Department of Computer Application
National Institute of Technology
Kurukshetra , Kurukshetra -136119

Operating System Lab
MCA-134
(2024-27)

Submitted by:

Name: Shrushranto Rajbongshi

Semester: 2nd

Section: Group-2

Roll no: 524110026

Submitted to:

Dr. Nidhi Gupta

Date:30-04-25

DECLARATION

I, **Shrushranto Rajbongshi**, bearing Roll No. **524110026**, hereby declare that the **Operating System Lab Manual** submitted by me is the result of my own sincere efforts, dedication, and understanding.

Each experiment and corresponding program included in this manual has been independently designed, implemented, and executed by me using the **C programming language** on the **Ubuntu operating system**. I have ensured that all code is written following appropriate syntax, logical structure, and industry-standard programming practices.
Throughout the completion of this lab, I have applied the concepts learned during lectures and practical sessions. While I have referred to authorized academic materials, textbooks, and class notes to reinforce my understanding, all implementations reflect my own logic, experimentation, and problem-solving skills.

No part of this submission has been copied from unauthorized sources, websites, or peers. I have taken every measure to ensure that the content of this lab manual upholds the principles of academic integrity and originality.
This lab work has been prepared in accordance with the ethical standards and academic guidelines prescribed by the institution. I take full responsibility for the authenticity of this submission.
I believe this manual is a genuine reflection of my learning and practical understanding of **Operating System concepts and their implementation**.

Date: 30-04-2025
Signature: *Shrushranto Rajbongshi*

INDEX

1. Study Of Hardware And Software Requirements Of Different Operating Systems
(UNIX , LINUX , WINDOWS XP , WINDOWS 7/8/10)

UNIX

Unix was developed in the late 1960s and early 1970s. The project began when Ken Thompson and Dennis Ritchie, working at Bell Labs, rewrote an operating system to play a space travel game on a DEC PDP-7 computer. They called the new system UNICS, a pun on MULTICS (Multiplexed Information and Computing Service), the original project from which Unix was derived. The name was later shortened to Unix. Development started in 1969, and the first manual was published internally in November 1971. Unix was announced outside Bell Labs in October 1973.

The hardware and software requirements for a Unix operating system vary depending on the specific implementation of Unix that you are using. Here are some general guidelines:

Hardware Requirements:

- **Processor**: A Unix system requires a processor that is compatible with the instruction set of the Unix kernel.
- **Memory**: The minimum memory requirement for Unix varies depending on the version, but generally, you'll need at least 1 GB of RAM.
- **Storage**: You'll need a hard drive with sufficient space to install the operating system, as well as any applications and data that you plan to use.
- **Input/Output Devices**: A keyboard and a pointing device (such as a mouse) are necessary for interacting with the Unix system.

Software Requirements:

- **Kernel**: A Unix system requires a Unix kernel, which is the core of the operating system.
- **Shell**: A Unix shell is a command-line interface that allows users to interact with the system. Common Unix shells include bash, csh, and ksh.
- **Utilities**: Unix comes with a suite of built-in utilities, including text editors, file managers, and network tool.

LINUX

Windows XP was released to manufacturing on August 24, 2001, and became generally available for retail sale on October 25, 2001. Development of Windows XP began in the late 1990s under the codename "Neptune," which was intended specifically for mainstream consumer use. Later, both Neptune and another project named "Odyssey" were scrapped in favor of a single operating system codenamed "WhistlLinux was created in 1991 by Linus Torvalds, a Finnish computer science student at the University of Helsinki. Initially, Torvalds aimed to develop a free operating system kernel as a hobby project. He began working on the project in 1991 and publicly released the first version of the Linux kernel on September 17, 1991.

The development of Linux was partly inspired by the lack of a freely available Unix-like kernel at the time. Torvalds had been using MINIX, a Unix-like operating system designed for educational purposes, but he wanted a more powerful kernel that could run on his personal computer with an 80386 processor. He developed Linux using the GNU C Compiler on MINIX and released the first version of the Linux kernel as open-source software.

The name "Linux" is a combination of Torvalds' first name, Linus, and "Unix," the operating system that inspired his project. Initially, Torvalds considered calling his invention "Freax," but it was later named "Linux" by Ari Lemmke, a volunteer administrator at the FTP server where the files were stored.

The hardware and software requirements for Linux Operating System can vary depending on the specific distribution of Linux being used. However, here are some general requirements that are typical for most Linux systems:

Hardware requirements:

- **Processor**: 1 GHz or faster
- **RAM**: 1 GB (minimum), 2 GB or more
- **Hard drive space**: At least 20 GB of free space
- **Graphics card and monitor**: Graphics card with a resolution of 1024x768 or higher; monitor capable of displaying at least 256 colors
- CD/DVD drive or USB port (for installation)

Software requirements:

- **Kernel**: Linux kernel (varies by distribution)
- **Desktop environment or window manager**: GNOME, KDE, Xfce, LXDE, etc. (varies by distribution)
- **Compiler**: GCC (GNU Compiler Collection)
- **Text editor**: Vim, Emacs, Nano, etc. (optional)

WINDOWS XP

Windows XP was released to manufacturing on August 24, 2001, and became generally available for retail sale on October 25, 2001. Development of Windows XP began in the late 1990s under the codename "Neptune," which was intended specifically for mainstream consumer use. Later, both Neptune and another project named "Odyssey" were scrapped in favor of a single operating system codenamed "Whistler," which aimed to unify both consumer and business markets under a single Windows NT platform.

Here are the hardware and software requirements for Windows XP

Hardware requirements:

- **Processor**: Pentium 233 MHz or higher
- **RAM**: 64 MB (minimum), 128 MB or more (recommended)
- **Hard drive space**: At least 1.5 GB of free space
- CD/DVD drive or USB port (for installation)

Software requirements:

- **Operating system**: Windows XP (32-bit or 64-bit)
- **Service Pack**: Service Pack 3 (SP3)
- **Internet Explorer**: Internet Explorer 6 or later (recommended for security updates)
- **Media Player**: Windows Media Player 9 or late

WINDOWS 7/8/10

Windows 7, Windows 8, and Windows 10 are major releases of the Windows NT operating system developed by Microsoft. Here's a summary of their release timelines and how they came about:
Regarding the support lifecycle:

Support for Windows 8.1 ended on January 10, 2023.
Support for Windows 7 ended on January 14, 2020, for most editions, but extended support for some versions lasted until January 10, 2023.
Windows 10 is currently supported, with the next major release, Windows 11, being available since October 5, 2021.
To upgrade from Windows 7 or 8 to Windows 10, you can use the MediaCreationTool provided by Microsoft. This tool will determine the appropriate edition of Windows 10 based on your current version and perform the upgrade process.

Common hardware and software requirements for Windows 10 and 11 operating systems:

Hardware requirements:

- **Processor**: 1 gigahertz (GHz) or faster processor or SoC
- **RAM**: 1 gigabyte (GB) for 32-bit or 2 GB for 64-bit
- **Hard disk space**: 16 GB for 32-bit OS or 20 GB for 64-bit OS
- **Graphics card**: DirectX 9 or later with WDDM 1.0 driver
- **Display**: 800 x 600

Software requirements:

- Internet connection
- Microsoft account

2. Execute various UNIX system calls for Process Management, File Management, and Input/Output device Management

Process Management System Calls:

In an operating system, process management system calls are used to manage processes. These calls allow programs to interact with the OS to control various aspects of process creation, execution, and termination. Below are some common process management system calls:

1. **Fork:**
    - Description: Creates a new process by duplicating the calling process. The new process is called the child process.
    - Behaviour: The new child process gets a copy of the parent's memory, but it has a unique process ID (PID).
2. **Exec:**
    - Description: Replaces the current process's image with a new one. It loads a different program into the current process's address space.
    - Behaviour: The current process is replaced by a new process, but it retains the same PID.
3. **Wait:**
    - Description: The calling process waits for one of its child processes to terminate.
    - Behaviour: The process will pause execution until the child process finishes, allowing the parent process to clean up or collect the exit status.
4. **Exit:**
    - Description: Terminates the calling process.
    - Behaviour: The process terminates, and the OS releases all resources associated with it. A status code can be returned to the parent process.
5. **Kill:**
    - Description: Sends a signal to a process to terminate or handle a specific event (like stopping or pausing).
    - Behaviour: A signal (e.g., SIGKILL, SIGSTOP) is sent to the specified process, allowing the process to respond accordingly.

CODE:

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

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        exit(1);
    }

    if (pid == 0) {
        printf("Child Process (PID: %d) executing ls...\n", getpid());
        char *args[] = {"/bin/ls", "-l", NULL};
        execvp(args[0], args);

        perror("execvp failed");
        exit(1);
    } else {

        printf("Parent Process (PID: %d), Child PID: %d\n", getpid(), pid);
        sleep(2);
        printf("Parent is killing the child process (PID: %d)\n", pid);
        kill(pid, SIGKILL);

        wait(NULL);
        printf("Child process terminated\n");
    }
     return 0;
}
```

OUTPUT:

```
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ nano fork.c
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ gcc fork.c -o fork
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ ./fork
Parent Process (PID: 1942), Child PID: 1943
Child Process (PID: 1943) executing ls...
total 24
-rwxr-xr-x 1 shru28_ shru28_ 16392 Apr 30 17:01 fork
-rw-r--r-- 1 shru28_ shru28_   756 Apr 30 17:00 fork.c
Parent is killing the child process (PID: 1943)
Child process terminated
```

File Management System Calls:

File management system calls in an operating system provide an interface for programs to interact with the file system. These system calls allow processes to create, open, read, write, modify, and delete files. Below are the common file management system calls:

1. Create a File:

- Description: Creates a new file or opens an existing one.
- Usage: create("filename", mode);
- Behaviour: If the file does not exist, it is created. The mode specifies the file's permissions.

2. Open a File:

- Description: Opens an existing file for reading, writing, or both.
- Usage: open("filename", O_RDONLY);
- Behaviour: Returns a file descriptor that is used for subsequent operations.

3. Read from a File:

- Description: Reads data from a file into a buffer.
- Usage: read(fd, buffer, size);
- Behaviour: Reads a specified number of bytes from the file descriptor into a buffer.

4. Write to a File:

- Description: Writes data from a buffer to a file.
- Usage: write(fd, buffer, size);
- Behaviour: Writes the given number of bytes from the buffer to the file.

5.Close a File:

- Description: Closes an open file.
- Usage: close(fd);
- Behaviour: Releases the file descriptor and flushes any pending writes.

CODE:

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

#define FILE_NAME "testfile.txt"

int main() {
   int fd;
   char write_data[] = "Hello, I'am Shrushranto Rajbongshi learning linux File closed successfully.";
   char read_data[50];
   fd = open(FILE_NAME, O_CREAT | O_RDWR, 0644);
   if (fd < 0) {
      perror("Error opening file");
      exit(1);
   }
   printf("File opened/created successfully.\n");

   write(fd, write_data, sizeof(write_data));
   printf("Data written to file.\n");

   lseek(fd, 0, SEEK_SET);

   read(fd, read_data, sizeof(read_data));
   printf("Data read from file: %s", read_data);

   close(fd);
   printf("File closed successfully.\n");

   return 0;
}
```

OUTPUT:

```
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ nano file.c
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ gcc file.c -o file
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ ./file
File opened/created successfully.
Data written to file.
Data read from file: Hello, I'am Shrushranto Rajbongshi learning linux File closed successfully.
```

3.Implement CPU scheduling schemes: FCFS, SJF, SRTF, Priority, Round Robin, Multi-Level Queue, and find out the average , turn-around time, avg. waiting time, response time, and throughput

What is CPU Scheduling?

