National Institute of Technology Kurukshetra, Kurukshetra - 136119

Master Of Computer Application

MCA – 134 (2024-27)

Operating System Lab

Lab coordinator: Dr. Nidhi Gupta

Submitted by:

Name: Jayesh Solanki

Semester: 2nd

Section: G-4

Roll no: 524110055

Submitted to:

Mr. Manmohan Singh Sir

Mr. Vijay Sir

Date: 01-05-2025

DECLARATION

I, **Jayesh Solanki**, hereby declare that the **Operating System Lab Manual** submitted by me is my own original and independently completed work.

While I have referred to online resources and sample code snippets to enhance my understanding of the concepts, the structure, implementation, and content of the manual are based entirely on my own logic and effort.

This manual is distinct from others because:

1. I have included **screenshots of program outputs**, organized in a folder named after me, as proof of successful execution on my own system.
2. Each program is **modular, well-structured**, and developed using my personal understanding of the topics.
3. **No part** of this manual has been copied from any student or external source.
4. The manual is systematically organized with a **comprehensive table of contents** for easy navigation.

I fully understand and uphold the principles of academic integrity and take complete responsibility for the work I have submitted.

INDEX

Study of hardware and software requirements of different operating systems (UNIX, LINUX, WINDOWS XP, WINDOWS 7/8/10)

1. UNIX:

UNIX is a multitasking, multi-user operating system developed for workstations, servers, and other devices. Its numerous applications include database management, software development, and networked applications. Systems equipped with UNIX are preferred for their security, flexibility, and stability. This operating system is the basis for numerous others, including macOS and Linux.

Hardware Requirements:

- Processor: Minimum of 1 GHz processor
- Memory (RAM): Minimum of 1 GB RAM
- Hard Disk Space: Minimum of 10 GB free disk space.
- Input Output Devices: A keyboard and a pointing device such as 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.
- Compiler and development tools (optional)
- X Window System for graphical user interface (optional)
- Networking tools for network communication (optional)

2. Linux:

Linux is a popular operating system that is preferred for its reliability and security, performing better than its competitors in terms of protection against viruses and malware. It is also resistant to slowed-down performance, crashing, and expensive repairs and users need not pay licensing fees as often as they do for other commercial operating systems. Linux features a zero cost of entry and can be legally installed on any computer without any associated cost whatsoever.

Hardware Requirements:

- Processor: Minimum of 1 GHz processor
- Memory (RAM): Minimum of 1 GB RAM (2 GB or more for better performance)
- Hard Disk Space: Minimum of 10 GB free disk space (20 GB or more recommended for better performance)

Software Requirements:

- Kernel: Linux kernel (varies by distribution)
- Graphical user interface (optional)
- Compiler and development tools (optional)
- Networking tools for network communication (optional)

3. Windows XP:

Window XP is the most developed operating system of its time, it was originated by the Microsoft company of America in 2004, it was about 64 bit, one user-defined and multitasking operating system in which networks, facts, E-mail, multimedia was present in a newer form. It was formed in such a way that either in business or in personal usage it can be worked properly and efficiently.

Hardware Requirements:

- Processor: Minimum of Pentium 233 MHz processor (300 MHz or higher recommended)
- Memory (RAM): Minimum of 64 MB RAM (128 MB or higher recommended)
- Hard Disk Space: Minimum of 1.5 GB free disk space

Software Requirements:

- Windows XP operating system
- DirectX 9 graphics device with WDDM driver (optional for graphical user interface)
- Networking tools for network communication (optional)

4. Windows 7/8:

Windows 7 followed Windows Vista and featured various changes from prior operating systems. One of these was the Quick Launch Toolbar, which revolutionized how the users find various commands and menu options. Windows 7 also includes items like support for virtual hard disks, gaming additions, and other new features.

Windows 8 offers many new features and represents a more substantial change than some former Windows versions. This starts with a more versatile interface that can accommodate touch screen use, as well as new authentication methods.

Hardware Requirements:

- Processor: Minimum of 1 GHz processor (1 GHz or higher recommended)
- Memory (RAM): Minimum of 1 GB RAM for 32-bit or 2 GB for 64-bit.
- Hard Disk Space: Minimum of 16 GB for 32-bit OS or 20 GB for 64-bit OS.

Software Requirements:

- Windows 7 or Windows 8 operating system
- DirectX 9 graphics device with WDDM driver (optional for graphical user interface)
- Networking tools for network communication (optional)

5. Windows 10:

Windows 10 is a Microsoft operating system, a version which succeeded Windows 8 and Windows 8.1, which were unveiled in 2012 and 2013, respectively. Code-named "Threshold," Windows 10 was released to the general public in July 2015.

Hardware Requirements:

- Processor: Minimum of 1 GHz processor (1 GHz or higher recommended)
- Memory (RAM): Minimum of 1 GB RAM for 32-bit or 2 GB for 64-bit.
- Hard Disk Space: Minimum of 16 GB for 32-bit OS or 20 GB for 64-bit OS.

Software Requirements:

- Windows 7 SPI or Windows 8.1 operating system
- DirectX 9 graphics device with WDDM 1.0 or higher driver (optional for graphical user interface)
- Networking tools for network communication (optional)

Execute various UNIX system calls for Process Management, File Management, and Input/Output, Device Management

What is a System Call?

A **system call** is the interface between a user program and the operating system's kernel. When a program requires access to system resources (like files, processes, or hardware), it uses system calls to request the operating system to perform the task on its behalf.

System calls provide controlled access to resources and ensure security, stability, and proper functioning of the system.

System calls enable user programs to:

1. Manage processes (fork(), exec(), wait(), kill()).
2. Perform file operations (open(), read(), write(), close(), unlink()).
3. Interact with I/O devices (read(), write(), ioctl()).

1. Process Management System Calls

These system calls manage the lifecycle and behavior of processes.

- **fork()**:
  Creates a new process by duplicating the calling process. The new process (called the child) runs concurrently with the parent process. Both processes continue from the point where fork() was called, but the child has its own unique process ID.
- **exec()**:
  Replaces the current process's program with a new program. Once this system call is executed, the process stops running the old program and starts executing the new one. Variants of exec() allow specifying the program's arguments and environment.
- **wait()**:
  Makes the parent process wait for the termination of its child process. It ensures synchronization between parent and child processes by allowing the parent to retrieve the child's exit status.
- **kill()**:
  Sends a signal to a process or a group of processes. It can be used to terminate a process, pause it, or send other signals for specific tasks like stopping or resuming the process.

2. File Management System Calls

These system calls handle the creation, manipulation, and deletion of files.

- **open()**:
  Opens an existing file or creates a new one. It returns a file descriptor (an integer) that is used to reference the file in subsequent operations.
- **read()**:
  Reads data from a file into a buffer. The amount of data read depends on the buffer size specified by the user.
- **write()**:
  Writes data from a buffer to a file. It is often used to save data to a file or send output to a device.
- **close()**:
  Closes an open file descriptor, ensuring that all resources associated with the file are released.
- **unlink()**:
  Deletes a file. The file's data is removed only if no process has it open; otherwise, it is deleted after the last process closes it.

3. Input/Output Device Management System Calls

These system calls are used to interact with input/output devices such as keyboards, monitors, printers, and other peripherals.

- **read() and write()**:
  These system calls are also used for devices, allowing data to be read from or written to them. For example, reading from a keyboard or writing output to a screen.
- **ioctl()**:
  A special-purpose system call used for device-specific input/output operations or configurations. It allows sending control commands to devices, such as adjusting device settings or querying device status.

