

Source Code:
```
#include <stdio.h>
#include <stdlib.h>
void recursivePart(int pages[])
{
   int st, len, k, c, j;
   printf("Enter the index of the starting block and its length: ");
   scanf("%d%d", &st, &len);
   k = len;
   if (pages[st] == 0)
   {
      for (j = st; j < (st + k); j++)
      {
         if (pages[j] == 0)
         {
            pages[j] = 1;
            printf("%d------>%d\n", j, pages[j]);
         }
         else
         {
            printf("The block %d is already allocated \n", j);
            k++;
         }
      }
   }
   else
      printf("The block %d is already allocated \n", st);
   printf("Do you want to enter more files? \n");
   printf("Enter 1 for Yes, Enter 0 for No: ");
   scanf("%d", &c);
   if (c == 1)
      recursivePart(pages);
   else
      exit(0);
   return;
}
int main()
{
   int pages[50], p, a;

   for (int i = 0; i < 50; i++)
      pages[i] = 0;
   printf("Enter the number of blocks already allocated: ");
   scanf("%d", &p);
```

```c
    if (p != 0)
    {
        printf("Enter the blocks already allocated: ");
        for (int i = 0; i < p; i++)
        {
            scanf("%d", &a);
            pages[a] = 1;
        }
    }
    recursivePart(pages);
    return 0;
}
```



Kiran Joshi Sukubhattu

```
Enter the number of blocks already allocated: 5
Enter the blocks already allocated: 1
2
3
4
5
Enter the index of the starting block and its length: 2
2
The block 2 is already allocated
Do you want to enter more files?
Enter 1 for Yes, Enter 0 for No: 0

--------------------------------
Process exited after 41.65 seconds with return value 0
Press any key to continue . . .
```

Experiment no 20.

AIM / TITLE: Write a C program to simulate disk scheduling algorithms.
a) FCFS b) SCAN c) LOOK

Disk scheduling algorithms are crucial for optimizing the performance of disk operations in an operating system. They determine the order in which disk I/O requests are processed.

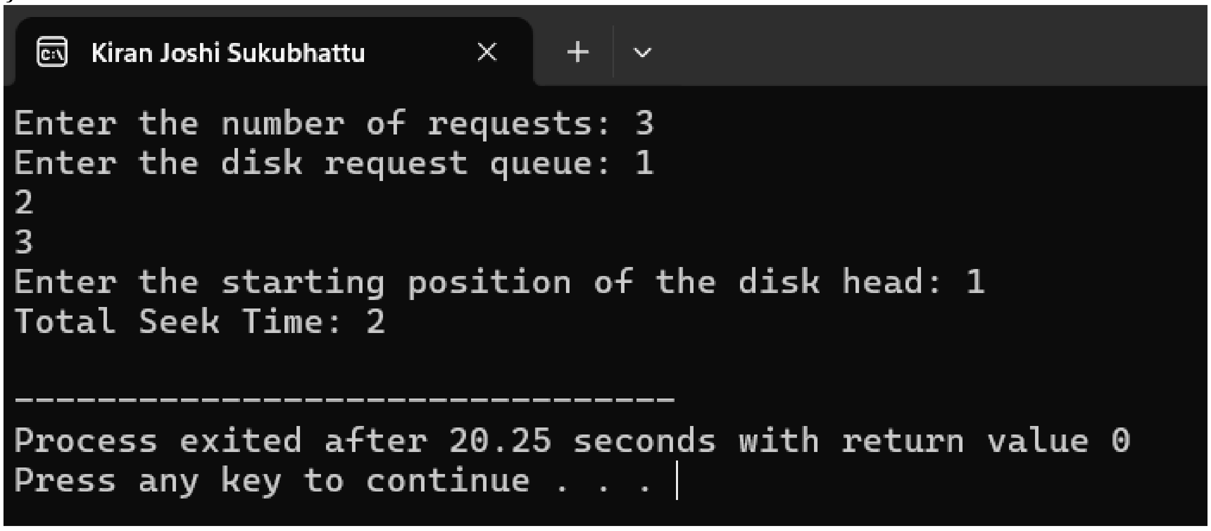
a. FCFS

FCFS is a straightforward scheduling algorithm that processes requests in the order they arrive. The disk arm moves to the track requested by the first request, then to the track requested by the second request, and so on. It operates like a queue: the first request is served first, and the last request is served last.