CPU scheduling is the process by which the operating system determines which process in the ready queue will be executed by the CPU. It is a fundamental part of multitasking and ensures that system resources are used efficiently while maintaining fairness among processes.

- **First-Come, First-Served (FCFS):**
- **Shortest Job First (SJF):**
- **Priority Scheduling:**
- **Round Robin (RR):**
- **Multi-Level Queue Scheduling:**

And the metric of measurement of these scheduling algorithms are:

- **Average Turnaround Time:** Sum of completion times minus arrival times of all processes divided by the number of processes.
- **Average Waiting Time:** Sum of waiting times (time spent in the ready queue) for all processes divided by the number of processes.
- **Response Time:** Time from submitting a request until the first response is produced.
- **Throughput:** Number of processes completed per unit of time.

Formulas –

1. Average Turnaround Time:

• Average Turnaround Time (ATT) = (Sum of Turnaround Times for all processes) / (Number of processes)

• Turnaround Time (TAT) for a process = Completion Time - Arrival Time

2. Average Waiting Time:

• Average Waiting Time (AWT) = (Sum of Waiting Times for all processes) / (Number of processes)

• Waiting Time for a process = Turnaround Time - Burst Time

3. Response Time:

• Response Time for a process = Time from submission until the first response is produced. It can vary depending on the scheduling and context switching overhead.

4. Throughput:

• Throughput = Number of processes completed / Total time.

1.First-Come First-Served (FCFS):

Processes are executed in the order they arrive. Simple and easy to implement, it suffers from poor average waiting times, especially for longer processes arriving first, causing potential delays for shorter ones.

Characteristics:

- Non-preemptive.
- Processes are executed in the order they arrive.
- Simple and easy to implement.
- Can lead to the convoy effect, where short processes have to wait for long processes to finish.

Program:

```
#include<stdio.h>


struct Process {
   int pid, at, bt , wt, tat, ct;
};
int main() {
   int n;
   printf("Enter the number of processes: ");
   scanf("%d", &n);

   struct Process p[n];

   for(int i = 0; i< n; i++) {
      p[i].pid = i + 1;
      printf("Enter Arrival Time and Burst Time for Process %d: ",p[i].pid);
      scanf("%d %d", &p[i].at, &p[i].bt);
   }

   int total_wt = 0, total_tat = 0;

   for(int i =0;i<n-1;i++) {
      for(int j = i+1;j<n;j++) {
         if(p[i].at > p[j].at) {
```

```
                struct Process temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }

    int current_time = 0;
    for(int i = 0; i< n; i++) {
        if(current_time < p[i].at) {
            current_time = p[i].at;
        }
        p[i].wt = current_time - p[i].at;
        p[i].ct = current_time + p[i].bt;
        p[i].tat = p[i].ct - p[i].at;
        current_time = p[i].ct;
    }

    printf("\nPID\tAT\tBT\tCT\tWT\tTAT\n");
    for(int i = 0; i< n; i++) {
        total_wt += p[i].wt;
        total_tat += p[i].tat;
        printf("%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].ct, p[i].wt, p[i].tat);
    }

    printf("\nAvg WT: %.2f", (float)total_wt / n);
    printf("\nAvg TAT: %.2f", (float)total_tat / n);
    printf("\nThroughput: %.2f\n",(float)n / p[n - 1].ct);

    return 0;
}
```

OUTPUT:

```
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ nano fcfs.c
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ gcc fcfs.c -o fcfs
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ ./fcfs
Enter the number of processes: 5
Enter Arrival Time and Burst Time for Process 1: 0 2
Enter Arrival Time and Burst Time for Process 2: 1 3
Enter Arrival Time and Burst Time for Process 3: 2 4
Enter Arrival Time and Burst Time for Process 4: 3 8
Enter Arrival Time and Burst Time for Process 5: 4 4

PID     AT      BT      CT      WT      TAT
1       0       2       2       0       2
2       1       3       5       1       4
3       2       4       9       3       7
4       3       8       17      6       14
5       4       4       21      13      17

Avg WT: 4.60
Avg TAT: 8.80
Throughput: 0.24
```

2. Shortest Job First (SJF):

Prioritizes shorter jobs, minimizing average waiting times. It can lead to starvation for longer jobs if shorter ones constantly arrive. Balancing fairness and efficiency is critical in its implementation.

Characteristics:

• Provides optimal average waiting time.

• May cause starvation of longer processes.

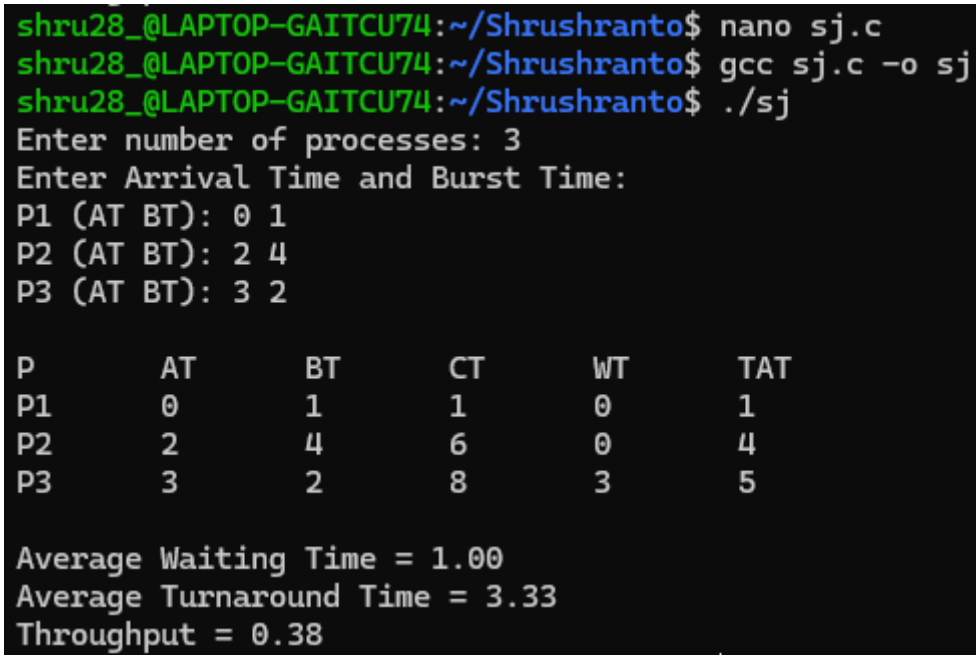
Program

```
#include <stdio.h>


int main() {
    int A[100][6];
    int i, j, n, total_wt = 0, total_tat = 0, index, temp;
    float avg_wt, avg_tat, throughput;

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

    printf("Enter Arrival Time and Burst Time:\n");
    for (i = 0; i < n; i++) {
        printf("P%d (AT BT): ", i + 1);
        scanf("%d %d", &A[i][1], &A[i][2]);
        A[i][0] = i + 1;
    }

    // Sorting based on Arrival Time, then Burst Time
    for (i = 0; i < n; i++) {
        index = i;
        for (j = i + 1; j < n; j++) {
            if (A[j][1] < A[index][1] || (A[j][1] == A[index][1] && A[j][2] < A[index][2])) {
                index = j;
            }
        }
        for (int k = 0; k < 3; k++) {
```

```
            temp = A[i][k];
            A[i][k] = A[index][k];
            A[index][k] = temp;
        }
    }

    int current_time = 0;

    for (i = 0; i < n; i++) {
        if (current_time < A[i][1]) {
            current_time = A[i][1];
        }
        A[i][3] = current_time - A[i][1]; // Waiting Time
        A[i][5] = current_time + A[i][2]; // Completion Time
        A[i][4] = A[i][5] - A[i][1]; // Turnaround Time
        current_time = A[i][5];
        total_wt += A[i][3];
        total_tat += A[i][4];
    }

    avg_wt = (float)total_wt / n;
    avg_tat = (float)total_tat / n;
    throughput = (float)n / current_time;

    printf("\nP\tAT\tBT\tCT\tWT\tTAT\n");
    for (i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\t%d\n", A[i][0], A[i][1], A[i][2], A[i][5], A[i][3],
A[i][4]);
    }

    printf("\nAverage Waiting Time = %.2f", avg_wt);
    printf("\nAverage Turnaround Time = %.2f", avg_tat);
    printf("\nThroughput = %.2f\n", throughput);
```

```
    return 0;
}
```

OUTPUT:



3.Shortest Remaining Time First (SRTF) - Preemptive SJF:

```
#include <stdio.h>

int main() {
    int n;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    int arrival[n], burst[n], remaining[n], completion[n];
    int waiting[n], turnaround[n], processID[n];
    int time = 0, completed = 0, minRemainingTime, minIndex;
    float totalWT = 0, totalTAT = 0, avgWT, avgTAT, throughput;

    for (int i = 0; i < n; i++) {
        processID[i] = i + 1;
        printf("\nEnter details for process %d:\n", processID[i]);
        printf("Arrival time: ");
        scanf("%d", &arrival[i]);
        printf("Burst time: ");
        scanf("%d", &burst[i]);
        remaining[i] = burst[i];
    }

    while (completed < n) {
        minRemainingTime = 999;
        minIndex = -1;

        for (int i = 0; i < n; i++) {
            if (arrival[i] <= time && remaining[i] > 0 && remaining[i] < minRemainingTime) {
                minRemainingTime = remaining[i];
                minIndex = i;
            }
        }
```

```
    if (minIndex != -1) {
        remaining[minIndex]--;
        time++;

        if (remaining[minIndex] == 0) {
            completed++;
            completion[minIndex] = time;
        }
    } else {
        time++;
    }
}

for (int i = 0; i < n; i++) {
    turnaround[i] = completion[i] - arrival[i];
    waiting[i] = turnaround[i] - burst[i];
    totalTAT += turnaround[i];
    totalWT += waiting[i];
}

avgTAT = totalTAT / n;
avgWT = totalWT / n;
throughput = (float)n / time;

printf("\nPID\tAT\tBT\tCT\tTAT\tWT\n");
for (int i = 0; i < n; i++) {
    printf("%d\t%d\t%d\t%d\t%d\t%d\n",
        processID[i],
        arrival[i],
        burst[i],
        completion[i],
        turnaround[i],
        waiting[i]);
```

```
    }

    printf("\nAverage Turnaround Time: %.2f", avgTAT);
    printf("\nAverage Waiting Time: %.2f", avgWT);
    printf("\nThroughput: %.2f\n", throughput);

    return 0;
}
```

OUTPUT:

```
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ nano srtf.c
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ gcc srtf.c -o srtf
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ ./srtf
Enter the number of processes: 3

Enter details for process 1:
Arrival time: 0
Burst time: 3

Enter details for process 2:
Arrival time: 0
Burst time: 2

Enter details for process 3:
Arrival time: 0
Burst time: 5

PID     AT      BT      CT      TAT     WT
1       0       3       5       5       2
2       0       2       2       2       0
3       0       5       10      10      5

Average Turnaround Time: 5.67
Average Waiting Time: 2.33
Throughput: 0.30
```

4.Priority Scheduling (Non-Preemptive):

Once a process starts executing, it runs to completion, even if a higher-priority process arrives.