1. Process Management System Calls:

C Program Using fork() (File name: fork.c)

```
#include <stdio.h>
#include <unistd.h>
int main() {
  pid_t pid = fork();
  if(pid==0){
    printf("Child process run PID = %d\n",getpid());
  }else if(pid>0){
  printf("Parent process run PID = %d\n",getpid());
  }else{
  printf("Error\n"); }
return 0;
}
```

Output:

```
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ gcc fork.c -o fork
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ ./fork
Parent process run PID = 1456
Child process run PID = 1457
```

C Program Using wait() (File name: wait.c)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
  pid_t pid = fork();
  if(pid==0){
    printf("Child process PID = %d\n",getpid());
    sleep(2);
    } else{
    printf("Parent process waiting for child process.\n");
    wait(NULL);
    printf("Child process complete.\n"); }
return 0;
}
```

Output:

```
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ gcc wait.c -o wait
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ ./wait
Parent process waiting for child process.
Child process PID = 1489
Child process complete.
```

C program using execv()

(First file: demo.c)

```
#include<stdio.h>
int main(){
   printf("Call\n");
   printf("\n");
return 0;
}
```

(Main File: main.c)

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
int main(){
   pid_t pid =fork();
   char *arr[]={"./demo",NULL};
   execv(arr[0],arr);
return 0;
}
```

Output:

```
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ gcc demo.c -o demo
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ gcc main.c -o main
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ ./main
Call

Call
```

2. File Management System Calls:

Here's a single C program that demonstrates the usage of open() ,read(),write(),close(),delink().

```
#include<stdio.h>
#include<fcntl.h>
#include<unistd.h>
#include<string.h>
int main(){
//open file or waiting for it to be created
int fd=open("example.txt",O_CREAT | O_WRONLY, 0644);
if(fd== -1){
perror("Error opening file");
return 1;}
//write to the file
const char *text= "Hello Abhishek,Ubuntu file management system calls!";
ssize_t bytes_written =write (fd, text, strlen(text));
if(bytes_written== -1){
perror("Error writing to file");
close(fd);
return 1;}
printf("Written %ld bytes to file\n",bytes_written);
//Close the file
if(close(fd)==-1){
perror("Error closing file");
return 1;
}
//open file for reading
fd=open("example.txt",O_RDONLY);
if(fd==-1){
perror("Error opeing file for reading");
return 1;}
//read from the file
char buffer[128];
ssize_t bytes_read= read(fd,buffer, sizeof(buffer)-1);
if(bytes_read== -1){
perror("Error reading from file");
close(fd);
return 1;}
buffer[bytes_read]='\0' ;
printf("Read from file: %s\n", buffer);
//close the file
if(close(fd)==-1){
perror("Error closing file ");
return 1;
}return 0;}
```

Output:

```
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ gcc file.c -o file
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ ./file
Written 49 bytes to file
Read from file: Hello Jayesh,Ubuntu file management system calls!
```

3.Input/Output Device Management System Calls:

Here's a single C program that demonstrates the usage of ioctl(), read(), write().

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

int main() {
   char buffer[100];
   struct winsize w;
   write(STDOUT_FILENO, "Enter a message: ", 17);
   int bytesRead = read(STDIN_FILENO, buffer, sizeof(buffer) - 1);
   buffer[bytesRead] = '\0'; // Null-terminate the string
   write(STDOUT_FILENO, "You entered: ", 13);
   write(STDOUT_FILENO, buffer, bytesRead);
   if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &w) == 0) {
      printf("Terminal size: %d rows x %d cols\n", w.ws_row, w.ws_col);
   } else {
      perror("ioctl error"); }
   return 0;
}
```

Output:

```
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ gcc device.c -o device
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ ./device
Enter a message: Jayesh solanki
You entered: Jayesh solanki
Terminal size: 30 rows x 120 cols
```

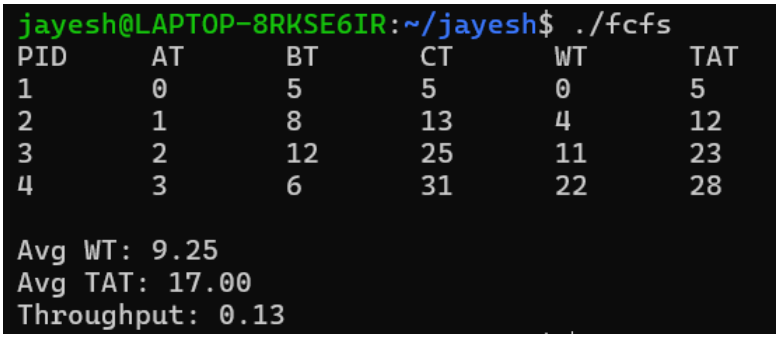
Implement CPU scheduling schemes: FCFS, SJF, SRTF, Priority, Round Robin, Multi-Level Queue, and find out the average , turn-around time, avg. waiting time, response time, and throughput

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

```
#include <stdio.h>

struct Process {
   int pid, at, bt, wt, tat, ct;
};

int main() {
   struct Process p[] = {{1, 0, 5}, {2, 1, 8}, {3, 2, 12}, {4, 3, 6}};
   int n = sizeof(p) / sizeof(p[0]);
   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("PID\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:

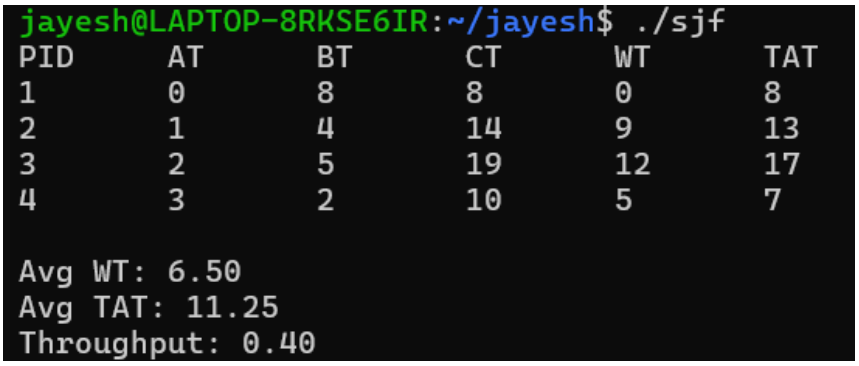


2. Shortest Job First (SJF):

```
#include <stdio.h>

struct Process {
   int pid, at, bt, wt, tat, ct;
};

int main() {
   struct Process p[] = {{1, 0, 8}, {2, 1, 4}, {3, 2, 5}, {4, 3, 2}};
   int n = sizeof(p) / sizeof(p[0]);
   int total_wt = 0, total_tat = 0, completed = 0, current_time = 0;

   int visited[n];
   for (int i = 0; i < n; i++) visited[i] = 0;

   while (completed < n) {
      int min_bt = 9999, min_index = -1;
      for (int i = 0; i < n; i++) {
         if (!visited[i] && p[i].at <= current_time && p[i].bt < min_bt) {
            min_bt = p[i].bt;
            min_index = i;
         }
      }

      if (min_index == -1) {
         current_time++;
      } else {
         visited[min_index] = 1;
         p[min_index].wt = current_time - p[min_index].at;
         p[min_index].ct = current_time + p[min_index].bt;
         p[min_index].tat = p[min_index].ct - p[min_index].at;
         current_time = p[min_index].ct;
         total_wt += p[min_index].wt;
         total_tat += p[min_index].tat;
         completed++;
      }
   }

   printf("PID\tAT\tBT\tCT\tWT\tTAT\n");
   for (int i = 0; i < n; i++) {
      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:

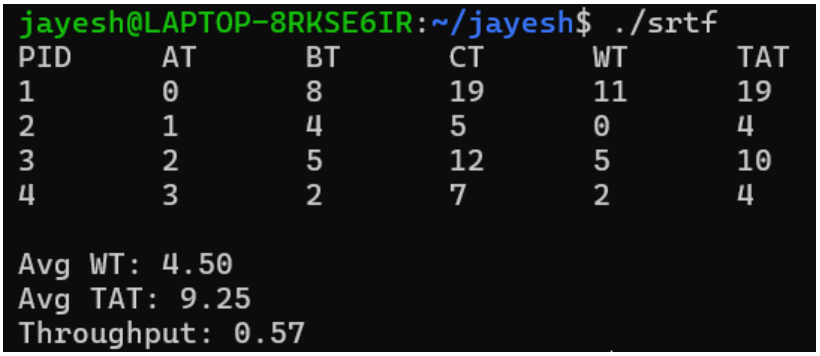


3. Shortest Remaining Time First (SRTF) - Preemptive SJF:

```
#include <stdio.h>

struct Process {
   int pid, at, bt, remaining_bt, wt, tat, ct;
};

int main() {
   struct Process p[] = {{1, 0, 8}, {2, 1, 4}, {3, 2, 5}, {4, 3, 2}};
   int n = sizeof(p) / sizeof(p[0]);
   int total_wt = 0, total_tat = 0, completed = 0, current_time = 0, min_index = -1;

   for (int i = 0; i < n; i++) p[i].remaining_bt = p[i].bt;

   while (completed < n) {
      int min_bt = 9999;
      min_index = -1;

      for (int i = 0; i < n; i++) {
         if (p[i].at <= current_time && p[i].remaining_bt > 0 && p[i].remaining_bt < min_bt) {
            min_bt = p[i].remaining_bt;
            min_index = i;
         }
      }

      if (min_index == -1) {
         current_time++;
         continue;
      }

      p[min_index].remaining_bt--;
      current_time++;

      if (p[min_index].remaining_bt == 0) {
         p[min_index].ct = current_time;
         p[min_index].tat = p[min_index].ct - p[min_index].at;
         p[min_index].wt = p[min_index].tat - p[min_index].bt;
         total_wt += p[min_index].wt;
         total_tat += p[min_index].tat;
         completed++;
      }
   }
```

```
    printf("PID\tAT\tBT\tCT\tWT\tTAT\n");

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



4. Priority Scheduling (Non-Preemptive):

```
#include <stdio.h>

struct Process {
   int pid, at, bt, priority, wt, tat, ct;
};

int main() {
   struct Process p[] = {{1, 0, 8, 2}, {2, 1, 4, 1}, {3, 2, 5, 3}, {4, 3, 2, 2}};
   int n = sizeof(p) / sizeof(p[0]);
   int total_wt = 0, total_tat = 0, completed = 0, current_time = 0;
   int visited[n];

   for (int i = 0; i < n; i++) visited[i] = 0;

   while (completed < n) {
      int max_priority = -1, min_index = -1;

      for (int i = 0; i < n; i++) {
         if (!visited[i] && p[i].at <= current_time) {
            if (p[i].priority > max_priority) {
               max_priority = p[i].priority;
               min_index = i;
            } else if (p[i].priority == max_priority && p[i].at < p[min_index].at) {
               min_index = i;
            }
         }
      }

      if (min_index == -1) {
         current_time++;
      } else {
         visited[min_index] = 1;
         p[min_index].wt = current_time - p[min_index].at;
         p[min_index].ct = current_time + p[min_index].bt;
         p[min_index].tat = p[min_index].ct - p[min_index].at;
         current_time = p[min_index].ct;
         total_wt += p[min_index].wt;
         total_tat += p[min_index].tat;
         completed++;
      }
   }
```

```
    printf("PID\tAT\tBT\tPriority\tCT\tWT\tTAT\n");

    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t\t%d\t%d\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].priority, 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:

```
jayesh@LAPTOP-8RKSE6IR:~/jayesh$ ./priorityNP
PID     AT      BT      Priority        CT      WT      TAT
1       0       8       2               8       0       8
2       1       4       1               19      14      18
3       2       5       3               13      6       11
4       3       2       2               15      10      12

Avg WT: 7.50
Avg TAT: 12.25
Throughput: 0.27
```

5. Priority Scheduling (Preemptive):

```
#include <stdio.h>

struct Process {
   int pid, at, bt, remaining_bt, priority, wt, tat, ct;
};

int main() {
   struct Process p[] = {{1, 0, 8, 2}, {2, 1, 4, 1}, {3, 2, 5, 3}, {4, 3, 2, 2}};
   int n = sizeof(p) / sizeof(p[0]);
   int total_wt = 0, total_tat = 0, completed = 0, current_time = 0, min_index = -1;

   for (int i = 0; i < n; i++) p[i].remaining_bt = p[i].bt;

   while (completed < n) {
      int max_priority = -1;
      min_index = -1;

      for (int i = 0; i < n; i++) {
         if (p[i].at <= current_time && p[i].remaining_bt > 0) {
            if (p[i].priority > max_priority) {
               max_priority = p[i].priority;
               min_index = i;
            }
         }
      }

      if (min_index == -1) {
         current_time++;
         continue;
      }

      p[min_index].remaining_bt--;
      current_time++;

      if (p[min_index].remaining_bt == 0) {
         p[min_index].ct = current_time;
         p[min_index].tat = p[min_index].ct - p[min_index].at;
         p[min_index].wt = p[min_index].tat - p[min_index].bt;
         total_wt += p[min_index].wt;
         total_tat += p[min_index].tat;
         completed++;
```

```
        }
    }

    printf("PID\tAT\tBT\tPriority\tCT\tWT\tTAT\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t%d\t%d\t\t%d\t%d\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].priority, 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:

```
jayesh@LAPTOP-8RKSE6IR:~/jayesh$ ./priorityP
PID     AT      BT      Priority        CT      WT      TAT
1       0       8       0               8       0       8
2       1       4       0               12      7       11
3       2       5       0               17      10      15
4       3       2       0               19      14      16

Avg WT: 7.75
Avg TAT: 12.50
Throughput: 0.21
```

6. Round Robin Scheduling:

```
#include <stdio.h>

struct Process {
   int pid, at, bt, remaining_bt, wt, tat, ct;
};

int main() {
   struct Process p[] = {{1, 0, 8}, {2, 1, 4}, {3, 2, 5}, {4, 3, 2}};
   int n = sizeof(p) / sizeof(p[0]);
   int time_quantum = 2, total_wt = 0, total_tat = 0, completed = 0, current_time = 0;
   int in_progress = 1;

   for (int i = 0; i < n; i++) p[i].remaining_bt = p[i].bt;

   while (completed < n) {
      in_progress = 0;

      for (int i = 0; i < n; i++) {
         if (p[i].remaining_bt > 0 && p[i].at <= current_time) {
            in_progress = 1;

            if (p[i].remaining_bt > time_quantum) {
               p[i].remaining_bt -= time_quantum;
               current_time += time_quantum;
            } else {
               current_time += p[i].remaining_bt;
               p[i].remaining_bt = 0;
               p[i].ct = current_time;
               p[i].tat = p[i].ct - p[i].at;
               p[i].wt = p[i].tat - p[i].bt;
               total_wt += p[i].wt;
               total_tat += p[i].tat;
               completed++;
            }
         }
      }

      if (!in_progress) current_time++;
   }

   printf("PID\tAT\tBT\tCT\tWT\tTAT\n");
```

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

```
jayesh@LAPTOP-8RKSE6IR:~/jayesh$ ./rr
PID     AT      BT      CT      WT      TAT
1       0       8       19      11      19
2       1       4       12      7       11
3       2       5       17      10      15
4       3       2       8       3       5

Avg WT: 7.75
Avg TAT: 12.50
Throughput: 0.50
```

Implement File Storage Allocation Techniques

File allocation is the process of assigning disk space to files in a way that ensures efficient storage and retrieval. It determines how files are stored on the disk and how the operating system accesses them.