Source Code:

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

int calculate_seek_time(int requests[], int n, int start)
{
   int total_seek_time = 0;
   int current_position = start;

   for (int i = 0; i < n; i++)
   {
      int seek_distance = abs(requests[i] - current_position);
      total_seek_time += seek_distance;
      current_position = requests[i];
   }
   return total_seek_time;
}




int main()
{
   // Example requests (track numbers)
   // int requests[] = {98, 183, 41, 122, 14, 124, 65, 67};
   // int n = sizeof(requests) / sizeof(requests[0]);
   // int starting_position = 53;
   int n, starting_position;
   printf("Enter the number of requests: ");
   scanf("%d", &n);

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

```
    printf("Enter the starting position of the disk head: ");
    scanf("%d", &starting_position);
    int seek_time = calculate_seek_time(requests, n, starting_position);
    printf("Total Seek Time: %d\n", seek_time);
    return 0;
}
```



```
Enter the number of requests: 3
Enter the disk request queue: 1
2
3
Enter the starting position of the disk head: 1
Total Seek Time: 2

--------------------------------
Process exited after 20.25 seconds with return value 0
Press any key to continue . . .
```

b. SCAN

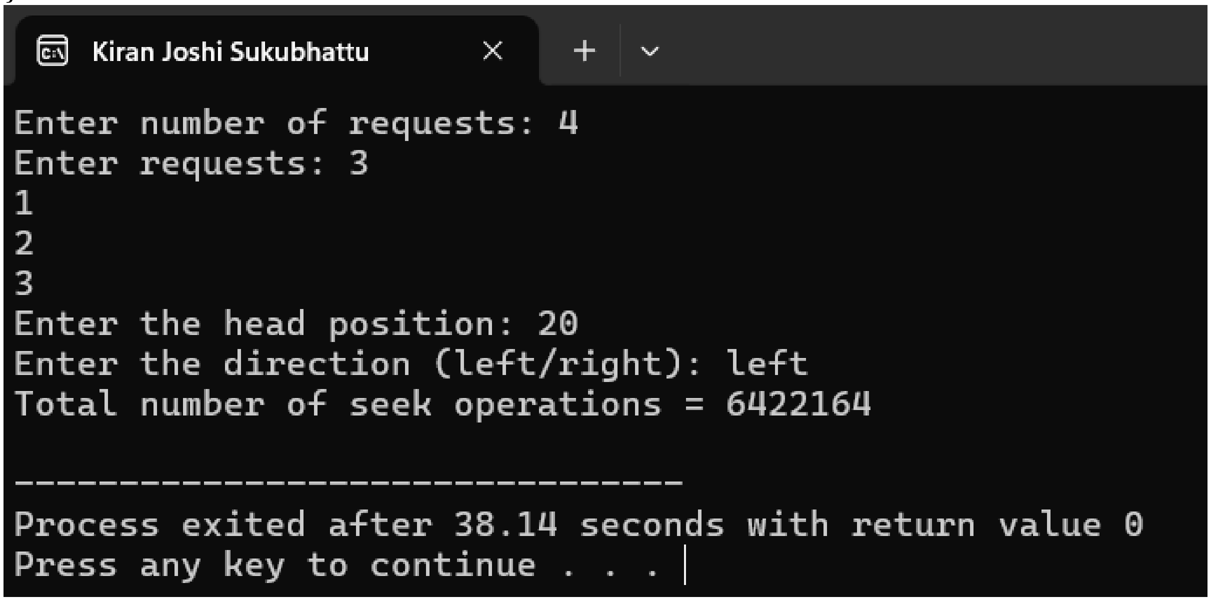
SCAN, also known as the Elevator Algorithm, is a disk scheduling algorithm designed to improve performance by reducing the number of disk head movements compared to FCFS. It works by moving the disk arm (or head) in one direction, servicing all requests until it reaches the end of the disk or the last request in that direction. Once the end is reached, the disk arm reverses direction and services requests in the opposite direction.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int size = 8;
int disk_size = 200;

int cmpfunc(const void *a, const void *b);

void SCAN(int arr[], int head, char *direction)
{
    int seek_count = 0;
    int distance, cur_track;
    int left[size], right[size];
    int left_count = 0, right_count = 0;

    if (strcmp(direction, "left") == 0)
        left[left_count++] = 0;
    else if (strcmp(direction, "right") == 0)
```