Program

```
#include <stdio.h>


struct Process {
   int id;
   int arrival;
   int burst;
   int priority;
   int waiting;
   int turnAround;
};


void sortProcesses(struct Process p[], int n) {
   struct Process temp;


   for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
         if (p[i].arrival > p[j].arrival || (p[i].arrival == p[j].arrival && p[i].priority > p[j].priority)) {
            temp = p[i];
            p[i] = p[j];
            p[j] = temp;  } } }}


void calculateTimes(struct Process p[], int n, int *totalTime) {
   int completed = 0, currentTime = 0;
   int processCount = n;
   int isCompleted[n];
   for (int i = 0; i < n; i++) isCompleted[i] = 0;


   while (completed < processCount) {
      int idx = -1;
```

```
        int highestPriority = 9999;

        for (int i = 0; i < n; i++) {
            if (p[i].arrival <= currentTime && isCompleted[i] == 0 && p[i].priority < highestPriority) {
                highestPriority = p[i].priority;
                idx = i;
            }
        }
        if (idx != -1) {
            currentTime += p[idx].burst;
            p[idx].turnAround = currentTime - p[idx].arrival;
            p[idx].waiting = p[idx].turnAround - p[idx].burst;
            isCompleted[idx] = 1;
            completed++;
        } else {
            currentTime++;
        }
    }
    *totalTime = currentTime;
}
void displayResults(struct Process p[], int n, int totalTime) {
    float totalWaiting = 0, totalTurnaround = 0;
    float throughput = (float)n / totalTime;
    printf("PID\tArrival\tBurst\tPriority\tWaiting\tTurnaround\n");
    for (int i = 0; i < n; i++) {
        totalWaiting += p[i].waiting;
        totalTurnaround += p[i].turnAround;
        printf("%d\t%d\t%d\t%d\t\t%d\t%d\n", p[i].id, p[i].arrival, p[i].burst, p[i].priority, p[i].waiting, p[i].turnAround);
    }

    printf("\nAverage Waiting Time: %.2f\n", totalWaiting / n);
```

```
    printf("Average Turnaround Time: %.2f\n", totalTurnaround / n);
    printf("Throughput: %.2f\n", throughput);
}

int main() {
    int n, totalTime;

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

    struct Process p[n];

    for (int i = 0; i < n; i++) {
        printf("\nEnter details for process %d:\n", i + 1);
        p[i].id = i + 1;
        printf("Arrival time: ");
        scanf("%d", &p[i].arrival);
        printf("Burst time: ");
        scanf("%d", &p[i].burst);
        printf("Priority: ");
        scanf("%d", &p[i].priority);
    }
    sortProcesses(p, n);
    calculateTimes(p, n, &totalTime);
    displayResults(p, n, totalTime);

    return 0;
}
```

OUTPUT:

```
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ nano priority.c
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ gcc priority.c -o prior
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ ./prior
Enter the number of processes: 3

Enter details for process 1:
Arrival time: 0
Burst time: 4
Priority: 0

Enter details for process 2:
Arrival time: 2
Burst time: 1
Priority: 3

Enter details for process 3:
Arrival time: 0
Burst time: 2
Priority: 1
PID     Arrival Burst   Priority        Waiting Turnaround
1       0       4       0               0       4
3       0       2       1               4       6
2       2       1       3               4       5

Average Waiting Time: 2.67
Average Turnaround Time: 5.00
Throughput: 0.43
```

5. Priority Scheduling (Preemptive):

Priority scheduling is one of the most common scheduling algorithms used by the operating system to schedule processes based on their priority. Each process is assigned a priority. The process with the highest priority is to be executed first and so on.

Preemptive Priority Scheduling: The process with the highest priority (lowest number) can preempt another running process.

Program:

```
#include <stdio.h>


struct Process {
    int id;
    int arrival;
    int burst;
    int priority;
    int remaining;
    int waiting;
    int turnAround;
};


void sortProcesses(struct Process p[], int n) {
    struct Process temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (p[i].priority > p[j].priority) {
                temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }
}


void calculateTimes(struct Process p[], int n) {
    int totalTime = 0, completed = 0, currentTime = 0;
```

```
    int processCount = n;

    while (completed < processCount) {
        int idx = -1;
        int highestPriority = 9999;

        for (int i = 0; i < n; i++) {
            if (p[i].arrival <= currentTime && p[i].remaining > 0 && p[i].priority <
highestPriority) {
                highestPriority = p[i].priority;
                idx = i;
            }
        }

        if (idx != -1) {
            p[idx].remaining--;
            currentTime++;

            if (p[idx].remaining == 0) {
                completed++;
                p[idx].turnAround = currentTime - p[idx].arrival;
                p[idx].waiting = p[idx].turnAround - p[idx].burst;
            }
        } else {
            currentTime++;
        }
    }
}

void displayResults(struct Process p[], int n) {
    float totalWaiting = 0, totalTurnaround = 0;
    int completionTime = 0;
```

```
    printf("PID\tArrival\tBurst\tPriority\tWaiting\tTurnaround\n");
    for (int i = 0; i < n; i++) {
        totalWaiting += p[i].waiting;
        totalTurnaround += p[i].turnAround;
        if (p[i].turnAround + p[i].arrival > completionTime) {
            completionTime = p[i].turnAround + p[i].arrival;
        }
        printf("%d\t%d\t%d\t%d\t\t%d\t%d\n", p[i].id, p[i].arrival, p[i].burst, p[i].priority,
p[i].waiting, p[i].turnAround);
    }

    printf("\nAverage Waiting Time: %.2f\n", totalWaiting / n);
    printf("Average Turnaround Time: %.2f\n", totalTurnaround / n);
    printf("Throughput: %.2f processes/unit time\n", (float)n / completionTime);
}

int main() {
    int n;

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

    struct Process p[n];

    for (int i = 0; i < n; i++) {
        printf("\nEnter details for process %d:\n", i + 1);
        p[i].id = i + 1;
        printf("Arrival time: ");
        scanf("%d", &p[i].arrival);
        printf("Burst time: ");
        scanf("%d", &p[i].burst);
        printf("Priority: ");
        scanf("%d", &p[i].priority);
```

```
        p[i].remaining = p[i].burst;
    }

    sortProcesses(p, n);
    calculateTimes(p, n);
    displayResults(p, n);
    return 0;
}
```

OUTPUT:

```
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ nano prePrior.c
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ gcc prePrior.c -o preP
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ ./preP
Enter the number of processes: 3

Enter details for process 1:
Arrival time: 0
Burst time: 5
Priority: 4

Enter details for process 2:
Arrival time: 2
Burst time: 3
Priority: 1

Enter details for process 3:
Arrival time: 0
Burst time: 3
Priority: 1
PID     Arrival Burst   Priority        Waiting Turnaround
2       2       3       1               0       3
3       0       3       1               3       6
1       0       5       4               6       11

Average Waiting Time: 3.00
Average Turnaround Time: 6.67
Throughput: 0.27 processes/unit time
```

6. Round Robin Scheduling:

Round robin is a CPU scheduling algorithm where each process is cyclically assigned a fixed time slot. It is the preemptive version of the First come First Serve CPU Scheduling algorithm.

Program

```
#include <stdio.h>


void round_robin(int n, int arrival[], int burst[], int processID[], int quantum) {
   int remaining[n], completion[n], waiting[n], turnaround[n];
   int time = 0, completed = 0;

   for (int i = 0; i < n; i++) {
      remaining[i] = burst[i];
   }

   while (completed < n) {
      for (int i = 0; i < n; i++) {
         if (arrival[i] <= time && remaining[i] > 0) {
            int execTime = (remaining[i] > quantum) ? quantum : remaining[i];
            remaining[i] -= execTime;
            time += execTime;

            if (remaining[i] == 0) {
               completed++;
               completion[i] = time;
               turnaround[i] = completion[i] - arrival[i];
               waiting[i] = turnaround[i] - burst[i];
            }
         }
      }
   }

   float avgWaitingTime = 0, avgTurnaroundTime = 0;
   for (int i = 0; i < n; i++) {
```

```
        avgWaitingTime += waiting[i];
        avgTurnaroundTime += turnaround[i];
    }

    printf("\nRound Robin Scheduling (Quantum = %d):\n", quantum);
    printf("Process ID\tArrival Time\tBurst Time\tCompletion Time\tTurnaround
Time\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n",
            processID[i], arrival[i], burst[i], completion[i],
            turnaround[i], waiting[i]);
    }
    printf("Average Waiting Time: %.2f\n", avgWaitingTime / n);
    printf("Average Turnaround Time: %.2f\n", avgTurnaroundTime / n);
    printf("Throughput: %.2f processes per unit time\n", (float)n / completion[n-1]);
}
int main() {
    int n, quantum;
    printf("Enter the number of processes: ");
    scanf("%d", &n);

    int arrival[n], burst[n], processID[n];
    for (int i = 0; i < n; i++) {
        processID[i] = i + 1;
        printf("\nEnter details for process %d:\n", processID[i]);
        printf("Arrival time: ");
        scanf("%d", &arrival[i]);
        printf("Burst time: ");
        scanf("%d", &burst[i]);
    }

    printf("Enter the time quantum: ");
    scanf("%d", &quantum);
```

```
    round_robin(n, arrival, burst, processID, quantum);

    return 0;
}
```

OUTPUT:

```
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ nano RR.c
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ gcc RR.c -o rr
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ ./rr
Enter the number of processes: 4

Enter details for process 1:
Arrival time: 0
Burst time: 5

Enter details for process 2:
Arrival time: 1
Burst time: 4

Enter details for process 3:
Arrival time: 1
Burst time: 2

Enter details for process 4:
Arrival time: 3
Burst time: 18
Enter the time quantum: 2

Round Robin Scheduling (Quantum = 2):
Process ID      Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time
1               0               5               15              15              10
2               1               4               12              11              7
3               1               2               6               5               3
4               3               18              29              26              8
Average Waiting Time: 7.00
Average Turnaround Time: 14.25
Throughput: 0.14 processes per unit time
```

7.Multilevel Queue Scheduling (MQS) algorithm

The Multilevel Queue Scheduling (MQS) algorithm is a scheduling algorithm where processes are divided into multiple queues, and each queue has its own scheduling algorithm. The system uses different queues based on process characteristics, such as priority, process type, or other factors. Each queue has a distinct priority, and processes within a queue are scheduled according to the algorithm defined for that queue.