- To optimize **storage space utilization**.
- To ensure **efficient access** and retrieval of files.
- To **prevent fragmentation** and manage free space effectively.

Types of File Allocation Methods:

1. **Contiguous Allocation**: Stores the entire file in a **continuous block** of memory.
2. **Linked Allocation**: Uses **linked blocks** where each block contains a pointer to the next block.
3. **Indexed Allocation**: Uses an **index table** to store addresses of all file blocks.

1. Contiguous Allocation:

Contiguous file allocation stores a file in a **continuous block of memory**. This ensures **fast access** but may lead to **fragmentation** when space is insufficient.

Code:

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

#define DISK_SIZE 50
int disk[DISK_SIZE];
typedef struct {
   char name[20];
   int startingBlock;
   int length;
} File;
void initializeDisk() {
   for (int i = 0; i < DISK_SIZE; i++) {
      disk[i] = 0;
   }
}
bool allocateFile(File* file, int size) {
   int start = -1, count = 0;
   for (int i = 0; i < DISK_SIZE; i++) {
      if (disk[i] == 0) {
         if (count == 0) start = i;
         count++;
      } else {
         count = 0;
```

```
        }
        if (count == size) {
            for (int j = start; j < start + size; j++) {
                disk[j] = 1;
            }
            file->startingBlock = start;
            file->length = size;
            printf("File '%s' allocated from block %d to %d\n", file->name, start, start + size - 1);
            return true;
        }
    }
    printf("File allocation failed for '%s': Not enough contiguous space.\n", file->name);
    return false;
}
void deleteFile(File* file) {
    if (file->startingBlock == -1) {
        printf("File '%s' is not allocated.\n", file->name);
        return;
    }
    for (int i = file->startingBlock; i < file->startingBlock + file->length; i++) {
        disk[i] = 0;
    }
    printf("File '%s' deleted from block %d to %d\n", file->name, file->startingBlock, file->startingBlock
+file->length - 1);

    file->startingBlock = -1;
    file->length = 0;
}
void displayDiskStatus() {
    printf("\nDisk Status: \n");
    for (int i = 0; i < DISK_SIZE; i++) {
        printf("%d", disk[i]);
        if ((i + 1) % 20 == 0) printf("\n");
    }
    printf("\n");
}

int main() {
    initializeDisk();
    File file1 = {"file1", -1, 0};
    File file2 = {"file2", -1, 0};
    File file3 = {"file3", -1, 0};

    allocateFile(&file1, 10);
    allocateFile(&file2, 5);
    displayDiskStatus();
```

```
    return 0;
}
```

Output:

```
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ ./contiguousAllocation
File 'file1' allocated from block 0 to 9
File 'file2' allocated from block 10 to 14

Disk Status:
11111111111111100000
00000000000000000000
0000000000
```

2. Linked List Allocation:

In **linked list allocation**, a file is stored in **non-contiguous** disk blocks, where each block contains a pointer to the next block. This method prevents fragmentation and allows efficient space utilization but may slow down access due to pointer traversal.

Code:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
    int blockNumber;
    struct Node *next;
} Node;

void linkedListAllocation(int blocks[], int length) {
    Node *head = NULL, *temp = NULL;
    for (int i = 0; i < length; i++) {
        Node *newNode = (Node *)malloc(sizeof(Node));
        if (newNode == NULL) {
            printf("Memory allocation failed.\n");
            exit(1);
        }

        newNode->blockNumber = blocks[i];
        newNode->next = NULL;

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

```
    printf("File allocated using Linked List Allocation: ");
    temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->blockNumber);
        temp = temp->next;
    }
    printf("End\n");

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

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

    int *blocks = (int *)malloc(length * sizeof(int));
    if (blocks == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }
    printf("Enter block numbers: ");
    for (int i = 0; i < length; i++) {
        scanf("%d", &blocks[i]);
    }

    linkedListAllocation(blocks, length);
    free(blocks);
    return 0;
}
```

Output:

```
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ ./linkedListAllocation
Enter the number of blocks: 3
Enter block numbers: 10 15 25
File allocated using Linked List Allocation: 10 -> 15 -> 25 -> End
```

3. Indexed Allocation:

Indexed allocation uses an **index block** that stores the addresses of all the blocks allocated to a file.

Code:

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

int memory[MAX_BLOCKS] = {0};

void indexedAllocation(int indexBlock, int blocks[], int length) {
    if (indexBlock < 0 || indexBlock >= MAX_BLOCKS) {
        printf("Error: Index block %d is out of range.\n", indexBlock);
        return;
    }
    if (memory[indexBlock] != 0) {
        printf("Error: Index block %d is already allocated.\n", indexBlock);
        return;
    }
    for (int i = 0; i < length; i++) {
        if (blocks[i] < 0 || blocks[i] >= MAX_BLOCKS) {
            printf("Error: Block %d is out of range.\n", blocks[i]);
            return;
        }
        if (memory[blocks[i]] != 0) {
            printf("Error: Block %d is already allocated.\n", blocks[i]);
            return;
        }
    }
    memory[indexBlock] = 1;
    for (int i = 0; i < length; i++) {
        memory[blocks[i]] = 1;
    }
    printf("File allocated using Indexed Allocation.\nIndex Block: %d\nAllocated Blocks: ", indexBlock);
    for (int i = 0; i < length; i++) {
        printf("%d ", blocks[i]);
    }
    printf("\n");
}
int main() {
    int indexBlock, length;

    printf("Enter index block: ");
    scanf("%d", &indexBlock);
    printf("Enter number of blocks: ");
```

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

    int *blocks = (int *)malloc(length * sizeof(int));
    if (blocks == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }
    printf("Enter block numbers: ");
    for (int i = 0; i < length; i++) {
        scanf("%d", &blocks[i]);
    }
    indexedAllocation(indexBlock, blocks, length);
    free(blocks);
    return 0;
}
```

Output:

Implement file directories: Single Level, Two Level, Tree Level, Acyclic Graph Directory

A file directory structure manages the organization, storage, and retrieval of files in an operating system. It acts as a container for files and other directories (subdirectories), providing a hierarchical organization of data. The directory maintains metadata such as file names, sizes, types, locations, and access permissions. Various directory structures exist, each offering different advantages and trade-offs.

Understanding various Directory Structure

1. Single Level Directory

All files are contained in a single directory, making it simple but inefficient for large systems.

Characteristics:

- Easy to implement.
- Easy and fast file retrieval.
- No hierarchical organisation.
- Naming conflicts arise when multiple user work on the system.
- Not suitable for multi-user environments.

Code:

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

#define MAX_FILES 100
#define MAX_NAME_LENGTH 50
typedef struct{
   char name[MAX_NAME_LENGTH];
int size;
   char content[1024];
} File;

typedef struct{
   File files[MAX_FILES];
   int file_count;
   char name[MAX_NAME_LENGTH];
} SingleLevelDirectory;

SingleLevelDirectory *create_directory(char *name){
```

```
   SingleLevelDirectory *dir = (SingleLevelDirectory *)malloc(sizeof(SingleLevelDirectory));
if (dir){
       dir->file_count = 0;
       strncpy(dir->name, name, MAX_NAME_LENGTH - 1);        dir-
>name[MAX_NAME_LENGTH - 1] = '\0';
   }
   else{
       printf("Memory allocation failed\n");
   }     return
dir;
}

int create_file(SingleLevelDirectory *dir, char *filename, char *content){
if (dir->file_count >= MAX_FILES) return -1;     for (int i = 0; i < dir-
>file_count; i++){
       if (strcmp(dir->files[i].name, filename) == 0)
return -1;
   }
   File *new_file = &dir->files[dir->file_count++];     strncpy(new_file-
>name, filename, MAX_NAME_LENGTH - 1);     strncpy(new_file-
>content, content, sizeof(new_file->content) - 1);     new_file->size =
strlen(content);
   return 0;
}

void list_files(SingleLevelDirectory *dir){
   printf("Directory: %s (%d files)\n", dir->name, dir->file_count);
   printf("---------------------------\n");
for (int i = 0; i < dir->file_count; i++){
       printf("File: %s, Size: %d bytes\n", dir->files[i].name, dir->files[i].size);
}}

File *find_file(SingleLevelDirectory *dir, char *filename){
   for (int i = 0; i < dir->file_count; i++){        if
(strcmp(dir->files[i].name, filename) == 0)
          return &dir->files[i];
   }
   return NULL;
}

int delete_file(SingleLevelDirectory *dir, char *filename){
   for (int i = 0; i < dir->file_count; i++){        if
(strcmp(dir->files[i].name, filename) == 0){
for (; i < dir->file_count - 1; i++)               dir-
>files[i] = dir->files[i + 1];           dir->file_count--;
return 0;        }}     return -1;
}
```

```
void free_directory(SingleLevelDirectory *dir){
free(dir);
}

int main(){
   SingleLevelDirectory *root = create_directory("system");
create_file(root, "file1.txt", "This is file 1");     create_file(root, "file2.txt",
"This is file 2");     create_file(root, "file3.txt", "This is file 3");
list_files(root);

   File *found = find_file(root, "file2.txt");
if (found){
      printf("\nFound file: %s\nContent: %s\n", found->name, found->content);
   }

   printf("\nDeleting file2.txt...\n");
delete_file(root, "file2.txt");
list_files(root);     free_directory(root);

   return 0;
}
```