```
        right[right_count++] = disk_size - 1;

    for (int i = 0; i < size; i++)
    {
        if (arr[i] < head)
            left[left_count++] = arr[i];
        if (arr[i] > head)
            right[right_count++] = arr[i];
    }

    qsort(left, left_count, sizeof(int), cmpfunc);
    qsort(right, right_count, sizeof(int), cmpfunc);

    int seek_sequence[size];
    int sequence_count = 0;

    int run = 2;
    while (run--)
    {
        if (strcmp(direction, "left") == 0)
        {
            for (int i = left_count - 1; i >= 0; i--)
            {
                cur_track = left[i];
                seek_sequence[sequence_count++] = cur_track;
                distance = abs(cur_track - head);
                seek_count += distance;
                head = cur_track;
            }
            strcpy(direction, "right");
        }
        else if (strcmp(direction, "right") == 0)
        {
            for (int i = 0; i < right_count; i++)
            {
                cur_track = right[i];
                seek_sequence[sequence_count++] = cur_track;
                distance = abs(cur_track - head);
                seek_count += distance;
                head = cur_track;
            }
            strcpy(direction, "left");
        }
    }
    printf("Total number of seek operations = %d\n", seek_count);
}

int cmpfunc(const void *a, const void *b)
{
    return (*(int *)a - *(int *)b);
```

```
}

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

    int arr[n];
    printf("Enter requests: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

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

    char direction[10];
    printf("Enter the direction (left/right): ");
    scanf("%s", direction);

    SCAN(arr, head, direction);

    return 0;
}
```



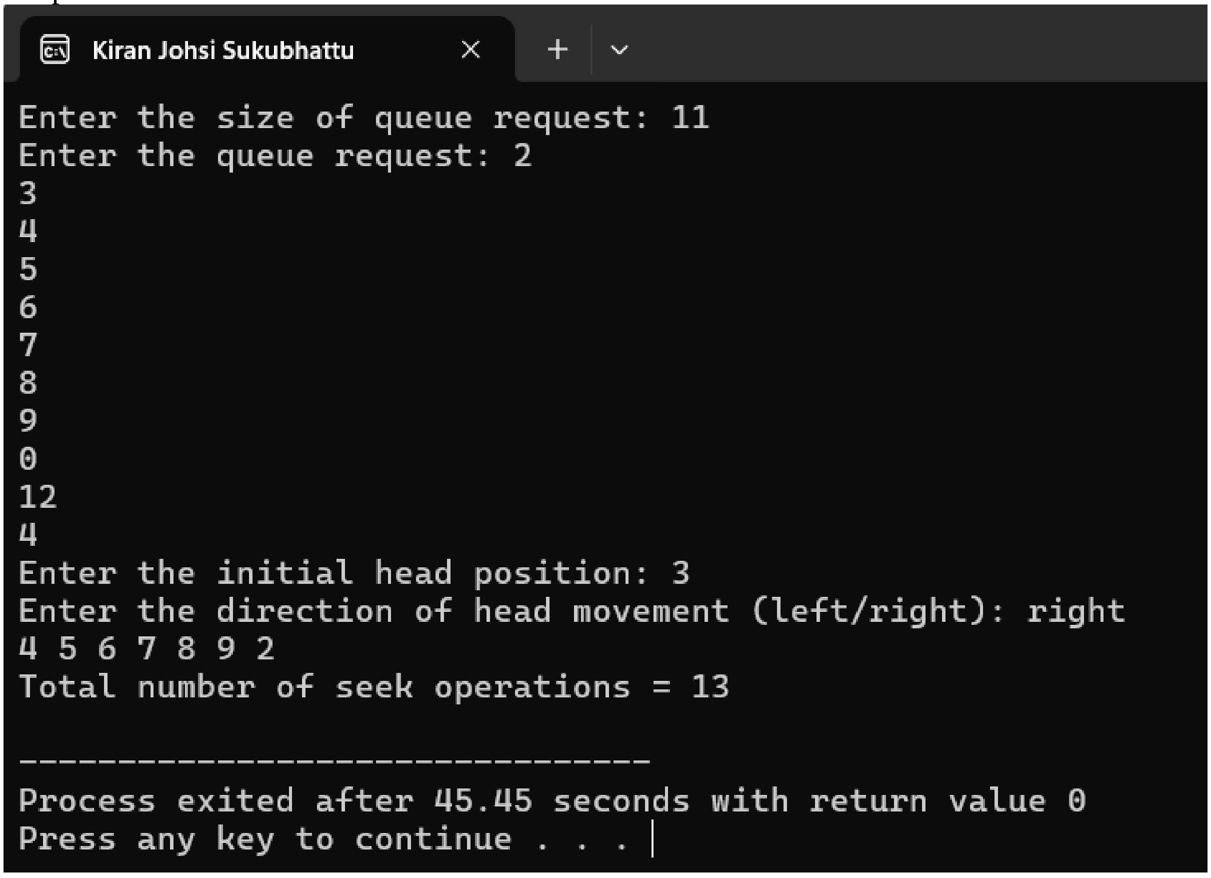
```
Enter number of requests: 4
Enter requests: 3
1
2
3
Enter the head position: 20
Enter the direction (left/right): left
Total number of seek operations = 6422164