A typical implementation involves the following steps:

1. Multiple Queues: Processes are classified into multiple queues based on some criteria, such as:

- High priority
- Interactive processes
- Batch processes, etc.

2. Scheduling for Each Queue: Each queue uses a different scheduling algorithm. For example:

- Round Robin for interactive processes (high priority)
- First Come First Serve (FCFS) for batch processes (low priority)

3. Queue Priority: The queues themselves are prioritized, with higher-priority queues being executed first.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
struct process {
   int priority;
   int burst_time;
   int tt_time;
   int total_time;
};
struct queues {
   int priority_start;
   int priority_end;
   int total_time;
   int length;
   struct process *p;
   bool executed;
```

```
};
bool notComplete(struct queues q[]) {
    bool a = false;
    int countInc = 0;
    for (int i = 0; i < 3; i++) {
        countInc = 0;
        for (int j = 0; j < q[i].length; j++) {
            if (q[i].p[j].burst_time != 0) {
                a = true;
            } else {
                countInc += 1;
            }
        }
        if (countInc == q[i].length) {
            q[i].executed = true;
        }
    }
    return a;
}
void sort_ps(struct queues *q) {

    for (int i = 1; i < q->length; i++) {
        for (int j = 0; j < q->length - 1; j++) {
            if (q->p[j].priority < q->p[j + 1].priority) {
                struct process temp = q->p[j + 1];
                q->p[j + 1] = q->p[j];
                q->p[j] = temp;
            }
        }
    }
}
void checkCompleteTimer(struct queues q[]) {
    bool a = notComplete(q);
```

```
    for (int i = 0; i < 3; i++) {
        if (q[i].executed == false) {
            for (int j = 0; j < q[i].length; j++) {
                if (q[i].p[j].burst_time != 0) {
                    q[i].p[j].total_time += 1;
                }
            }
            q[i].total_time += 1;
        }
    }
}
int main() {
    struct queues q[3];
    q[0].priority_start = 7;
    q[0].priority_end = 9;
    q[1].priority_start = 4;
    q[1].priority_end = 6;
    q[2].priority_start = 1;
    q[2].priority_end = 3;
    int no_of_processes, priority_of_process, burst_time_of_process;
    printf("Enter the number of processes::\n");
    scanf("%d", &no_of_processes);
    struct process p1[no_of_processes];
    for (int i = 0; i < no_of_processes; i++) {
        printf("Enter the priority of the process %d::\n",i+1);
        scanf("%d", &priority_of_process);
        printf("Enter the burst time of the process %d::\n",i+1);
        scanf("%d", &burst_time_of_process);
        p1[i].priority = priority_of_process;
        p1[i].burst_time = burst_time_of_process;
        p1[i].tt_time = burst_time_of_process;
        p1[i].total_time = 0;
        for (int j = 0; j < 3; j++) {
```

```
            if (q[j].priority_start <= priority_of_process && priority_of_process <=
q[j].priority_end) {
                q[j].length++;
            }
        }
    }
    for (int i = 0; i < 3; i++) {
        int len = q[i].length;
        q[i].p = (struct process *)malloc(len * sizeof(struct process));
        q[i].executed = false;
        q[i].total_time = 0;
    }
    int a = 0, b = 0, c = 0;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < no_of_processes; j++) {
            if ((q[i].priority_start <= p1[j].priority) && (p1[j].priority <= q[i].priority_end)) {
                if (i == 0) {
                    q[i].p[a++] = p1[j];
                } else if (i == 1) {
                    q[i].p[b++] = p1[j];
                } else {
                    q[i].p[c++] = p1[j];
                }
            }
        }
    }
    for (int i = 0; i < 3; i++) {
        printf("Queue %d : \t", i + 1);
        for (int j = 0; j < q[i].length; j++) {
            printf("%d->", q[i].p[j].priority);
        }
        printf("NULL\n");
    }
```

```
int timer = 0, l = -1, rr_timer = 4;
int counter = 0, counterps = 0, counterfcfs = 0;
while (notComplete(q)) {
    if (timer == 10) {
        timer = 0;
    }
    l++;
    if (l >= 3) {
        l = l % 3;
    }
    if (q[l].executed == true) {
        printf("Queue %d completed\n", l + 1);
        l++;
        if (l >= 3) {
            l = l % 3;
        }
        continue;
    }
    if (l == 0) {
        printf("Queue %d in hand\n", l + 1);
        if (rr_timer == 0) {
            rr_timer = 4;
        }
        for (int i = 0; i < q[l].length; i++) {
            if (q[l].p[i].burst_time == 0) {
                counter++;
                continue;
            }
            if (counter == q[l].length) {
                break;
            }
            while (rr_timer > 0 && q[l].p[i].burst_time != 0 && timer != 10) {
```

```
                printf("Executing queue 1 and %d process for a unit time. Process has priority of %d\n", i + 1, q[l].p[i].priority);
                q[l].p[i].burst_time--;
                checkCompleteTimer(q);
                rr_timer--;
                timer++;
            }
            if (timer == 10) {
                break;
            }
            if (q[l].p[i].burst_time == 0 && rr_timer == 0) {
                rr_timer = 4;
                if (i == (q[i].length - 1)) {
                    i = -1;
                }
                continue;
            }
            if (q[l].p[i].burst_time == 0 && rr_timer > 0) {
                if (i == (q[i].length - 1)) {
                    i = -1;
                }
                continue;
            }
            if (rr_timer <= 0) {
                rr_timer = 4;
                if (i == (q[i].length - 1)) {
                    i = -1;
                }
                continue;
            }
        }
    }
    else if (l == 1) {
```

```
            printf("Queue %d in hand\n", l + 1);
            sort_ps(&q[l]);
            for (int i = 0; i < q[l].length; i++) {
                if (q[l].p[i].burst_time == 0) {
                    counterps++;
                    continue;
                }
                if (counterps == q[l].length) {
                    break;
                }
                while (q[l].p[i].burst_time != 0 && timer != 10) {
                    printf("Executing queue 2 and %d process for a unit time. Process has priority of
%d\n", i + 1, q[l].p[i].priority);
                    q[l].p[i].burst_time--;
                    checkCompleteTimer(q);
                    timer++;
                }
                if (timer == 10) {
                    break;
                }
                if (q[l].p[i].burst_time == 0) {
                    continue;
                }
            }
        } else {
            printf("Queue %d in hand\n", l + 1);
            for (int i = 0; i < q[l].length; i++) {
                if (q[l].p[i].burst_time == 0) {
                    counterfcfs++;
                    continue;
                }
                if (counterfcfs == q[l].length) {
                    break;
```

```
            }
            while (q[l].p[i].burst_time != 0 && timer != 10) {
                printf("Executing queue 3 and %d process for a unit time. Process has priority of %d\n", i + 1, q[l].p[i].priority);
                q[l].p[i].burst_time--;
                checkCompleteTimer(q);
                timer++;
            }
            if (timer == 10) {
                break;
            }
            if (q[l].p[i].burst_time == 0) {
                continue;
            }
        }
    }
    printf("Broke from queue %d\n", l + 1);
}
for (int i = 0; i < 3; i++) {
    printf("\nTime taken for queue %d to execute: %d\n", i + 1, q[i].total_time);
    printf("Turnaround times for the queue are: \n");
    for (int j = 0; j < q[i].length; j++) {
        printf("Process %d's turnaround time = %d\n", j + 1, q[i].p[j].total_time);
    }
}
return 0;
```

OUTPUT:

```
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ nano Q.c
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ gcc Q.c -o queue
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ ./queue
Enter the number of processes::
3
Enter the priority of the process 1::
0
Enter the burst time of the process 1::
5
Enter the priority of the process 2::
2
Enter the burst time of the process 2::
10
Enter the priority of the process 3::
1
Enter the burst time of the process 3::
5
Queue 1 :       NULL
Queue 2 :       NULL
Queue 3 :       2->1->NULL
Queue 1 completed
Queue 3 in hand
Executing queue 3 and 1 process for a unit time. Process has priority of 2
Executing queue 3 and 1 process for a unit time. Process has priority of 2
Executing queue 3 and 1 process for a unit time. Process has priority of 2
Executing queue 3 and 1 process for a unit time. Process has priority of 2
Executing queue 3 and 1 process for a unit time. Process has priority of 2
Executing queue 3 and 1 process for a unit time. Process has priority of 2
Executing queue 3 and 1 process for a unit time. Process has priority of 2
Executing queue 3 and 1 process for a unit time. Process has priority of 2
Executing queue 3 and 1 process for a unit time. Process has priority of 2
Executing queue 3 and 1 process for a unit time. Process has priority of 2
Broke from queue 3
Queue 1 completed
Queue 3 in hand
Executing queue 3 and 2 process for a unit time. Process has priority of 1
Executing queue 3 and 2 process for a unit time. Process has priority of 1
Executing queue 3 and 2 process for a unit time. Process has priority of 1
Executing queue 3 and 2 process for a unit time. Process has priority of 1
Executing queue 3 and 2 process for a unit time. Process has priority of 1
Broke from queue 3

Time taken for queue 1 to execute: 0
Turnaround times for the queue are:

Time taken for queue 2 to execute: 0
Turnaround times for the queue are:

Time taken for queue 3 to execute: 14
Turnaround times for the queue are:
Process 1's turnaround time = 9
Process 2's turnaround time = 14
```

4.Implement file storage allocation techniques: Contiguous allocation (using array), Linked-list allocation (using linked list), and indirect allocation (using indexing).

1.Contiguous Memory Allocation-

Contiguous memory allocation involves assigning a single, continuous block of memory to store data. This method promotes faster processing, as memory is laid out sequentially, making access more efficient. However, when memory is dynamically allocated and freed, it can result in fragmentation, where finding a large enough block of free memory becomes challenging over time due to scattered gaps between allocated spaces.

Program

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

#define MEMORY_SIZE 20

char memory[MEMORY_SIZE];

void initMemory() {
   for (int i = 0; i < MEMORY_SIZE; i++) {
      memory[i] = '\0';
   }
}

int storeString(int startIndex, const char* str) {
   int length = strlen(str);
   if (startIndex + length > MEMORY_SIZE) {
      printf("Error: Not enough space in memory to store the string.\n");
      return -1;
   }
   for (int i = 0; i < length; i++) {
      memory[startIndex + i] = str[i];
   }
   printf("String '%s' stored at index %d.\n", str, startIndex);
   return 0;
}

void retrieveString(int startIndex, int length) {
   if (startIndex + length > MEMORY_SIZE) {
      printf("Error: Retrieval goes out of memory bounds.\n");
      return;
   }
   char retrievedStr[MEMORY_SIZE];
   for (int i = 0; i < length; i++) {
      retrievedStr[i] = memory[startIndex + i];
   }
   retrievedStr[length] = '\0';
   printf("Retrieved string from index %d: '%s'\n", startIndex, retrievedStr);
}

void displayMemory() {
   printf("Current memory state: ");
   for (int i = 0; i < MEMORY_SIZE; i++) {
      if (memory[i] != '\0') {
```

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

int main() {
    initMemory();
    storeString(0, "Aseem");
    storeString(6, "Nihal");
    retrieveString(0, 5);
    retrieveString(6, 5);
    displayMemory();
    return 0;
}
```

OUTPUT:

```
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ nano ContMemoryAlloc
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ nano ContMemoryAlloc.c
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ gcc ContMemoryAlloc.c -o ContMemoryAlloc
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ ./ContMemoryAlloc
String 'Shrushranto' stored at index 0.
String 'Rajbongshi' stored at index 6.
Retrieved string from index 0: 'Shru'
Retrieved string from index 0: 'ShrushRajb'
Current memory state: ShrushRajbongshi____
```

2.Linked List-

In contrast, linked list allocation assigns memory dynamically in individual blocks, known as nodes, each containing a reference to the next one. This approach avoids fragmentation and offers more flexibility, as new elements can be inserted without the need to move existing data. However, it comes with added overhead since extra memory is required for storing pointers, and access times tend to be slower compared to contiguous memory allocation.

Program

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

struct Node {
    char c;
    struct Node* next;
};

struct Node* create_node(char value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    newNode->c = value;
    newNode->next = NULL;
    return newNode;
}

struct Node* linked_list() {
    struct Node* head = NULL;
    struct Node* temp = NULL;
    int n;
    char value;

    printf("Enter the number of characters for this string: ");
    scanf("%d", &n);
    getchar();

    printf("Enter the characters for the string: ");
    for (int i = 0; i < n; i++) {
        scanf(" %c", &value);
        struct Node* newNode = create_node(value);
        if (head == NULL) {
            head = newNode;
            temp = newNode;
        } else {
            temp->next = newNode;
            temp = newNode;
        }
    }
    return head;
}

void traverse(struct Node* head) {
    if (head == NULL) {
        printf("Empty\n");
        return;
    }
```

```
    while (head != NULL) {
        printf("%c ", head->c);
        head = head->next;
    }
    printf("\n");
}

void free_list(struct Node* head) {
    struct Node* temp;
    while (head != NULL) {
        temp = head;
        head = head->next;
        free(temp);
    }
}

int main() {
    int num_strings;
    struct Node** linkedLists;

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

    linkedLists = (struct Node**)malloc(num_strings * sizeof(struct Node*));

    if (linkedLists == NULL) {
        printf("Memory allocation failed for linked lists array!\n");
        exit(1);
    }

    for (int i = 0; i < num_strings; i++) {
        printf("\nCreating Linked List for String %d:\n", i + 1);
        linkedLists[i] = linked_list();
    }

    for (int i = 0; i < num_strings; i++) {
        printf("\nLinked List %d: ", i + 1);
        traverse(linkedLists[i]);
    }

    for (int i = 0; i < num_strings; i++) {
        free_list(linkedLists[i]);
    }

    free(linkedLists);

    return 0;
}
```

OUTPUT:

```
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ nano LLAlloc.c
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ gcc LLAlloc.c -o llalloc
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ ./llalloc
Enter the number of strings: 3

Creating Linked List for String 1:
Enter the number of characters for this string: 2
Enter the characters for the string: r
t

Creating Linked List for String 2:
Enter the number of characters for this string: 4
Enter the characters for the string: S H R U

Creating Linked List for String 3:
Enter the number of characters for this string: 4
Enter the characters for the string: R A J B

Linked List 1: r t

Linked List 2: S H R U

Linked List 3: R A J B
```

3.Indirect Allocation-

Indirect allocation uses pointers to refer to memory locations rather than assigning memory directly. This method allows for dynamic access to data structures and function calls without being tied to a specific memory location. It offers flexibility and efficient memory management, making it ideal for complex data structures and function pointers, though it introduces an added layer of complexity during program execution.

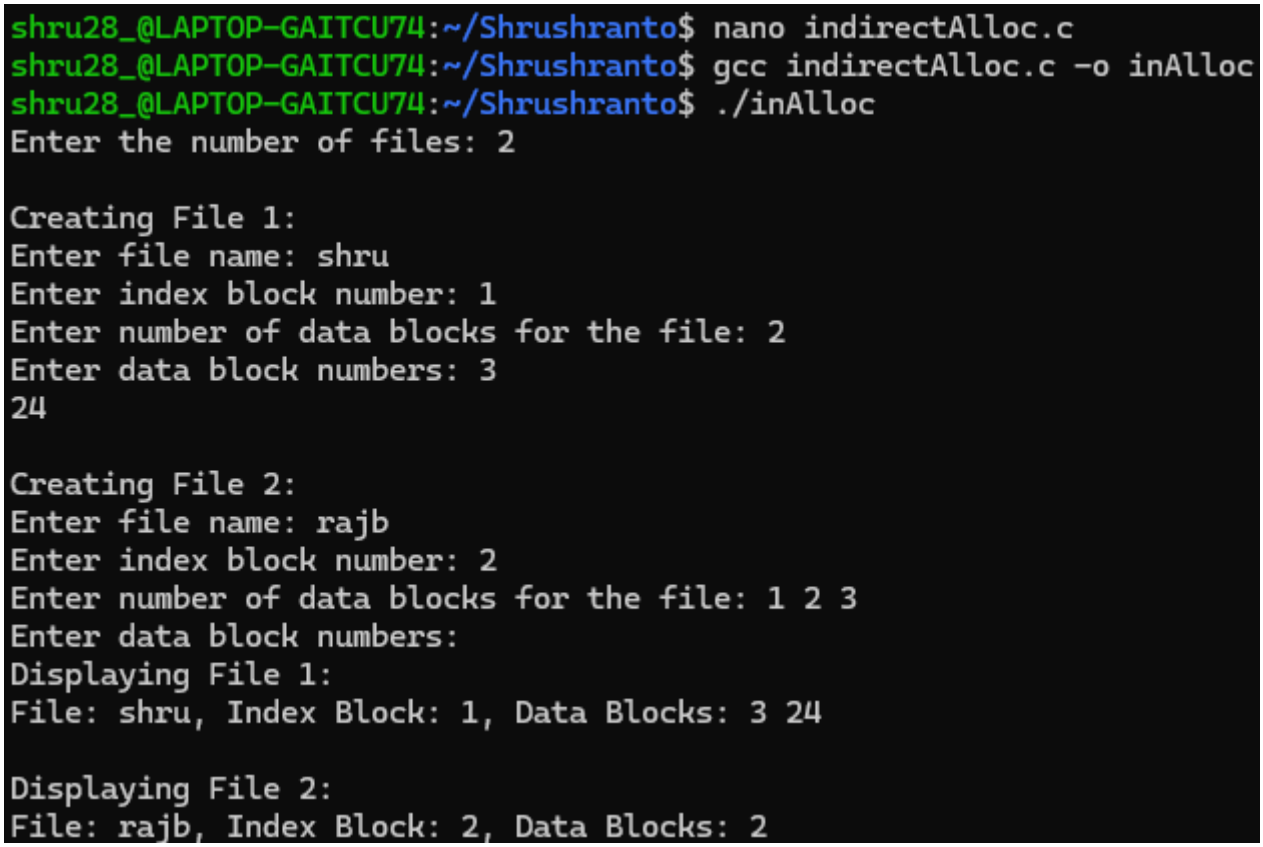
Program

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

struct Block {
    int blockNumber;
    struct Block* next;
};

struct File {
    char name[20];
    int indexBlock;
    struct Block* dataBlocks;
};

struct Block* create_block(int blockNumber) {
    struct Block* newBlock = (struct Block*)malloc(sizeof(struct Block));
    if (newBlock == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    newBlock->blockNumber = blockNumber;
    newBlock->next = NULL;
    return newBlock;
}

struct File* create_file() {
    struct File* file = (struct File*)malloc(sizeof(struct File));
    if (file == NULL) {
        printf("Memory allocation failed for file!\n");
        exit(1);
    }

    printf("Enter file name: ");
    scanf("%s", file->name);

    printf("Enter index block number: ");
    scanf("%d", &file->indexBlock);

    int n, blockNumber;
    printf("Enter number of data blocks for the file: ");
    scanf("%d", &n);

    struct Block* head = NULL;
    struct Block* temp = NULL;

    printf("Enter data block numbers: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &blockNumber);
        struct Block* newBlock = create_block(blockNumber);
```

```
        if (head == NULL) {
            head = newBlock;
            temp = newBlock;
        } else {
            temp->next = newBlock;
            temp = newBlock;
        }
    }

    file->dataBlocks = head;
    return file;
}

void display_file(struct File* file) {
    if (file == NULL) {
        printf("File not found!\n");
        return;
    }
    printf("File: %s, Index Block: %d, Data Blocks: ", file->name, file->indexBlock);
    struct Block* temp = file->dataBlocks;
    while (temp != NULL) {
        printf("%d ", temp->blockNumber);
        temp = temp->next;
    }
    printf("\n");
}

void free_file(struct File* file) {
    struct Block* temp;
    while (file->dataBlocks != NULL) {
        temp = file->dataBlocks;
        file->dataBlocks = file->dataBlocks->next;
        free(temp);
    }
    free(file);
}

int main() {
    int num_files;
    struct File** files;

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

    files = (struct File**)malloc(num_files * sizeof(struct File*));
    if (files == NULL) {
        printf("Memory allocation failed for files array!\n");
        exit(1);
    }

    for (int i = 0; i < num_files; i++) {
        printf("\nCreating File %d:\n", i + 1);
        files[i] = create_file();
    }

    for (int i = 0; i < num_files; i++) {
        printf("\nDisplaying File %d:\n", i + 1);
```