Output:

```
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ ./sld
Directory: system (3 files)
--------------------------
File: file1.txt, Size: 14 bytes
File: file2.txt, Size: 14 bytes
File: file3.txt, Size: 14 bytes

Found file: file2.txt
Content: This is file 2

Deleting file2.txt...
Directory: system (2 files)
--------------------------
File: file1.txt, Size: 14 bytes
File: file3.txt, Size: 14 bytes
```

2. Two Level Directory

Each user has their own directory under a master directory, enhancing file organization.

Characteristics:

- Solves naming conflicts by providing separate directories for users.
- More structured than a single-level directory. • Still lacks flexibility in organizing files within a user's directory.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_USERS 10
#define MAX_FILES_PER_USER 20
#define MAX_NAME_LENGTH 50

typedef struct{
   char name[MAX_NAME_LENGTH];
int size;
   char content[1024];
} File;

typedef struct{
   char username[MAX_NAME_LENGTH];
File files[MAX_FILES_PER_USER];
   int file_count;
} UserDirectory;

typedef struct{
   UserDirectory user_dirs[MAX_USERS];
int user_count;
} TwoLevelDirectory;

TwoLevelDirectory *create_two_level_directory(){
   TwoLevelDirectory *dir_system = (TwoLevelDirectory *)malloc(sizeof(TwoLevelDirectory));
if (dir_system)
      dir_system->user_count = 0;
else
      printf("Memory allocation failed\n");
return dir_system;
}

int add_user(TwoLevelDirectory *dir_system, char *username){
if (dir_system->user_count >= MAX_USERS)        return -1;
   for (int i = 0; i < dir_system->user_count; i++){
```

```
        if (strcmp(dir_system->user_dirs[i].username, username) == 0)
return -1;     }
    UserDirectory *new_user = &dir_system->user_dirs[dir_system->user_count++];     strncpy(new_user-
>username, username, MAX_NAME_LENGTH - 1);     new_user->file_count = 0;
    return 0;
}

UserDirectory *get_user_directory(TwoLevelDirectory *dir_system, char *username){
for (int i = 0; i < dir_system->user_count; i++){
        if (strcmp(dir_system->user_dirs[i].username, username) == 0)
return &dir_system->user_dirs[i];
    }
    return NULL;
}

int create_file(TwoLevelDirectory *dir_system, char *username, char *filename, char *content){
    UserDirectory *user_dir = get_user_directory(dir_system, username);
if (!user_dir || user_dir->file_count >= MAX_FILES_PER_USER)
return -1;
    for (int i = 0; i < user_dir->file_count; i++){         if
(strcmp(user_dir->files[i].name, filename) == 0)
return -1;
    }
    File *new_file = &user_dir->files[user_dir->file_count++];
strncpy(new_file->name, filename, MAX_NAME_LENGTH - 1);
strncpy(new_file->content, content, sizeof(new_file->content) - 1);
new_file->size = strlen(content);
    return 0;
}

void list_user_files(TwoLevelDirectory *dir_system, char *username){
UserDirectory *user_dir = get_user_directory(dir_system, username);
if (!user_dir)         return;
    printf("User: %s (%d files)\n", user_dir->username, user_dir->file_count);
printf("--------------------------\n");     for (int i = 0; i < user_dir->file_count;
i++){
        printf("File: %s, Size: %d bytes\n", user_dir->files[i].name, user_dir->files[i].size);
}}

void list_users(TwoLevelDirectory *dir_system){
printf("System Users (%d users)\n", dir_system->user_count);
printf("--------------------------\n");     for (int i = 0; i <
dir_system->user_count; i++){
    printf("User: %s (%d files)\n", dir_system->user_dirs[i].username,
dir_system->user_dirs[i].file_count);
    }}
```

```
void free_two_level_directory(TwoLevelDirectory *dir_system){
free(dir_system);
}



int main(){
   TwoLevelDirectory *system = create_two_level_directory();

   add_user(system, "user1");
add_user(system, "user2");     add_user(system,
"user3");

   create_file(system, "user1", "notes.txt", "User1's notes");
create_file(system, "user1", "data.csv", "User1's data");
create_file(system, "user2", "notes.txt", "User2's notes");
   create_file(system, "user3", "photo.jpg", "User3's photo content");

   list_users(system);
printf("\n");
   list_user_files(system, "user1");
printf("\n");
   list_user_files(system, "user2");
   free_two_level_directory(system);

   return 0;
}
```

Output:

```
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ ./tld
System Users (3 users)
--------------------------
User: user1 (2 files)
User: user2 (1 files)
User: user3 (1 files)

User: user1 (2 files)
--------------------------
File: notes.txt, Size: 13 bytes
File: data.csv, Size: 12 bytes

User: user2 (1 files)
--------------------------
File: notes.txt, Size: 13 bytes
```

3. Tree Directory

Directories form a hierarchical tree structure, improving accessibility and organization.

Characteristics:

- Allows user to create subdirectories within their directories.
- Provides efficient file searching and organization.
- More complex than a single or two level directory but highly scalable.
- Requires more overhead for directory maintenance.

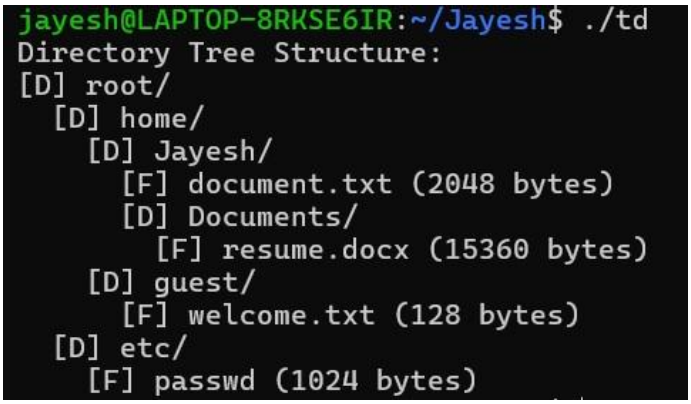
Code:

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

#define MAX_NAME_LENGTH 256
#define MAX_CHILDREN 100

typedef struct File{
   char name[MAX_NAME_LENGTH];
   size_t size;
} File;

typedef struct Directory{
   char name[MAX_NAME_LENGTH];
   struct Directory *parent;
   struct Directory *children[MAX_CHILDREN];
   int numChildren;
   File *files[MAX_CHILDREN];
int numFiles;
} Directory;

Directory *createDirectory(const char *name, Directory *parent){
Directory *dir = (Directory *)malloc(sizeof(Directory));     if (!dir)
return NULL;
   strncpy(dir->name, name, MAX_NAME_LENGTH - 1);
dir->parent = parent;     dir->numChildren = 0;
   dir->numFiles = 0;
   if (parent && parent->numChildren < MAX_CHILDREN){
parent->children[parent->numChildren++] = dir;
   }     return
dir;
}
```

```
File *createFile(const char *name, size_t size, Directory *parent){
if (!parent || parent->numFiles >= MAX_CHILDREN)        return
NULL;
   File *file = (File *)malloc(sizeof(File));
if (!file)
      return NULL;
   strncpy(file->name, name, MAX_NAME_LENGTH - 1);     file-
>size = size;
   parent->files[parent->numFiles++] = file;
   return file;
}

void printDirectoryTree(Directory *dir, int depth){
if (!dir)        return;
   for (int i = 0; i < depth; i++)       printf("
");    printf("[D] %s/\n", dir->name);    for
(int i = 0; i < dir->numFiles; i++){       for
(int j = 0; j < depth + 1; j++)
          printf("  ");
       printf("[F] %s (%zu bytes)\n", dir->files[i]->name, dir->files[i]->size);
   }
   for (int i = 0; i < dir->numChildren; i++){
      printDirectoryTree(dir->children[i], depth + 1);
   }
}

void freeDirectory(Directory *dir){
   if (!dir)
      return;
   for (int i = 0; i < dir->numFiles; i++)        free(dir-
>files[i]);
   for (int i = 0; i < dir->numChildren; i++)        freeDirectory(dir-
>children[i]);
   free(dir);
}

int main(){
   Directory *root = createDirectory("root", NULL);
   Directory *home = createDirectory("home", root);
   Directory *user1 = createDirectory("Jayesh", home);
   Directory *user2 = createDirectory("guest", home);
Directory *documents = createDirectory("Documents", user1);
createFile("passwd", 1024, createDirectory("etc", root));
createFile("welcome.txt", 128, user2);
createFile("document.txt", 2048, user1);
createFile("resume.docx", 15360, documents);     printf("Directory
```

```
Tree Structure:\n");     printDirectoryTree(root, 0);
freeDirectory(root);     return 0;
}
```

Output:



4. Acyclic Graph Directory

Supports shared directories and files, preventing duplication while maintaining accessibility.

Characteristics:

- Allows multiple users to share files without redundancy.
- Uses links to maintain file references.
- Prevents cycles to maintain system integrity.
- Facilitates efficient data management and accessibility.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_CHILDREN 10
#define MAX_NAME_LENGTH 256

typedef struct DirectoryEntry{
   char name[MAX_NAME_LENGTH];
   struct DirectoryNode *target;
int isFile;
} DirectoryEntry;

typedef struct DirectoryNode{     char
name[MAX_NAME_LENGTH];
   DirectoryEntry *entries[MAX_CHILDREN];
int entryCount;
} DirectoryNode;

DirectoryNode *createDirectory(const char *name){
   DirectoryNode *dir = (DirectoryNode *)malloc(sizeof(DirectoryNode));     strncpy(dir-
>name, name, MAX_NAME_LENGTH - 1);    dir->entryCount = 0;     return dir;
}

void addEntryToDirectory(DirectoryNode *parent, DirectoryNode *child, const char *name, int isFile){
if (!parent || (!child && !isFile))        return;
   DirectoryEntry *entry = (DirectoryEntry *)malloc(sizeof(DirectoryEntry));
   strncpy(entry->name, name, MAX_NAME_LENGTH - 1);
entry->target = child;     entry->isFile = isFile;
   parent->entries[parent->entryCount++] = entry;
}

void printDirectory(DirectoryNode *dir, int depth){
if (!dir)        return;
```

```
   for (int i = 0; i < depth; i++)
printf("  ");
   printf("%s/\n", dir->name);     for (int i =
0; i < dir->entryCount; i++){
DirectoryEntry *entry = dir->entries[i];
for (int j = 0; j < depth + 1; j++)
          printf("  ");
       printf("%s%s\n", entry->name, entry->isFile ? "" : "/");
if (!entry->isFile)
          printDirectory(entry->target, depth + 1);
   }}

void listAllPaths(DirectoryNode *dir, const char *currentPath){
char newPath[1024];     for (int i = 0; i < dir->entryCount; i++){
DirectoryEntry *entry = dir->entries[i];
       sprintf(newPath, "%s/%s", currentPath, entry->name);
if (entry->isFile){
          printf("%s\n", newPath);
       }
else{
          listAllPaths(entry->target, newPath);
       }}}

int main(){
   DirectoryNode *root = createDirectory("root");
   DirectoryNode *home = createDirectory("home");
   DirectoryNode *user1 = createDirectory("Jayesh");
   DirectoryNode *user2 = createDirectory("user2");
DirectoryNode *shared = createDirectory("shared");
addEntryToDirectory(root, home, "home", 0);
addEntryToDirectory(home, user1, "user1", 0);
addEntryToDirectory(home, user2, "user2", 0);
addEntryToDirectory(user1, shared, "shared", 0);
addEntryToDirectory(user2, shared, "shared", 0);
addEntryToDirectory(user1, NULL, "file1.txt", 1);
addEntryToDirectory(user2, NULL, "file2.txt", 1);
addEntryToDirectory(shared, NULL, "sharedFile.txt", 1);
   printf("Directory Structure:\n");
printDirectory(root, 0);     printf("\nAll
paths to files:\n");
   listAllPaths(root, "root");

   return 0;
}
```

Output:

```
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ ./agd
Directory Structure:
root/
  home/
  home/
    user1/
    Jayesh/
      shared/
      shared/
        sharedFile.txt
      file1.txt
    user2/
    user2/
      shared/
      shared/
        sharedFile.txt
      file2.txt

All paths to files:
root/home/user1/shared/sharedFile.txt
root/home/user1/file1.txt
root/home/user2/shared/sharedFile.txt
root/home/user2/file2.txt
```

Implement Disk Scheduling: FCFS, SSTF, SCAN, Circular SCAN, Look and Circular Look

Introduction to Disk Scheduling Algorithms

Disk scheduling is a crucial aspect of operating systems to manage disk I/O efficiently. It determines the order in which disk access requests are processed to minimize seek time and improve system performance.

Understanding various Disk Scheduling Algorithms

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

- **Working:** Requests are processed in the order they arrive.
- **Characteristics:**
  - Simple and easy to implement.
  - Fair, as all requests are processed in arrival order.
  - Can result in high seek time if requests are far apart.
  - Poor performance for large request queues.
- **Best For:** Low request loads where fairness is a priority.

Code:

```
#include <stdio.h>
#include <stdlib.h>
void fcfs_disk_scheduling(int requests[], int n, int head_start)
{
   int total_head_movement = 0;
   int current_position = head_start;
   printf("Order: %d", head_start);
   for (int i = 0; i < n; i++)
   {
      int distance = abs(current_position - requests[i]);
      total_head_movement += distance;
      current_position = requests[i];
      printf(" -> %d", requests[i]);
   }
   printf("\nTotal head movement: %d\n", total_head_movement);
   printf("Average head movement: %.2f\n", (float)total_head_movement / n);
}
int main()
{
   int requests[] = {98, 183, 41, 122, 14, 124, 65, 67};
```

```
    int n = sizeof(requests) / sizeof(requests[0]);
    int head_start = 53;
    printf("FCFS\n");

    fcfs_disk_scheduling(requests, n, head_start);
    return 0;
}
```

Output:

```
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ ./fcfs
FCFS
Order: 53 -> 98 -> 183 -> 41 -> 122 -> 14 -> 124 -> 65 -> 67
Total head movement: 632
Average head movement: 79.00
```

2. SSTF (Shortest Seek Time First)

- **Working:** Chooses the request closest to the current head position.
- **Characteristics:**
    - Reduces average seek time compared to FCFS.
    - More efficient for disk movement.
    - Can lead to starvation if new requests keep appearing closer to the head.
- **Best For:** When minimizing seek time is the top priority.

Code:

```
#include <stdio.h>
#include <stdlib.h>
void sstf_disk_scheduling(int requests[], int n, int head_start)
{
    int total_head_movement = 0;
    int current_position = head_start;
    int visited[n];
    for (int i = 0; i < n; i++)
    {
        visited[i] = 0;
    }
    int count = 0;
    int min_distance, min_index;
    printf("Order: %d", head_start);
    while (count != n)
    {
```

```
        min_distance = 999999;
        min_index = -1;
        for (int i = 0; i < n; i++)
        {
            int dist = abs(requests[i] - current_position);
            if (dist < min_distance && !visited[i])
            {
                min_distance = dist;
                min_index = i;
            }
        }
        int distance = abs(current_position - requests[min_index]);
        total_head_movement += distance;
        current_position = requests[min_index];
        visited[min_index] = 1;
        printf(" -> %d", requests[min_index]);
        count++;
    }
    printf("\nTotal head movement: %d\n", total_head_movement);
    printf("Average seek time: %.2f\n", (float)total_head_movement / n);
}
int main()
{
    int requests[] = {98, 183, 41, 122, 14, 124, 65, 67};
    int n = sizeof(requests) / sizeof(requests[0]);
    int head_start = 53;
    printf("SSTF\n");
    sstf_disk_scheduling(requests, n, head_start);
    return 0;
}
```

Output:

```
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ ./sstf
SSTF
Order: 53 -> 41 -> 65 -> 67 -> 98 -> 122 -> 124 -> 183 -> 14
Total head movement: 323
Average seek time: 40.38
```

3. SCAN (Elevator Algorithm)

- **Working:** Moves in one direction, servicing requests until the end, then reverses.
- **Characteristics:**
    - Ensures all requests are eventually served.
    - Reduces starvation compared to SSTF.
    - Higher seek time compared to LOOK.
    - Requests at the end may wait longer.
- **Best For**: Systems where fairness and efficiency are required.

Code:

```
#include <stdio.h>
#include <stdlib.h>
void scan_disk_scheduling(int requests[], int n, int head_start, int direction)
{
   int total_head_movement = 0;
   int current_position = head_start;
   int visited[n];
   for (int i = 0; i < n; i++)
   {
      visited[i] = 0;
   }
   int count = 0;
   int distance;
   int index = 0;
   for (int i = 0; i < n; i++)
   {
      for (int j = 0; j < n - i - 1; j++)
      {
         if (requests[j] > requests[j + 1])
         {
            int temp = requests[j];
            requests[j] = requests[j + 1];
            requests[j + 1] = temp;
         }
      }
   }
   for (int i = 0; i < n; i++)
   {
      if (requests[i] >= head_start)
      {
```

```
            index = i;
            break;
        }
    }
    printf("Order: %d", head_start);
    if (direction == 1) // Move right
    {
        for (int i = index; i < n; i++)
        {
            if (!visited[i])
            {
                distance = abs(current_position - requests[i]);
                total_head_movement += distance;
                current_position = requests[i];
                visited[i] = 1;
                printf(" -> %d", requests[i]);
                count++;
            }
        }
        distance = abs(current_position - 199);
        total_head_movement += distance;
        current_position = 199;
        printf(" -> %d", current_position);
        for (int i = n - 1; i >= 0; i--)
        {
            if (!visited[i])
            {
                distance = abs(current_position - requests[i]);
                total_head_movement += distance;
                current_position = requests[i];
                visited[i] = 1;
                printf(" -> %d", requests[i]);
                count++;
            }
        }
    }
    else
    {
        for (int i = index - 1; i >= 0; i--)
        {
            if (!visited[i])
            {
                distance = abs(current_position - requests[i]);
                total_head_movement += distance;
                current_position = requests[i];
                visited[i] = 1;
                printf(" -> %d", requests[i]);
```

```
            count++;
        }
    }
    distance = abs(current_position - 0);
    total_head_movement += distance;
    current_position = 0;
    printf(" -> %d", current_position);
    for (int i = 0; i < n; i++)
    {
        if (!visited[i])
        {
            distance = abs(current_position - requests[i]);
            total_head_movement += distance;
            current_position = requests[i];
            visited[i] = 1;
            printf(" -> %d", requests[i]);
            count++;
        }
    }
}
printf("\nTotal head movement: %d\n", total_head_movement);
printf("Average seek time: %.2f\n", (float)total_head_movement / n);
}
int main()
{
    int requests[] = {98, 183, 41, 122, 14, 124, 65, 67};
    int n = sizeof(requests) / sizeof(requests[0]);
    int head_start = 53;
    int direction = 1; // 0 for left 1 for right
    printf("SCAN\n");
    scan_disk_scheduling(requests, n, head_start, direction);
    return 0;
}
```

Output:

```
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ ./scan
SCAN
Order: 53 -> 65 -> 67 -> 98 -> 122 -> 124 -> 183 -> 199 -> 41 -> 14
Total head movement: 331
Average seek time: 41.38
```

4. C-SCAN (Circular SCAN)

- **Working:** Moves in one direction, servicing requests, and jumps back to the start instead of reversing.
- **Characteristics:**
    - Provides uniform waiting times.
    - Prevents requests at one end from waiting too long.
    - Slightly higher seek time compared to LOOK variations.
- **Best For:** High-load disk systems that require uniform response times.

Code:

```
#include <stdio.h>
#include <stdlib.h>
void C_scan_disk_scheduling(int requests[], int n, int head_start, int direction)
{
   int total_head_movement = 0;
   int current_position = head_start;
   int visited[n];
   for (int i = 0; i < n; i++)
   {
      visited[i] = 0;
   }
   int count = 0;
   int distance;

   int index = 0;
   for (int i = 0; i < n; i++)
   {
      for (int j = 0; j < n - i - 1; j++)
      {
         if (requests[j] > requests[j + 1])
         {
            int temp = requests[j];
            requests[j] = requests[j + 1];
            requests[j + 1] = temp;
         }
      }
   }
   for (int i = 0; i < n; i++)
   {
      if (requests[i] >= head_start)
      {
```

```
            index = i;
            break;
        }
    }
    printf("Order: %d", head_start);
    if (direction == 1) // Move right
    {
        for (int i = index; i < n; i++)
        {
            if (!visited[i])
            {
                distance = abs(current_position - requests[i]);
                total_head_movement += distance;
                current_position = requests[i];
                visited[i] = 1;
                printf(" -> %d", requests[i]);
                count++;
            }
        }
        distance = abs(current_position - 199);
        total_head_movement += distance;
        current_position = 199;
        printf(" -> %d", current_position);
        distance = abs(current_position - 0);
        total_head_movement += distance;
        current_position = 0;
        printf(" -> %d", current_position);
        for (int i = 0; i < n; i++)
        {
            if (!visited[i])
            {
                distance = abs(current_position - requests[i]);
                total_head_movement += distance;
                current_position = requests[i];
                visited[i] = 1;
                printf(" -> %d", requests[i]);
                count++;
            }
            else
            {
                break;
            }
        }
    }
    else
    {
        for (int i = index - 1; i >= 0; i--)
```

```
        {
            if (!visited[i])
            {
                distance = abs(current_position - requests[i]);
                total_head_movement += distance;
                current_position = requests[i];
                visited[i] = 1;
                printf(" -> %d", requests[i]);
                count++;
            }
        }
        distance = abs(current_position - 0);
        total_head_movement += distance;
        current_position = 0;
        printf(" -> %d", current_position);
        distance = abs(current_position - 199);
        total_head_movement += distance;
        current_position = 199;
        printf(" -> %d", current_position);
        for (int i = n - 1; i >= 0; i--)
        {
            if (!visited[i])
            {
                distance = abs(current_position - requests[i]);
                total_head_movement += distance;
                current_position = requests[i];
                visited[i] = 1;
                printf(" -> %d", requests[i]);
                count++;
            }
            else
            {
                break;
            }
        }
    }
    printf("\nTotal head movement: %d\n", total_head_movement);
    printf("Average seek time: %.2f\n", (float)total_head_movement / n);
}
int main()
{
    int requests[] = {98, 183, 41, 122, 14, 124, 65, 67};
    int n = sizeof(requests) / sizeof(requests[0]);
    int head_start = 53;
    int direction = 1; // 0 for left 1 for right
    printf("C-SCAN\n");
    C_scan_disk_scheduling(requests, n, head_start, direction);
```

```
    return 0;
}
```

Output:

```
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ ./cscan
C-SCAN
Order: 53 -> 65 -> 67 -> 98 -> 122 -> 124 -> 183 -> 199 -> 0 -> 14 -> 41
Total head movement: 386
Average seek time: 48.25
```

5. LOOK

- **Working:** Similar to SCAN but stops at the last request in the direction instead of moving to the end of the disk.
- **Characteristics:**
    - Avoids unnecessary movement to the end of the disk.
    - More efficient than SCAN.
    - Still has higher seek times compared to C-LOOK.
- **Best For:** Systems where reducing seek time is important but fairness is still needed

Code:

```
#include <stdio.h>
#include <stdlib.h>
void look_disk_scheduling(int requests[], int n, int head_start, int direction)
{
    int total_head_movement = 0;
    int current_position = head_start;
    int visited[n];
    for (int i = 0; i < n; i++)
    {
        visited[i] = 0;
    }
    int count = 0;
    int distance;

    int index = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
```

```
            if (requests[j] > requests[j + 1])
            {
                int temp = requests[j];
                requests[j] = requests[j + 1];
                requests[j + 1] = temp;
            }
        }
    }
    for (int i = 0; i < n; i++)
    {
        if (requests[i] >= head_start)
        {
            index = i;
            break;
        }
    }
    printf("Order: %d", head_start);
    if (direction == 1) // Move right
    {
        for (int i = index; i < n; i++)
        {
            if (!visited[i])
            {
                distance = abs(current_position - requests[i]);
                total_head_movement += distance;
                current_position = requests[i];
                visited[i] = 1;
                printf(" -> %d", requests[i]);
                count++;
            }
        }
        for (int i = n - 1; i >= 0; i--)
        {
            if (!visited[i])
            {
                distance = abs(current_position - requests[i]);
                total_head_movement += distance;
                current_position = requests[i];
                visited[i] = 1;
                printf(" -> %d", requests[i]);
                count++;
            }
        }
    }
    else
    {
        for (int i = index - 1; i >= 0; i--)
```

```
        {
            if (!visited[i])
            {
                distance = abs(current_position - requests[i]);
                total_head_movement += distance;
                current_position = requests[i];
                visited[i] = 1;
                printf(" -> %d", requests[i]);
                count++;
            }
        }
        for (int i = 0; i < n; i++)
        {
            if (!visited[i])
            {
                distance = abs(current_position - requests[i]);
                total_head_movement += distance;
                current_position = requests[i];
                visited[i] = 1;
                printf(" -> %d", requests[i]);
                count++;
            }
        }
    }
    printf("\nTotal head movement: %d\n", total_head_movement);
    printf("Average seek time: %.2f\n", (float)total_head_movement / n);
}
int main()
{
    int requests[] = {98, 183, 41, 122, 14, 124, 65, 67};
    int n = sizeof(requests) / sizeof(requests[0]);
    int head_start = 53;
    int direction = 1; // 0 for left 1 for right
    printf("LOOK\n");
    look_disk_scheduling(requests, n, head_start, direction);
    return 0;
}
```

Output:

```
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ ./look
LOOK
Order: 53 -> 65 -> 67 -> 98 -> 122 -> 124 -> 183 -> 41 -> 14
Total head movement: 299
Average seek time: 37.38
```

6. C-LOOK (Circular LOOK)

- **Working:** Similar to C-SCAN but stops at the last request instead of moving to the end, then jumps to the smallest request.
- **Characteristics:**
    - Minimizes seek time compared to C-SCAN.
    - More efficient for large queues.
    - Slightly complex to implement.
- **Best For:** Large disk queues requiring efficient scheduling.

Code:

```
#include <stdio.h>
#include <stdlib.h>
void C_look_disk_scheduling(int requests[], int n, int head_start, int direction)
{
   int total_head_movement = 0;
   int current_position = head_start;
   int visited[n];
   for (int i = 0; i < n; i++)
   {
      visited[i] = 0;
   }
   int count = 0;
   int distance;
   int index = 0;
   for (int i = 0; i < n; i++)
   {
      for (int j = 0; j < n - i - 1; j++)
      {
         if (requests[j] > requests[j + 1])
         {
            int temp = requests[j];
            requests[j] = requests[j + 1];
            requests[j + 1] = temp;
         }
      }
   }
   for (int i = 0; i < n; i++)
   {
      if (requests[i] >= head_start)
      {
         index = i;
```

```
            break;
        }
    }
    printf("Order: %d", head_start);
    if (direction == 1) // Move right
    {
        for (int i = index; i < n; i++)
        {
            if (!visited[i])
            {
                distance = abs(current_position - requests[i]);
                total_head_movement += distance;
                current_position = requests[i];
                visited[i] = 1;
                printf(" -> %d", requests[i]);
                count++;
            }
        }
        for (int i = 0; i < n; i++)
        {
            if (!visited[i])
            {
                distance = abs(current_position - requests[i]);
                total_head_movement += distance;
                current_position = requests[i];
                visited[i] = 1;
                printf(" -> %d", requests[i]);
                count++;
            }
            else
            {
                break;
            }
        }
    }
    else
    {
        for (int i = index - 1; i >= 0; i--)
        {
            if (!visited[i])
            {
                distance = abs(current_position - requests[i]);
                total_head_movement += distance;
                current_position = requests[i];
                visited[i] = 1;
                printf(" -> %d", requests[i]);
                count++;
```

```
            }
        }
        for (int i = n - 1; i >= 0; i--)
        {
            if (!visited[i])
            {
                distance = abs(current_position - requests[i]);
                total_head_movement += distance;
                current_position = requests[i];
                visited[i] = 1;
                printf(" -> %d", requests[i]);
                count++;
            }
            else
            {
                break;
            }
        }
    }

    printf("\nTotal head movement: %d\n", total_head_movement);
    printf("Average seek time: %.2f\n", (float)total_head_movement / n);
}

int main()
{
    int requests[] = {98, 183, 41, 122, 14, 124, 65, 67};
    int n = sizeof(requests) / sizeof(requests[0]);
    int head_start = 53;
    int direction = 1; // 0 for left 1 for right

    printf("C-LOOK\n");
    C_look_disk_scheduling(requests, n, head_start, direction);
    return 0;
}
```

Output:

```
jayesh@LAPTOP-8RKSE6IR:~/Jayesh$ ./clook
C-LOOK
Order: 53 -> 65 -> 67 -> 98 -> 122 -> 124 -> 183 -> 14 -> 41
Total head movement: 326
Average seek time: 40.75
```