--------------------------------
Process exited after 38.14 seconds with return value 0
Press any key to continue . . .
```

c. LOOK
LOOK is a variation of the SCAN disk scheduling algorithm that optimizes the movement of the disk arm to further reduce seek time and improve performance. Unlike SCAN, where the disk arm moves all the way to the end of the disk before reversing direction, LOOK only moves as far as the last request in the current direction before reversing.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
const int size = 8;
// Compare function for qsort
int compare(const void *a, const void *b)
{
    return (*(int *)a - *(int *)b);
}

void LOOK(int arr[], int head, char *direction)
{
    int seek_count = 0;
    int distance, cur_track;
    int left[size], right[size];
    int left_size = 0, right_size = 0;
    for (int i = 0; i < size; i++)
    {
        if (arr[i] < head)
            left[left_size++] = arr[i];
        if (arr[i] > head)
            right[right_size++] = arr[i];
    }

    qsort(left, left_size, sizeof(int), compare);
    qsort(right, right_size, sizeof(int), compare);
    int run = 2;
    while (run--)
    {
        if (strcmp(direction, "left") == 0)
        {
            for (int i = left_size - 1; i >= 0; i--)
            {
                cur_track = left[i];
                printf("%d ", cur_track);
                distance = abs(cur_track - head);
                seek_count += distance;
                head = cur_track;
            }
            strcpy(direction, "right");
        }
        else if (strcmp(direction, "right") == 0)
        {
            for (int i = 0; i < right_size; i++)
            {
                cur_track = right[i];
                printf("%d ", cur_track);
                distance = abs(cur_track - head);
                seek_count += distance;
                head = cur_track;
            }
            strcpy(direction, "left");
        }
    }
    printf("\nTotal number of seek operations = %d\n", seek_count);
}
int main()
{
    int n, head;
    printf("Enter the size of queue request: ");
    scanf("%d", &n);
    int arr[n];
    printf("Enter the queue request: ");
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    printf("Enter the initial head position: ");
    scanf("%d", &head);
    char direction[10];
    printf("Enter the direction of head movement (left/right): ");
    scanf("%s", direction);
    LOOK(arr, head, direction);
    return 0;
}
```

Output:



```
Enter the size of queue request: 11
Enter the queue request: 2
3
4
5
6
7
8
9
0
12
4
Enter the initial head position: 3
Enter the direction of head movement (left/right): right
4 5 6 7 8 9 2
Total number of seek operations = 13

--------------------------------
Process exited after 45.45 seconds with return value 0
Press any key to continue . . .
```