```
        display_file(files[i]);
    }

    for (int i = 0; i < num_files; i++) {
        free_file(files[i]);
    }

    free(files);
    return 0;
}
```

OUTPUT:



5.Implement File Directories: Single Level, Two Level, Tree Level, Acyclic Graph Directory.

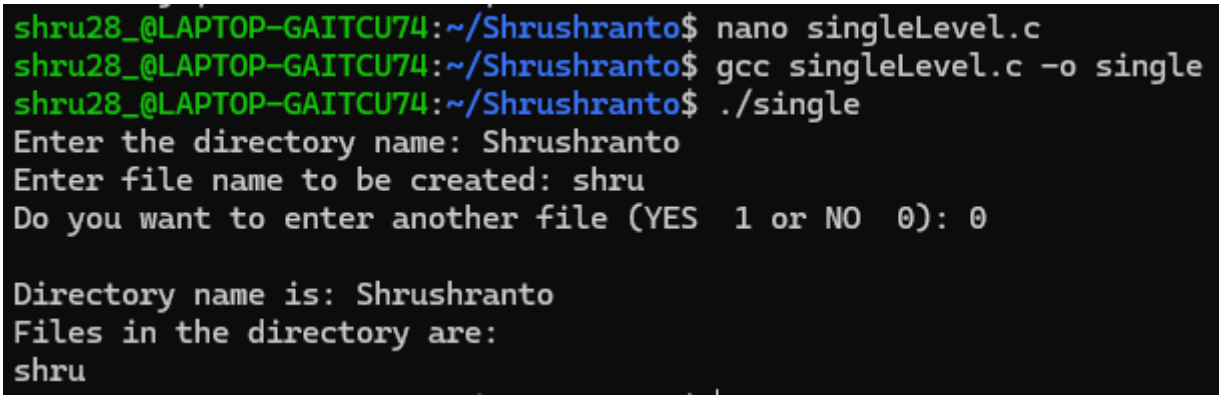
1.Single Level:

A **Single-Level Directory** is the simplest form of directory structure used in operating systems to organize and manage files. In this structure, all files are stored in a single directory, and there is no hierarchy or subdirectory organization.

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

#define MAX_FILENAME_LEN 100

void main() {
   int nf = 0, i = 0, ch;
   char mdname[MAX_FILENAME_LEN], name[MAX_FILENAME_LEN];

   char **fname = NULL;

   printf("Enter the directory name: ");
   scanf("%s", mdname);

   do {
      printf("Enter file name to be created: ");
      scanf("%s", name);

      int fileExists = 0;
      for (i = 0; i < nf; i++) {
         if (strcmp(name, fname[i]) == 0) {
            printf("There is already a file named %s\n", name);
            fileExists = 1;
            break;
         }
      }

      if (!fileExists) {
```

```
            fname = realloc(fname, (nf + 1) * sizeof(char *));
            if (fname == NULL) {
                printf("Memory allocation failed\n");
                return;
            }

            fname[nf] = malloc((strlen(name) + 1) * sizeof(char));
            if (fname[nf] == NULL) {
                printf("Memory allocation failed\n");
                return;
            }
            strcpy(fname[nf], name);
            nf++;
        }
        printf("Do you want to enter another file (YES  1 or NO  0): ");
        scanf("%d", &ch);

    } while (ch == 1);

    printf("\nDirectory name is: %s\n", mdname);
    printf("Files in the directory are:\n");
    for (i = 0; i < nf; i++) {
        printf("%s\n", fname[i]);
        free(fname[i]);
    }
    free(fname);
}
```

OUTPUT:


shru28_@LAPTOP-GAITCU74:~/Shrushranto$ nano singleLevel.c
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ gcc singleLevel.c -o single
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ ./single
Enter the directory name: Shrushranto
Enter file name to be created: shru
Do you want to enter another file (YES 1 or NO 0): 0
Directory name is: Shrushranto
Files in the directory are:
shru

2.Two Level:

A **Two-Level Directory** structure is an improvement over the **Single-Level Directory** that introduces **user-specific directories** to solve problems like name conflicts and file organization.

Instead of a single directory for all users, the root directory contains separate directories for each user.

Each user has a personal directory where they can store their files.

Users cannot access files from other users' directories unless permissions allow it.

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

#define MAX_NAME_LENGTH 100
#define MAX_SUBDIRS 10
#define MAX_FILES 10

struct st {
   char dname[MAX_NAME_LENGTH];
   char sdname[MAX_SUBDIRS][MAX_NAME_LENGTH];
   char fname[MAX_SUBDIRS][MAX_FILES][MAX_NAME_LENGTH];
   int ds;
   int sds[MAX_SUBDIRS];
} dir[MAX_SUBDIRS];

int main() {
   int i, j, k, n;

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

   for (i = 0; i < n; i++) {
      printf("Enter directory %d name: ", i + 1);
      scanf("%9s", dir[i].dname);

      printf("Enter size of directory %d (number of subdirectories): ", i + 1);
      scanf("%d", &dir[i].ds);
```

```
        for (j = 0; j < dir[i].ds; j++) {
            printf("Enter subdirectory %d name: ", j + 1);
            scanf("%9s", dir[i].sdname[j]);

            printf("Enter number of files in subdirectory %s: ", dir[i].sdname[j]);
            scanf("%d", &dir[i].sds[j]);

            for (k = 0; k < dir[i].sds[j]; k++) {
                printf("Enter file name for subdirectory %s: ", dir[i].sdname[j]);
                scanf("%9s", dir[i].fname[j][k]);
            }
        }
    }

    printf("\nDirectory Structure:\n");
    printf("Directory Name\tSubdirectory Name\tNo. of Files\tFile Names\n");
    printf("**********************************************************\n");

    for (i = 0; i < n; i++) {
        printf("%s\t", dir[i].dname);
        for (j = 0; j < dir[i].ds; j++) {
            printf("%s\t\t", dir[i].sdname[j]);
            printf("%d\t", dir[i].sds[j]);
            for (k = 0; k < dir[i].sds[j]; k++) {
                printf("%s\t", dir[i].fname[j][k]);
            }
            printf("\n\t\t");
        }
        printf("\n");
    }

    return 0;
}
```

OUTPUT:

```
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ ./two
Enter number of directories: 2

Enter name for directory 1: Shrushranto
Enter number of subdirectories in Shrushranto: 0

Enter name for directory 2: OsLabs
Enter number of subdirectories in OsLabs: 0


========== Directory Structure ==========

Directory: Shrushranto

Directory: OsLabs
```

3.Tree Level:

A **Tree-Level Directory Structure** extends the **Two-Level Directory** by introducing **hierarchical organization** through subdirectories. This is the most commonly used directory structure in modern operating systems. The system starts with a root directory (/).Each user has a home directory under the root. Inside each user's directory, subdirectories can be created for better organization.

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

#define MAX_DIRS 3
#define MAX_SUBDIRS 3
#define MAX_FILES 3
#define MAX_NAME_LENGTH 100

typedef struct {
    char filename[MAX_NAME_LENGTH];
} File;

typedef struct {
    char subdirName[MAX_NAME_LENGTH];
    File files[MAX_FILES];
    int fileCount;
} Subdirectory;

typedef struct {
    char dirName[MAX_NAME_LENGTH];
    Subdirectory subdirs[MAX_SUBDIRS];
    int subdirCount;
} Directory;

typedef struct {
    Directory directories[MAX_DIRS];
    int dirCount;
} MainDirectory;
```

```
void addDirectory(MainDirectory *dir, const char *dirName) {
    if (dir->dirCount < MAX_DIRS) {
        strcpy(dir->directories[dir->dirCount].dirName, dirName);
        dir->directories[dir->dirCount].subdirCount = 0;
        dir->dirCount++;
        printf("Directory '%s' added.\n", dirName);
    } else {
        printf("Max directory limit reached.\n");
    }
}

void addSubdirectory(Directory *dir, const char *subdirName) {
    if (dir->subdirCount < MAX_SUBDIRS) {
        strcpy(dir->subdirs[dir->subdirCount].subdirName, subdirName);
        dir->subdirs[dir->subdirCount].fileCount = 0;
        dir->subdirCount++;
        printf("Subdirectory '%s' added to '%s'.\n", subdirName, dir->dirName);
    } else {
        printf("Max subdirectory limit reached in '%s'.\n", dir->dirName);
    }
}

void addFileToSubdirectory(Directory *dir, const char *subdirName, const char *filename) {
    for (int i = 0; i < dir->subdirCount; i++) {
        if (strcmp(dir->subdirs[i].subdirName, subdirName) == 0) {
            if (dir->subdirs[i].fileCount < MAX_FILES) {
                strcpy(dir->subdirs[i].files[dir->subdirs[i].fileCount].filename, filename);
                dir->subdirs[i].fileCount++;
                printf("File '%s' added to subdirectory '%s'.\n", filename, subdirName);
            } else {
                printf("Max file limit reached in subdirectory '%s'.\n", subdirName);
            }
            return;
        }
```

```
    }
    printf("Subdirectory '%s' not found.\n", subdirName);
}

void listFilesInSubdirectory(Directory *dir, const char *subdirName) {
    for (int i = 0; i < dir->subdirCount; i++) {
        if (strcmp(dir->subdirs[i].subdirName, subdirName) == 0) {
            printf("Files in subdirectory '%s':\n", subdirName);
            for (int j = 0; j < dir->subdirs[i].fileCount; j++) {
                printf("%d. %s\n", j + 1, dir->subdirs[i].files[j].filename);
            }
            return;
        }
    }
    printf("Subdirectory '%s' not found.\n", subdirName);
}

int main() {
    MainDirectory mainDir = {0};
    int choice;
    char dirName[MAX_NAME_LENGTH], subdirName[MAX_NAME_LENGTH],
filename[MAX_NAME_LENGTH];

    while (1) {
        printf("\n1. Add Directory\n2. Add Subdirectory\n3. Add File to Subdirectory\n4. List Files in
Subdirectory\n5. Exit\nEnter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter directory name: ");
                scanf("%s", dirName);
                addDirectory(&mainDir, dirName);
                break;
            case 2:
```

```
            printf("Enter directory name to add subdirectory to: ");
            scanf("%s", dirName);
            for (int i = 0; i < mainDir.dirCount; i++) {
                if (strcmp(mainDir.directories[i].dirName, dirName) == 0) {
                    printf("Enter subdirectory name: ");
                    scanf("%s", subdirName);
                    addSubdirectory(&mainDir.directories[i], subdirName);
                    break;
                }
            }
            break;
        case 3:
            printf("Enter directory name to add file to a subdirectory: ");
            scanf("%s", dirName);
            printf("Enter subdirectory name: ");
            scanf("%s", subdirName);
            printf("Enter file name: ");
            scanf("%s", filename);
            for (int i = 0; i < mainDir.dirCount; i++) {
                if (strcmp(mainDir.directories[i].dirName, dirName) == 0) {
                    addFileToSubdirectory(&mainDir.directories[i], subdirName, filename);
                    break;
                }
            }
            break;
        case 4:
            printf("Enter directory name to list files in subdirectory: ");
            scanf("%s", dirName);
            printf("Enter subdirectory name: ");
            scanf("%s", subdirName);
            for (int i = 0; i < mainDir.dirCount; i++) {
                if (strcmp(mainDir.directories[i].dirName, dirName) == 0) {
                    listFilesInSubdirectory(&mainDir.directories[i], subdirName);
                    break;
```

```
            }
        }
        break;
    case 5:
        return 0;
    default:
        printf("Invalid choice.\n");
  }
 }
}
```

OUTPUT:

```
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ nano treeLevel.c
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ gcc treeLevel.c -o tree
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ ./tree

1. Add Directory
2. Add Subdirectory
3. Add File to Subdirectory
4. List Files in Subdirectory
5. Exit
Enter choice: 1
Enter directory name: shru
Directory 'shru' added.

1. Add Directory
2. Add Subdirectory
3. Add File to Subdirectory
4. List Files in Subdirectory
5. Exit
Enter choice: 2
Enter directory name to add subdirectory to: Shrushranto

1. Add Directory
2. Add Subdirectory
3. Add File to Subdirectory
4. List Files in Subdirectory
5. Exit
Enter choice: 1
Enter directory name: os
Directory 'os' added.

1. Add Directory
2. Add Subdirectory
3. Add File to Subdirectory
4. List Files in Subdirectory
5. Exit
Enter choice: 5
shru28_@LAPTOP-GAITCU74:~/Shrushranto$
```

4.Acyclic Graph Directory:

An **Acyclic Graph Directory** structure is an advanced file system organization that allows **files and directories to be shared among multiple users** while preventing **circular references** (cycles). It is an improvement over the **Tree-Level Directory**, allowing **directories and files to have multiple parents** using links.

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

#define MAX_NAME_LENGTH 100
#define MAX_NODES 100

typedef struct Node {
    char name[MAX_NAME_LENGTH];
    struct Node *links[MAX_NODES];
    int linkCount;
} Node;

typedef struct Graph {
    Node *nodes[MAX_NODES];
    int nodeCount;
} Graph;

Graph directoryGraph = {0};

Node *createNode(const char *name) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    if (!newNode) {
        printf("Memory allocation failed\n");
        return NULL;
    }
    strcpy(newNode->name, name);
    newNode->linkCount = 0;
    return newNode;
}
```

```
void addNode(Graph *graph, const char *name) {
    if (graph->nodeCount >= MAX_NODES) {
        printf("Max node limit reached.\n");
        return;
    }
    Node *newNode = createNode(name);
    graph->nodes[graph->nodeCount++] = newNode;
    printf("Node '%s' added.\n", name);
}

Node *findNode(Graph *graph, const char *name) {
    for (int i = 0; i < graph->nodeCount; i++) {
        if (strcmp(graph->nodes[i]->name, name) == 0) {
            return graph->nodes[i];
        }
    }
    return NULL;
}

void addLink(Graph *graph, const char *from, const char *to) {
    Node *fromNode = findNode(graph, from);
    Node *toNode = findNode(graph, to);

    if (!fromNode || !toNode) {
        printf("One or both nodes not found.\n");
        return;
    }

    for (int i = 0; i < toNode->linkCount; i++) {
        if (toNode->links[i] == fromNode) {
            printf("Cycle detected! Cannot add link from '%s' to '%s'.\n", from, to);
            return;
        }
```

```
    }

    if (fromNode->linkCount < MAX_NODES) {
        fromNode->links[fromNode->linkCount++] = toNode;
        printf("Link created from '%s' to '%s'.\n", from, to);
    } else {
        printf("Max link limit reached for '%s'.\n", from);
    }
}

void displayGraph(Graph *graph) {
    if(graph->nodes[0]->name==NULL){
        printf("No Directory found\n");
    }
    else{
        printf("\nGraph Structure:\n");
        for (int i = 0; i < graph->nodeCount; i++) {
            printf("%s -> ", graph->nodes[i]->name);
            for (int j = 0; j < graph->nodes[i]->linkCount; j++) {
                printf("%s ", graph->nodes[i]->links[j]->name);
            }
            printf("\n");
        }
    }
}

int main() {
    int choice;
    char name1[MAX_NAME_LENGTH], name2[MAX_NAME_LENGTH];

    while (1) {
        printf("\n1. Add Node\n2. Add Link\n3. Display Graph\n4. Exit\nEnter choice: ");
        scanf("%d", &choice);
```

```
    switch (choice) {
        case 1:
            printf("Enter node name: ");
            scanf("%s", name1);
            addNode(&directoryGraph, name1);
            break;
        case 2:
            printf("Enter source node: ");
            scanf("%s", name1);
            printf("Enter target node: ");
            scanf("%s", name2);
            addLink(&directoryGraph, name1, name2);
            break;
        case 3:
            displayGraph(&directoryGraph);
            break;
        case 4:
            return 0;
        default:
            printf("Invalid choice.\n");
    }}}
```

OUTPUT:

```
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ ./acylic

1. Add Node
2. Add Link
3. Display Graph
4. Exit
Enter choice: 1
Enter node name: shru
Node 'shru' added.

1. Add Node
2. Add Link
3. Display Graph
4. Exit
Enter choice: 1
Enter node name: rajb
Node 'rajb' added.

1. Add Node
2. Add Link
3. Display Graph
4. Exit
Enter choice: @
Enter node name: Node '@' added.

1. Add Node
2. Add Link
3. Display Graph
4. Exit
Enter choice: 1
Enter node name: gmail.com
Node 'gmail.com' added.
```

```
1. Add Node
2. Add Link
3. Display Graph
4. Exit
Enter choice: 3

Graph Structure:
shru -> rajb
rajb -> @
@ -> gmail.com
gmail.com ->

1. Add Node
2. Add Link
3. Display Graph
4. Exit
```

6.Disk Scheduling Algorithms

Disk scheduling algorithms determine the order in which disk I/O requests are serviced. Efficient scheduling improves overall system performance by minimizing seek time, which is the time taken by the disk arm to move to the desired cylinder.

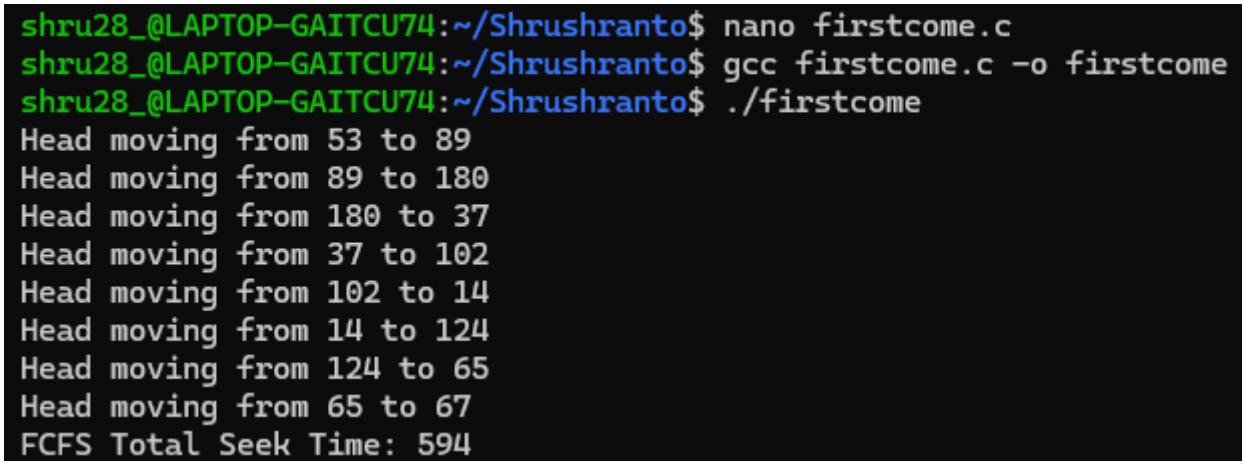
1. FCFS (First-Come, First-Served)

- Requests are processed in the order they arrive.
- Simple but can lead to long wait times if requests are scattered.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
void fcfs(int requests[], int n, int head) {
    int seek = 0;
    for (int i = 0; i < n; i++) {
        printf("Head moving from %d to %d\n", head, requests[i]);
        seek += abs(requests[i] - head);
        head = requests[i];
    }
    printf("FCFS Total Seek Time: %d\n", seek);
}
int main() {
    int requests[] = {89, 180, 37, 102, 14, 124, 65, 67};
    int n = sizeof(requests) / sizeof(requests[0]);
    int head = 53;
    fcfs(requests, n, head);
    return 0;
}
```

OUTPUT:



2. SSTF (Shortest Seek Time First)

- Selects the request closest to the current head position.
- Reduces total seek time but may cause starvation for distant requests.

Code:

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

void sstf(int requests[], int n, int head) {
    int seek = 0, done[n], count = 0;
    for (int i = 0; i < n; i++) done[i] = 0;
    while (count < n) {
        int min = 1e9, index = -1;
        for (int i = 0; i < n; i++) {
            if (!done[i] && abs(requests[i] - head) < min) {
                min = abs(requests[i] - head);
                index = i;
            }
        }
  printf("Head moving from %d to %d\n", head, requests[index]);
        seek += abs(requests[index] - head);
        head = requests[index];
        done[index] = 1;
        count++;
    }
    printf("SSTF Total Seek Time: %d\n", seek);
}

int main() {
    int requests[] = {23, 134, 57, 122, 14, 189, 73, 67};
    int n = sizeof(requests) / sizeof(requests[0]);
    int head = 53;
    sstf(requests, n, head);
    return 0;
}
```

OUTPUT:

```
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ nano sstf.c
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ gcc sstf.c -o sstf
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ ./sstf
Head moving from 53 to 57
Head moving from 57 to 67
Head moving from 67 to 73
Head moving from 73 to 122
Head moving from 122 to 134
Head moving from 134 to 189
Head moving from 189 to 23
Head moving from 23 to 14
SSTF Total Seek Time: 311
shru28_@LAPTOP-GAITCU74:~/Shrushranto$
```

3. SCAN (Elevator Algorithm)

- Moves the head in one direction, servicing requests until the end, then reverses direction.
- More efficient than FCFS and fairer than SSTF.

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
void scan(int requests[], int n, int head, int disk_size) {
   int seek = 0;
   int left[100], right[100], l = 0, r = 0;
   for (int i = 0; i < n; i++) {
      if (requests[i] < head) left[l++] = requests[i];
      else right[r++] = requests[i];
   }
   left[l++] = 0;
   for (int i = 0; i < r - 1; i++)
      for (int j = 0; j < r - i - 1; j++)
         if (right[j] > right[j + 1]) {
            int temp = right[j];
right[j] = right[j + 1];
right[j + 1] = temp;
         }
   for (int i = 0; i < l - 1; i++)
      for (int j = 0; j < l - i - 1; j++)
         if (left[j] < left[j + 1]) {
            int temp = left[j];
left[j] = left[j + 1];
left[j + 1] = temp;
         }
   for (int i = 0; i < r; i++) {
printf("Head moving from %d to %d\n", head, right[i]);
      seek += abs(right[i] - head);
      head = right[i];
   }
   for (int i = 0; i < l; i++) {
printf("Head moving from %d to %d\n", head, left[i]);
      seek += abs(left[i] - head);
      head = left[i];
   }
   printf("SCAN Total Seek Time: %d\n", seek);
}
int main() {
   int requests[] = {23, 134, 57, 122, 14, 189, 73, 67};
   int n = sizeof(requests) / sizeof(requests[0]);
   int head = 53;
   int disk_size = 200;
   scan(requests, n, head, disk_size);
   return 0;
}
```

OUTPUT:

```
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ nano scan.c
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ gcc scan.c -o scan
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ ./scan
Head moving from 53 to 57
Head moving from 57 to 67
Head moving from 67 to 73
Head moving from 73 to 122
Head moving from 122 to 134
Head moving from 134 to 189
Head moving from 189 to 23
Head moving from 23 to 14
Head moving from 14 to 0
SCAN Total Seek Time: 325
shru28_@LAPTOP-GAITCU74:~/Shrushranto$
```

4. C-SCAN (Circular SCAN)

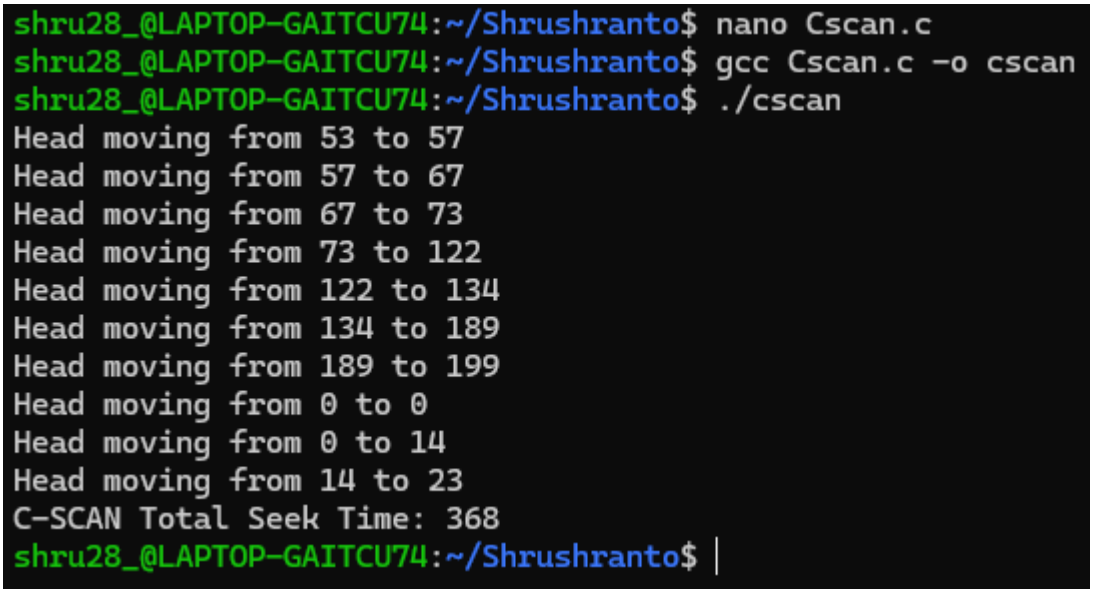
- Moves the head in one direction, servicing requests, and jumps to the beginning without servicing on the return.

Code:

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

void cscan(int requests[], int n, int head, int disk_size) {
   int seek = 0;
   int left[100], right[100], l = 0, r = 0;
   for (int i = 0; i < n; i++) {
      if (requests[i] < head) left[l++] = requests[i];
      else right[r++] = requests[i];
   }
   left[l++] = 0;
   right[r++] = disk_size - 1;
   for (int i = 0; i < r - 1; i++)
      for (int j = 0; j < r - i - 1; j++)
         if (right[j] > right[j + 1]) {
            int temp = right[j];
right[j] = right[j + 1];
right[j + 1] = temp;
         }
   for (int i = 0; i < l - 1; i++)
      for (int j = 0; j < l - i - 1; j++)
         if (left[j] > left[j + 1]) {
            int temp = left[j];
left[j] = left[j + 1];
left[j + 1] = temp;
         }
   for (int i = 0; i < r; i++) {
           printf("Head moving from %d to %d\n", head, right[i]);
      seek += abs(right[i] - head);
      head = right[i];
   }
   head = 0;
   seek += disk_size - 1;
   for (int i = 0; i < l; i++) {
           printf("Head moving from %d to %d\n", head, left[i]);
      seek += abs(left[i] - head);
      head = left[i];
   }
   printf("C-SCAN Total Seek Time: %d\n", seek);
}

int main() {
   int requests[] = {23, 134, 57, 122, 14, 189, 73, 67};
   int n = sizeof(requests) / sizeof(requests[0]);
   int head = 53;
   int disk_size = 200;
   cscan(requests, n, head, disk_size);
```

```
    return 0;
}
```

OUTPUT:



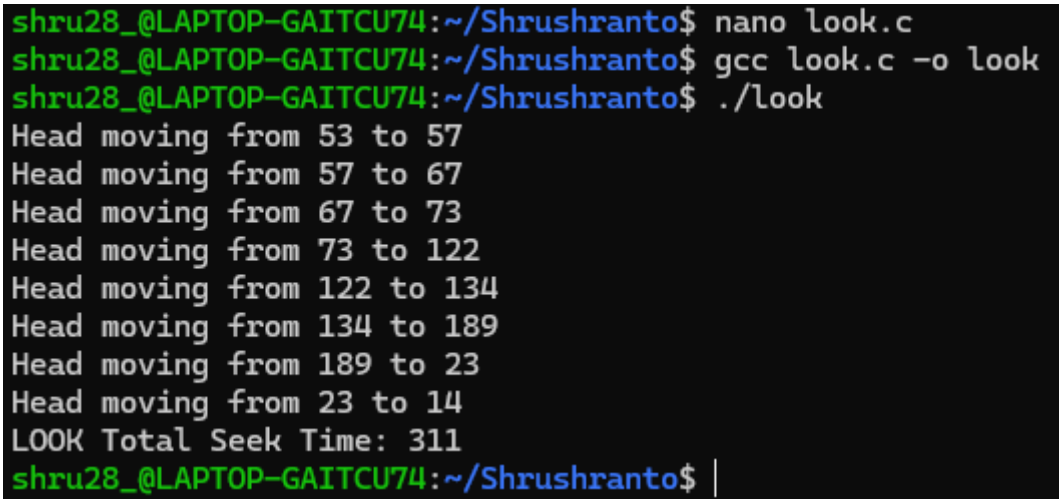
5. LOOK

- Similar to SCAN, but the head only goes as far as the final request in each direction, not to the end.

Code:

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

void look(int requests[], int n, int head) {
   int seek = 0;
   int left[100], right[100], l = 0, r = 0;
   for (int i = 0; i < n; i++) {
      if (requests[i] < head) left[l++] = requests[i];
      else right[r++] = requests[i];
   }
   for (int i = 0; i < r - 1; i++)
      for (int j = 0; j < r - i - 1; j++)
         if (right[j] > right[j + 1]) {
            int temp = right[j]; right[j] = right[j + 1]; right[j + 1] = temp;
         }
   for (int i = 0; i < l - 1; i++)
      for (int j = 0; j < l - i - 1; j++)
         if (left[j] < left[j + 1]) {
            int temp = left[j]; left[j] = left[j + 1]; left[j + 1] = temp;
         }
   for (int i = 0; i < r; i++) {
                        printf("Head moving from %d to %d\n", head, right[i]);
      seek += abs(right[i] - head);
      head = right[i];
   }
   for (int i = 0; i < l; i++) {
                        printf("Head moving from %d to %d\n", head, left[i]);
      seek += abs(left[i] - head);
      head = left[i];
   }
   printf("LOOK Total Seek Time: %d\n", seek);
}

int main(){
   int requests[] = {23, 134, 57, 122, 14, 189, 73, 67};
   int n = sizeof(requests) / sizeof(requests[0]);
   int head = 53;
   look(requests, n, head);
   return 0;
}
```

OUTPUT:


shru28_@LAPTOP-GAITCU74:~/Shrushranto$ nano look.c
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ gcc look.c -o look
shru28_@LAPTOP-GAITCU74:~/Shrushranto$ ./look
Head moving from 53 to 57
Head moving from 57 to 67
Head moving from 67 to 73
Head moving from 73 to 122
Head moving from 122 to 134
Head moving from 134 to 189
Head moving from 189 to 23
Head moving from 23 to 14
LOOK Total Seek Time: 311
shru28_@LAPTOP-GAITCU74:~/Shrushranto$

6. C-LOOK

Similar to C-SCAN, but jumps back to the lowest request instead of the start of the disk.

Code:

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

void clook(int requests[], int n, int head) {
   int seek = 0;
   int left[100], right[100], l = 0, r = 0;
   for (int i = 0; i < n; i++) {
      if (requests[i] < head) left[l++] = requests[i];
      else right[r++] = requests[i];
   }
   for (int i = 0; i < r - 1; i++)
      for (int j = 0; j < r - i - 1; j++)
         if (right[j] > right[j + 1]) {
            int temp = right[j];
right[j] = right[j + 1];
right[j + 1] = temp;
          }
   for (int i = 0; i < l - 1; i++)
      for (int j = 0; j < l - i - 1; j++)
         if (left[j] > left[j + 1]) {
            int temp = left[j];
left[j] = left[j + 1];
left[j + 1] = temp;
          }
   for (int i = 0; i < r; i++) {
            printf("Head moving from %d to %d\n", head, right[i]);
      seek += abs(right[i] - head);
      head = right[i];
   }
   if (l > 0) {
            printf("Head moving from %d to %d\n", head, left[0]);
      seek += abs(head - left[0]);
      head = left[0];
   }
   for (int i = 0; i < l; i++) {
            printf("Head moving from %d to %d\n", head, left[i]);
      seek += abs(left[i] - head);
      head = left[i];
   }
   printf("C-LOOK Total Seek Time: %d\n", seek);
}


int main(){
   int requests[] = {23, 134, 57, 122, 14, 189, 73, 67};
   int n = sizeof(requests) / sizeof(requests[0]);
   int head = 53;
   clook(requests, n, head);
```

```
    return 0;
}
```

OUTPUT: