

Operating Systems

Lab File

Name: Chandrabhushan mishra

Roll No.: 12213139

Section: IT(b)-07

Differentiation between UNIX and Windows operating system

UNIX (UNiplexed Information Computing System) and Windows are two families of operating systems with distinct characteristics. Some key differences between Unix and Windows are:

i) Target Users:

- **Unix:** Originally designed for mainframes and later adapted for servers and workstations. It is known for its multi-user and multitasking capabilities.
- **Windows:** Developed by Microsoft, Windows was initially designed for personal computers but has since evolved to support servers and other devices. It has a graphical user interface (GUI) that is a prominent feature.

ii) User Interface:

- **Unix:** Traditionally used a command-line interface (CLI), though many Unix-based systems also have graphical user interfaces. The CLI is highly powerful and customizable. They

are relatively harder to understand in one go and produce significant barriers for newcomers because it works more on terminal commands.

- **Windows:** On the other hand, it is designed with the outset of keeping the UI as simple and user-friendly as possible, so it is easy for beginners to work on. Windows is known for its graphical user interface, with a desktop environment and a variety of point-and-click applications. Command Prompt and PowerShell provide command-line interfaces on Windows.

iii) File System:

- **Unix:** Typically uses file systems like ext4, XFS. Hierarchical file system, with a root directory ("/") and a tree-like structure. Case sensitivity in file names is a common characteristic.
- **Windows:** It uses FAT32 and NTFS file systems for most installations. Originally used a drive-letter system (C:, D:, etc.) and a hierarchical structure within each drive. File names are not case-sensitive in Windows.

iv) Security Model:

- **UNIX:** UNIX/Linux follows a principle of least privilege, where users typically operate with limited permissions. Administrative tasks are performed using the sudo command. Typically uses a more granular and fine-tuned permission system based on users, groups, and others. The "root" user has superuser privileges.

- **Windows:** Uses a permissions system, with user accounts having specific rights and privileges. The Administrator account is akin to the superuser in Unix.

v) Networking:

- **Unix:** Historically strong in networking capabilities. Many internet servers run on Unix-based systems.
- **Windows:** Windows Server editions are widely used for networking, and Windows has extensive support for network protocols. Windows is commonly used in enterprise environments for file sharing and domain-based networking.

vi) Multitasking and Multi-user:

- **Unix:** Built with multitasking and multi-user capabilities from the beginning. Multiple users can log in simultaneously and run processes concurrently.
- **Windows:** While modern versions of Windows support multitasking, the emphasis was initially on single-user systems. Server editions of Windows support multi user capabilities for remote access.

vii) Cost and Licensing:

- **Unix:** Historically, many Unix variants were proprietary and required licensing fees. Some open-source Unix-like operating systems, like Linux and FreeBSD, are freely available.
- **Windows:** Generally, requires licensing fees, especially for server editions. Desktop versions often come preinstalled on new computers, and licensing costs are included in the overall system price.

viii) Development and Application Support:

- **Unix:** Rich development environment with extensive support for programming languages. Many open-source and commercial applications are available.
- **Windows:** Widely supported by commercial software vendors. Microsoft provides development tools like Visual Studio for creating Windows applications.

Basic Linux Commands

- **pwd:** Shows the path of our current directory.
- **cd:** Used to move between directories.
- **mkdir:** Make new file in current directory.
- **mv:** Move the file in another folder.
- **cp:** Copy the file in another folder.
- **touch:** Create a file in present directory.
- **touch .file:** Create a hidden file having name “.file”.
- **sudo:** We can use sudo before any command to tell as root user.
- **sudo su:** From this whether we aren't root user we can work as root user.
- **ls:** Shows all the sub directories present in our parent directory.
- **ls -R:** Used to get all the sub directories.
- **ls -a:** Used to get all sub files including hidden files.
- **ls -l:** Use a long listing format.
- **ls -r:** Show in reverse order.
- **ls -t:** Sort by time.
- **clear:** Clear the terminal.
- **history:** Shows all past commands.

Process Management System Calls

System calls:

System calls are the interface between user-level processes and the operating system kernel. They provide a way for applications to request services from the operating system, such as performing I/O operations (like reading from or writing to files), creating new processes, managing memory, and more. Some system calls are:

1) Fork()

fork() is a system call in Unix-like operating systems used to create a new process. It duplicates the calling process, resulting in two identical processes, one called the parent and the other the child. The child inherits the parent's memory, file descriptors, and other attributes, starting execution from the next instruction.

Implementation:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    int count=0;
    int n;
    printf("Enter no. of forks to be performed: ");
    scanf("%d", &n);
    printf("\n");
    for (int i=0;i<n;i++){
```

```

pid_t p1 = fork();
if(p1>0){
    printf("Parent ");
    printf("count:%d", ++count);
    printf(" p1:%d getpid():%d\n", p1, getpid());
}else if(p1==0){
    printf("Child ");
    printf("count:%d", ++count);
    printf(" p1:%d getpid():%d\n", p1, getpid());
}else{
    printf("Fork Failed");
}
}
}

```

Output:

```

Enter no. of forks to be performed: 3
Parent count:1 p1:7688 getpid():7672
Child count:1 p1:0 getpid():7688
Parent count:2 p1:7689 getpid():7672
Child count:2 p1:0 getpid():7689
Parent count:2 p1:7690 getpid():7688
Parent count:3 p1:7691 getpid():7672
Child count:2 p1:0 getpid():7690
Child count:3 p1:0 getpid():7691
Parent count:3 p1:7692 getpid():7689
Parent count:3 p1:7693 getpid():7688
Child count:3 p1:0 getpid():7692
Parent count:3 p1:7694 getpid():7690
Child count:3 p1:0 getpid():7693
Child count:3 p1:0 getpid():7694

```

2) Execute()

The `exec()` system call replaces the current process's memory image with a new program. It loads the specified executable file into memory, overwriting the current process's code and data.

This allows for dynamic program execution, enabling processes to change their behavior without creating additional processes.

Implementation:

File 1: exec2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
    printf("We are in program 1 with pid: %d\n", getpid());
    char *args[]={"/exec1",NULL};
    execvp(args[0],args);
    printf("We are back in program 1\n");
}
```

File 2: exec1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
    printf("we are in program 2 with p id: %d\n",getpid());
}
```

Output:

```
We are in program 1 with pid: 27276
we are in program 2 with p id: 27276
```


Q) Differentiate between fork and execute.

The `fork()` system call creates a new process by duplicating the current one, resulting in two separate processes (parent and child) with identical memory and execution context. In contrast, the `exec()` system call replaces the current process's memory image with a new program, loading a different executable file into memory while preserving the process identity. `fork()` is used for process creation, while `exec()` is used for changing the program executed within a process. Together, they facilitate process creation and dynamic program execution in UNIX-like operating systems.

Continuation of Previous Lab

3) Sleep()

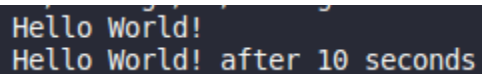
The `sleep()` function is a system call that suspends the execution of a program for a specified amount of time, typically in seconds. It allows a program to delay its execution without consuming CPU resources actively. After the specified time elapses, the program resumes execution.

Implementation:

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

int main()
{
    printf("Hello World!\n");
    sleep(5);
    printf("Hello World! after 10 seconds\n");
}
```

Output:



```
Hello World!
Hello World! after 10 seconds
```

4) Wait()

The `wait()` function is a system call that suspends the execution of a program for a specified amount of time, typically in seconds. It allows a program to delay its execution without consuming

CPU resources actively. After the specified time elapses, the program resumes execution.

Implementation:

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

int main()
{
    // printf("p1 before\n");
    // printf("#### %d ####", getpid());
    int count=0;
    int n;
    printf("Enter no. of forks to be performed: ");
    scanf("%d", &n);
    printf("\n");
    for (int i=0;i<n;i++){
        pid_t p1 = fork();
        if(p1>0){
            printf("Parent ");
            printf("count:%d", ++count);
            printf(" p1:%d getpid():%d\n", p1, getpid());
            wait(NULL);
            printf("Child has ended\n");
        }else if(p1==0){
            printf("Child ");
            printf("count:%d", ++count);
            printf(" p1:%d getpid():%d\n", p1, getpid());
        }else{
            printf("Fork Failed");
        }
    }
}
```

```
}  
}
```

Output:

```
Enter no. of forks to be performed: 3  
  
Parent count:1 p1:31253 getpid():31234  
Child count:1 p1:0 getpid():31253  
Parent count:2 p1:31254 getpid():31253  
Child count:2 p1:0 getpid():31254  
Parent count:3 p1:31255 getpid():31254  
Child count:3 p1:0 getpid():31255  
Child has ended  
Child has ended  
Parent count:3 p1:31256 getpid():31253  
Child count:3 p1:0 getpid():31256  
Child has ended  
Child has ended  
Parent count:2 p1:31257 getpid():31234  
Child count:2 p1:0 getpid():31257  
Parent count:3 p1:31258 getpid():31257  
Child count:3 p1:0 getpid():31258  
Child has ended  
Child has ended  
Parent count:3 p1:31259 getpid():31234  
Child count:3 p1:0 getpid():31259  
Child has ended
```

Q) Differentiate between sleep and wait.

Sleep() is a function that pauses the execution of a program for a specified amount of time, without any dependency on child processes. It's primarily used for timing or delaying actions within a single process.

Wait(), on the other hand, is a system call that suspends a parent process until one of its child processes terminates. It enables synchronization between parent and child processes, allowing the parent to collect exit statuses and manage child process execution. Wait() is crucial for coordinating concurrent activities within a multi-process environment.

5) Kill()

The kill() system call sends a signal to a specified process or group of processes, allowing for inter-process communication. It can be used to terminate processes, handle errors, or trigger specific actions based on the signal received. It's a versatile mechanism for managing process behavior in Unix-like operating systems.

Implementation:

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

int main(){
    int a=0;
    while(a<10){
        printf("Current Process ID: %d\n", getpid());
        a++;
    }
    kill(getpid(), 9);
    printf("Testing");
}
```

Output:

```
Current Process ID: 31502
Current Process ID: 31502
Current Process ID: 31502
Current Process ID: 31502
Current Process ID: 31502
Current Process ID: 31502
Current Process ID: 31502
Current Process ID: 31502
Current Process ID: 31502
Current Process ID: 31502
Killed
```

Task Scheduling Algorithms

1) First Come First Serve (FCFS)

FCFS (First-Come, First-Served) scheduling in OS prioritizes processes based on their arrival time, executing them in the order they arrive. It's simple but can lead to poor performance due to the lack of consideration for process burst time or priority. It's like serving customers in the order they enter a queue.

Implementation:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
int arrival_upper=10;
int arrival_lower=0;
int burst_upper=10;
int burst_lower=1;
int main(){
    int n;
    printf("\n\nEnter no. of processes: ");
    scanf("%d", &n);
    int processes[n];
    int arrival_times[n];
    int burst_times[n];
    int wait_times[n];
    int turnaround_times[n];
    int completion_times[n];

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

```

        processes[i]=i+1;
        arrival_times[i]=(rand()%(arrival_upper-
arrival_lower+1))+arrival_lower;
        burst_times[i]=(rand()%(burst_upper-
burst_lower+1))+burst_lower;
    }
    printf("\nProcesses, Arrival Times, Burst Times\n\n");
    for (int i=0;i<n;i++){
        printf("P:%d ", processes[i]);
        printf("AT:%d ", arrival_times[i]);
        printf("BT:%d ", burst_times[i]);
        printf("\n");
    }
    printf("\n");
    int time=0;
    int arrival_times_clone[n];
    for (int i=0;i<n;i++){
        arrival_times_clone[i]=arrival_times[i];
    }
    for(int i=0;i<n;i++){
        int min=999;
        int min_idx=0;
        for (int i=0;i<n;i++){
            if(arrival_times_clone[i]<min){
                min=arrival_times_clone[i];
                min_idx=i;
            }
        }
        printf("Process %d run at time %d for %d units of time\n",
processes[min_idx], time, burst_times[min_idx]);
        wait_times[min_idx]=time;
        time+=burst_times[min_idx];
        arrival_times_clone[min_idx]=999;
    }

```



```

turnaround_times[min_idx]=wait_times[min_idx]+burst_times[min_idx]
;
completion_times[min_idx]=turnaround_times[min_idx]+arrival_times[
min_idx];
}
printf("\n");
printf("Process, Arrival Time, Burst Time, Wait Time, Turn Around
Time, Completion Time\n\n");
for (int i=0;i<n;i++){
    printf("P:%d ", processes[i]);
    printf("AT:%d ", arrival_times[i]);
    printf("BT:%d ", burst_times[i]);
    printf("WT:%d ", wait_times[i]);
    printf("TAT:%d ", turnaround_times[i]);
    printf("CT:%d ", completion_times[i]);
    printf("\n");
}
int total_wait=0;
for(int i=0;i<n;i++){
    total_wait+=wait_times[i];
}
float avg_wait = (float)(total_wait)/n;
printf("\nAverage Waiting Time: %f\n\n", avg_wait);

FILE *fpt;
fpt = fopen("fcfs.csv", "w+");
fprintf(fpt,"Process No., Arrival Time, Burst Time, Wait Time, Turn
Around Time, Completion Time\n");
for (int i=0;i<n;i++){
    fprintf(fpt, "%d, %d, %d, %d, %d, %d\n", processes[i],
arrival_times[i], burst_times[i], wait_times[i], turnaround_times[i],
completion_times[i]);
}

```

```
fprintf(fpt, "\n\n\n");
fprintf(fpt, "Average Waiting Time\n%f\n", avg_wait);
fclose(fpt);
}
```

Output:

```
Enter no. of processes: 10

Processes, Arrival Times, Burst Times

P:1 AT:6 BT:7
P:2 AT:6 BT:6
P:3 AT:1 BT:6
P:4 AT:0 BT:3
P:5 AT:3 BT:2
P:6 AT:8 BT:8
P:7 AT:5 BT:10
P:8 AT:7 BT:7
P:9 AT:9 BT:7
P:10 AT:2 BT:7

Process 4 run at time 0 for 3 units of time
Process 3 run at time 3 for 6 units of time
Process 10 run at time 9 for 7 units of time
Process 5 run at time 16 for 2 units of time
Process 7 run at time 18 for 10 units of time
Process 1 run at time 28 for 7 units of time
Process 2 run at time 35 for 6 units of time
Process 8 run at time 41 for 7 units of time
Process 6 run at time 48 for 8 units of time
Process 9 run at time 56 for 7 units of time

Process, Arrival Time, Burst Time, Wait Time, Turn Around Time, Completion Time

P:1 AT:6 BT:7 WT:28 TAT:35 CT:41
P:2 AT:6 BT:6 WT:35 TAT:41 CT:47
P:3 AT:1 BT:6 WT:3 TAT:9 CT:10
P:4 AT:0 BT:3 WT:0 TAT:3 CT:3
P:5 AT:3 BT:2 WT:16 TAT:18 CT:21
P:6 AT:8 BT:8 WT:48 TAT:56 CT:64
P:7 AT:5 BT:10 WT:18 TAT:28 CT:33
P:8 AT:7 BT:7 WT:41 TAT:48 CT:55
P:9 AT:9 BT:7 WT:56 TAT:63 CT:72
P:10 AT:2 BT:7 WT:9 TAT:16 CT:18

Average Waiting Time: 25.400000
```

	A	B	C	D	E	F
1	Process No.	Arrival Time	Burst Time	Wait Time	Turn Around	Completion Time
2	1	6	7	28	35	41
3	2	6	6	35	41	47
4	3	1	6	3	9	10
5	4	0	3	0	3	3
6	5	3	2	16	18	21
7	6	8	8	48	56	64
8	7	5	10	18	28	33
9	8	7	7	41	48	55
10	9	9	7	56	63	72
11	10	2	7	9	16	18
12						
13						
14						
15	Average Waiting Time					
16	25.4					
17						

2) Shortest Job First (SJF)

SJS (Shortest Job Next) scheduling in OS selects the process with the smallest burst time for execution next. It minimizes average waiting time and turnaround time, prioritizing short tasks. However, it can lead to starvation for longer tasks and requires knowledge of future execution times, which may not always be available.

Implementation:

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

int arrival_upper=10;
int arrival_lower=0;
int burst_upper=10;
int burst_lower=1;
```

```

int main(){
    int n;
    printf("\n\nEnter no. of processes: ");
    scanf("%d", &n);

    int processes[n];
    int arrival_times[n];
    int burst_times[n];
    int wait_times[n];
    int turnaround_times[n];
    int completion_times[n];

    for (int i=0;i<n;i++){
        processes[i]=i+1;
        arrival_times[i]=(rand()%(arrival_upper-
arrival_lower+1))+arrival_lower;
        burst_times[i]=(rand()%(burst_upper-
burst_lower+1))+burst_lower;
    }
    printf("\nProcesses, Arrival Times, Burst Times\n\n");
    for (int i=0;i<n;i++){
        printf("P:%d ", processes[i]);
        printf("AT:%d ", arrival_times[i]);
        printf("BT:%d ", burst_times[i]);
        printf("\n");
    }
    printf("\n");
    int time=0;
    int burst_times_clone[n];
    for (int i=0;i<n;i++){
        burst_times_clone[i]=burst_times[i];
    }
    for(int i=0;i<n;i++){

```

```

int min=999;
int min_idx=0;
for (int i=0;i<n;i++){
    if(burst_times_clone[i]<min && arrival_times[i]<=time){
        min=burst_times_clone[i];
        min_idx=i;
    }
}
printf("Process %d run at time %d for %d units of time\n",
processes[min_idx], time, burst_times[min_idx]);
wait_times[min_idx]=time-arrival_times[min_idx];
time+=burst_times[min_idx];
burst_times_clone[min_idx]=999;
completion_times[min_idx]=time;
turnaround_times[min_idx]=completion_times[min_idx]-
arrival_times[min_idx];
}

printf("\n");
printf("Process, Arrival Time, Burst Time, Wait Time, Turn Around
Time, Completion Time\n\n");
for (int i=0;i<n;i++){
    printf("P:%d ", processes[i]);
    printf("AT:%d ", arrival_times[i]);
    printf("BT:%d ", burst_times[i]);
    printf("WT:%d ", wait_times[i]);
    printf("TAT:%d ", turnaround_times[i]);
    printf("CT:%d ", completion_times[i]);
    printf("\n");
}

int total_wait=0;
for(int i=0;i<n;i++){
    total_wait+=wait_times[i];
}

```

```

    }

    float avg_wait = (float)(total_wait)/n;
    printf("\nAverage Waiting Time: %f\n\n", avg_wait);

    FILE *fpt;

    fpt = fopen("sjf.csv", "w+");
    fprintf(fpt, "Process No., Arrival Time, Burst Time, Wait Time, Turn
    Around Time, Completion Time\n");
    for (int i=0; i<n; i++){
        fprintf(fpt, "%d, %d, %d, %d, %d, %d\n", processes[i],
        arrival_times[i], burst_times[i], wait_times[i], turnaround_times[i],
        completion_times[i]);
    }
    fprintf(fpt, "\n\n\n");
    fprintf(fpt, "Average Waiting Time\n%f\n", avg_wait);
    fclose(fpt);
}

```

Output:

```
Enter no. of processes: 10

Processes, Arrival Times, Burst Times

P:1 AT:6 BT:7
P:2 AT:6 BT:6
P:3 AT:1 BT:6
P:4 AT:0 BT:3
P:5 AT:3 BT:2
P:6 AT:8 BT:8
P:7 AT:5 BT:10
P:8 AT:7 BT:7
P:9 AT:9 BT:7
P:10 AT:2 BT:7

Process 4 run at time 0 for 3 units of time
Process 5 run at time 3 for 2 units of time
Process 3 run at time 5 for 6 units of time
Process 2 run at time 11 for 6 units of time
Process 1 run at time 17 for 7 units of time
Process 8 run at time 24 for 7 units of time
Process 9 run at time 31 for 7 units of time
Process 10 run at time 38 for 7 units of time
Process 6 run at time 45 for 8 units of time
Process 7 run at time 53 for 10 units of time

Process, Arrival Time, Burst Time, Wait Time, Turn Around Time, Completion Time

P:1 AT:6 BT:7 WT:11 TAT:18 CT:24
P:2 AT:6 BT:6 WT:5 TAT:11 CT:17
P:3 AT:1 BT:6 WT:4 TAT:10 CT:11
P:4 AT:0 BT:3 WT:0 TAT:3 CT:3
P:5 AT:3 BT:2 WT:0 TAT:2 CT:5
P:6 AT:8 BT:8 WT:37 TAT:45 CT:53
P:7 AT:5 BT:10 WT:48 TAT:58 CT:63
P:8 AT:7 BT:7 WT:17 TAT:24 CT:31
P:9 AT:9 BT:7 WT:22 TAT:29 CT:38
P:10 AT:2 BT:7 WT:36 TAT:43 CT:45

Average Waiting Time: 18.000000
```

	A	B	C	D	E	F	G
1	Process No.	Arrival Time	Burst Time	Wait Time	Turn Around	Completion Time	
2	1	6	7	11	18	24	
3	2	6	6	5	11	17	
4	3	1	6	4	10	11	
5	4	0	3	0	3	3	
6	5	3	2	0	2	5	
7	6	8	8	37	45	53	
8	7	5	10	48	58	63	
9	8	7	7	17	24	31	
10	9	9	7	22	29	38	
11	10	2	7	36	43	45	
12							
13							
14							
15	Average Waiting Time						
16	18						
17							

3) Shortest Remaining Time First (SRTF) (Preemptive SJF)

Shortest Remaining Time First (SRTF) scheduling in OS preempts the currently running process if a new process with a shorter burst time arrives. It optimizes for minimal remaining time, reducing response time. However, it can cause frequent context switches and potential starvation for longer processes.

Implementation:

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

int arrival_upper = 10;
int arrival_lower = 0;
int burst_upper = 10;
int burst_lower = 1;
```



```

int main(){
    int n;
    printf("\n\nEnter no. of processes: ");
    scanf("%d", &n);

    int processes[n];
    int arrival_times[n];
    int burst_times[n];
    int wait_times[n];
    int turnaround_times[n];
    int completion_times[n];

    for (int i = 0; i < n; i++){
        processes[i] = i + 1;
        arrival_times[i] = (rand() % (arrival_upper - arrival_lower + 1)) +
arrival_lower;
        burst_times[i] = (rand() % (burst_upper - burst_lower + 1)) +
burst_lower;
    }
    printf("\nProcesses, Arrival Times, Burst Times\n\n");
    for (int i = 0; i < n; i++){
        printf("P:%d ", processes[i]);
        printf("AT:%d ", arrival_times[i]);
        printf("BT:%d ", burst_times[i]);
        printf("\n");
    }
    printf("\n");
    int time = 0;
    int remaining_burst[n];
    for (int i = 0; i < n; i++){
        remaining_burst[i] = burst_times[i];
    }
    int completed = 0;

```

```

while (completed < n) {
    int min = 999;
    int min_idx = -1;

    for (int i = 0; i < n; i++) {
        if (remaining_burst[i] < min && arrival_times[i] <= time &&
remaining_burst[i] > 0) {
            min = remaining_burst[i];
            min_idx = i;
        }
    }

    if (min_idx == -1) {
        time++;
        continue;
    }

    printf("Process %d run at time %d for 1 unit of time\n",
processes[min_idx], time);
    remaining_burst[min_idx]--;
    time++;

    if (remaining_burst[min_idx] == 0) {
        completion_times[min_idx] = time;
        turnaround_times[min_idx] = completion_times[min_idx] -
arrival_times[min_idx];
        wait_times[min_idx] = turnaround_times[min_idx] -
burst_times[min_idx];
        completed++;
    }
}

printf("\n");

```

```

printf("Process, Arrival Time, Burst Time, Wait Time, Turn Around
Time, Completion Time\n\n");
for (int i = 0; i < n; i++){
    printf("P:%d ", processes[i]);
    printf("AT:%d ", arrival_times[i]);
    printf("BT:%d ", burst_times[i]);
    printf("WT:%d ", wait_times[i]);
    printf("TAT:%d ", turnaround_times[i]);
    printf("CT:%d ", completion_times[i]);
    printf("\n");
}

int total_wait = 0;
for (int i = 0; i < n; i++){
    total_wait += wait_times[i];
}

float avg_wait = (float)(total_wait) / n;
printf("\nAverage Waiting Time: %f\n\n", avg_wait);

FILE *fpt;

fpt = fopen("srtf.csv", "w+");
fprintf(fpt, "Process No., Arrival Time, Burst Time, Wait Time, Turn
Around Time, Completion Time\n");
for (int i = 0; i < n; i++){
    fprintf(fpt, "%d, %d, %d, %d, %d, %d\n", processes[i],
arrival_times[i], burst_times[i], wait_times[i], turnaround_times[i],
completion_times[i]);
}
fprintf(fpt, "\n\n\n");
fprintf(fpt, "Average Waiting Time\n%f\n", avg_wait);
fclose(fpt);
}

```

Output:

```
Enter no. of processes: 10
```

```
Processes, Arrival Times, Burst Times
```

```
P:1 AT:6 BT:7  
P:2 AT:6 BT:6  
P:3 AT:1 BT:6  
P:4 AT:0 BT:3  
P:5 AT:3 BT:2  
P:6 AT:8 BT:8  
P:7 AT:5 BT:10  
P:8 AT:7 BT:7  
P:9 AT:9 BT:7  
P:10 AT:2 BT:7
```

```
Process 4 run at time 0 for 1 unit of time  
Process 4 run at time 1 for 1 unit of time  
Process 4 run at time 2 for 1 unit of time  
Process 5 run at time 3 for 1 unit of time  
Process 5 run at time 4 for 1 unit of time  
Process 3 run at time 5 for 1 unit of time  
Process 3 run at time 6 for 1 unit of time  
Process 3 run at time 7 for 1 unit of time  
Process 3 run at time 8 for 1 unit of time  
Process 3 run at time 9 for 1 unit of time  
Process 3 run at time 10 for 1 unit of time  
Process 2 run at time 11 for 1 unit of time  
Process 2 run at time 12 for 1 unit of time  
Process 2 run at time 13 for 1 unit of time  
Process 2 run at time 14 for 1 unit of time  
Process 2 run at time 15 for 1 unit of time  
Process 2 run at time 16 for 1 unit of time  
Process 1 run at time 17 for 1 unit of time  
Process 1 run at time 18 for 1 unit of time  
Process 1 run at time 19 for 1 unit of time  
Process 1 run at time 20 for 1 unit of time  
Process 1 run at time 21 for 1 unit of time  
Process 1 run at time 22 for 1 unit of time  
Process 1 run at time 23 for 1 unit of time  
Process 8 run at time 24 for 1 unit of time  
Process 8 run at time 25 for 1 unit of time  
Process 8 run at time 26 for 1 unit of time  
Process 8 run at time 27 for 1 unit of time  
Process 8 run at time 28 for 1 unit of time  
Process 8 run at time 29 for 1 unit of time  
Process 8 run at time 30 for 1 unit of time  
Process 9 run at time 31 for 1 unit of time  
Process 9 run at time 32 for 1 unit of time  
Process 9 run at time 33 for 1 unit of time  
Process 9 run at time 34 for 1 unit of time  
Process 9 run at time 35 for 1 unit of time  
Process 9 run at time 36 for 1 unit of time  
Process 9 run at time 37 for 1 unit of time  
Process 10 run at time 38 for 1 unit of time  
Process 10 run at time 39 for 1 unit of time  
Process 10 run at time 40 for 1 unit of time  
Process 10 run at time 41 for 1 unit of time  
Process 10 run at time 42 for 1 unit of time  
Process 10 run at time 43 for 1 unit of time  
Process 10 run at time 44 for 1 unit of time  
Process 6 run at time 45 for 1 unit of time  
Process 6 run at time 46 for 1 unit of time  
Process 6 run at time 47 for 1 unit of time  
Process 6 run at time 48 for 1 unit of time  
Process 6 run at time 49 for 1 unit of time  
Process 6 run at time 50 for 1 unit of time  
Process 6 run at time 51 for 1 unit of time  
Process 6 run at time 52 for 1 unit of time  
Process 7 run at time 53 for 1 unit of time  
Process 7 run at time 54 for 1 unit of time  
Process 7 run at time 55 for 1 unit of time  
Process 7 run at time 56 for 1 unit of time  
Process 7 run at time 57 for 1 unit of time  
Process 7 run at time 58 for 1 unit of time  
Process 7 run at time 59 for 1 unit of time  
Process 7 run at time 60 for 1 unit of time  
Process 7 run at time 61 for 1 unit of time  
Process 7 run at time 62 for 1 unit of time
```

Process, Arrival Time, Burst Time, Wait Time, Turn Around Time, Completion Time

P:1 AT:6 BT:7 WT:11 TAT:18 CT:24
P:2 AT:6 BT:6 WT:5 TAT:11 CT:17
P:3 AT:1 BT:6 WT:4 TAT:10 CT:11
P:4 AT:0 BT:3 WT:0 TAT:3 CT:3
P:5 AT:3 BT:2 WT:0 TAT:2 CT:5
P:6 AT:8 BT:8 WT:37 TAT:45 CT:53
P:7 AT:5 BT:10 WT:48 TAT:58 CT:63
P:8 AT:7 BT:7 WT:17 TAT:24 CT:31
P:9 AT:9 BT:7 WT:22 TAT:29 CT:38
P:10 AT:2 BT:7 WT:36 TAT:43 CT:45

Average Waiting Time: 18.000000

	A	B	C	D	E	F	
1	Process No.	Arrival Time	Burst Time	Wait Time	Turn Around	Completion Time	
2	1	6	7	11	18	24	
3	2	6	6	5	11	17	
4	3	1	6	4	10	11	
5	4	0	3	0	3	3	
6	5	3	2	0	2	5	
7	6	8	8	37	45	53	
8	7	5	10	48	58	63	
9	8	7	7	17	24	31	
10	9	9	7	22	29	38	
11	10	2	7	36	43	45	
12							
13							
14							
15	Average Waiting Time						
16	18						
17							

Continuation of the previous lab

4) Priority Scheduling

Priority scheduling in operating systems assigns priorities to processes and executes them based on their priority levels. Higher priority processes are executed before lower priority ones. It aims to optimize system performance by ensuring that important tasks are completed promptly. However, it can lead to starvation if lower priority processes are constantly preempted by higher priority ones. Dynamic priority adjustments and techniques like aging are employed to mitigate starvation and ensure fairness in resource allocation.

Implementation:

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

int arrival_upper=10;
int arrival_lower=0;
int burst_upper=10;
int burst_lower=1;
int priority_upper=10;
int priority_lower=1;

int main(){
    int n;
    printf("\n\nEnter no. of processes: ");
```

```

scanf("%d", &n);

int processes[n];
int arrival_times[n];
int burst_times[n];
int priority[n];
int wait_times[n];
int turnaround_times[n];
int completion_times[n];

for (int i=0;i<n;i++){
    processes[i]=i+1;
    arrival_times[i]=(rand()%(arrival_upper-
arrival_lower+1))+arrival_lower;
    priority[i]=(rand()%(priority_upper-
priority_lower+1))+priority_lower;
    burst_times[i]=(rand()%(burst_upper-
burst_lower+1))+burst_lower;
}
printf("\nProcesses, Arrival Times, Burst Times, Priority\n\n");
for (int i=0;i<n;i++){
    printf("P:%d ", processes[i]);
    printf("AT:%d ", arrival_times[i]);
    printf("BT:%d ", burst_times[i]);
    printf("P:%d ", priority[i]);
    printf("\n");
}
printf("\n");
int time=0;
int priority_clone[n];
for (int i=0;i<n;i++){
    priority_clone[i]=priority[i];
}
for(int i=0;i<n;i++){

```

```

int min=999;
int min_idx=0;
for (int i=0;i<n;i++){
    if(priority_clone[i]<=min && arrival_times[i]<=time){
        if(priority_clone[i]==min){
            if (burst_times[min_idx]<burst_times[i]){
                continue;
            }
        }
        min=priority_clone[i];
        min_idx=i;
    }
}
printf("Process %d run at time %d for %d units of time\n",
processes[min_idx], time, burst_times[min_idx]);
wait_times[min_idx]=time-arrival_times[min_idx];
time+=burst_times[min_idx];
priority_clone[min_idx]=999;
completion_times[min_idx]=time;
turnaround_times[min_idx]=completion_times[min_idx]-
arrival_times[min_idx];
}

printf("\n");
printf("Process, Arrival Time, Burst Time, Priority, Wait Time, Turn
Around Time, Completion Time\n\n");
for (int i=0;i<n;i++){
    printf("P:%d ", processes[i]);
    printf("AT:%d ", arrival_times[i]);
    printf("BT:%d ", burst_times[i]);
    printf("P:%d ", priority[i]);
    printf("WT:%d ", wait_times[i]);
    printf("TAT:%d ", turnaround_times[i]);
    printf("CT:%d ", completion_times[i]);

```



```

        printf("\n");
    }

    int total_wait=0;
    for(int i=0;i<n;i++){
        total_wait+=wait_times[i];
    }

    float avg_wait = (float)(total_wait)/n;
    printf("\nAverage Waiting Time: %f\n\n", avg_wait);

    FILE *fpt;

    fpt = fopen("priority.csv", "w+");
    fprintf(fpt,"Process No., Arrival Time, Burst Time, Priority, Wait Time,
Turn Around Time, Completion Time\n");
    for (int i=0;i<n;i++){
        fprintf(fpt, "%d, %d, %d, %d, %d, %d, %d\n", processes[i],
arrival_times[i], burst_times[i], priority[i], wait_times[i],
turnaround_times[i], completion_times[i]);
    }
    fprintf(fpt,"\n\n\n");
    fprintf(fpt, "Average Waiting Time\n%f\n", avg_wait);
    fclose(fpt);
}

```

Output:

```
Enter no. of processes: 10
```

```
Processes, Arrival Times, Burst Times, Priority
```

```
P:1 AT:6 BT:8 P:7  
P:2 AT:2 BT:6 P:4  
P:3 AT:0 BT:10 P:3  
P:4 AT:1 BT:8 P:3  
P:5 AT:5 BT:4 P:10  
P:6 AT:4 BT:7 P:1  
P:7 AT:2 BT:2 P:7  
P:8 AT:8 BT:10 P:8  
P:9 AT:4 BT:3 P:1  
P:10 AT:10 BT:6 P:8
```

```
Process 3 run at time 0 for 10 units of time  
Process 9 run at time 10 for 3 units of time  
Process 6 run at time 13 for 7 units of time  
Process 4 run at time 20 for 8 units of time  
Process 2 run at time 28 for 6 units of time  
Process 7 run at time 34 for 2 units of time  
Process 1 run at time 36 for 8 units of time  
Process 10 run at time 44 for 6 units of time  
Process 8 run at time 50 for 10 units of time  
Process 5 run at time 60 for 4 units of time
```

```
Process, Arrival Time, Burst Time, Priority, Wait Time, Turn Around Time, Completion Time
```

```
P:1 AT:6 BT:8 P:7 WT:30 TAT:38 CT:44  
P:2 AT:2 BT:6 P:4 WT:26 TAT:32 CT:34  
P:3 AT:0 BT:10 P:3 WT:0 TAT:10 CT:10  
P:4 AT:1 BT:8 P:3 WT:19 TAT:27 CT:28  
P:5 AT:5 BT:4 P:10 WT:55 TAT:59 CT:64  
P:6 AT:4 BT:7 P:1 WT:9 TAT:16 CT:20  
P:7 AT:2 BT:2 P:7 WT:32 TAT:34 CT:36  
P:8 AT:8 BT:10 P:8 WT:42 TAT:52 CT:60  
P:9 AT:4 BT:3 P:1 WT:6 TAT:9 CT:13  
P:10 AT:10 BT:6 P:8 WT:34 TAT:40 CT:50
```

```
Average Waiting Time: 25.299999
```

	A	B	C	D	E	F	G
1	Process No.	Arrival Time	Burst Time	Priority	Wait Time	Turn Around	Completion Time
2	1	6	8	7	30	38	44
3	2	2	6	4	26	32	34
4	3	0	10	3	0	10	10
5	4	1	8	3	19	27	28
6	5	5	4	10	55	59	64
7	6	4	7	1	9	16	20
8	7	2	2	7	32	34	36
9	8	8	10	8	42	52	60
10	9	4	3	1	6	9	13
11	10	10	6	8	34	40	50
12							
13							
14							
15	Average Waiting Time						
16	25.299999						
17							

5)Round Robin Scheduling

Round-robin (RR) scheduling in operating systems allocates CPU time to processes in fixed time slices, typically called quantum. Each process receives CPU time for a quantum duration, and if it doesn't complete within that time, it's preempted and placed at the end of the ready queue. RR ensures fairness among processes, prevents starvation, and provides better response times for interactive tasks. However, it may suffer from high context-switching overhead and might not be optimal for long-running tasks.

Implementation:

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

int arrival_upper = 10;
int arrival_lower = 0;
```

```

int burst_upper = 10;
int burst_lower = 1;
int time_quantum;

int main() {
    int n;
    printf("\n\nEnter no. of processes: ");
    scanf("%d", &n);
    printf("\n\nEnter time quantum: ");
    scanf("%d", &time_quantum);

    int processes[n];
    int arrival_times[n];
    int burst_times[n];
    int wait_times[n];
    int turnaround_times[n];
    int completion_times[n];

    for (int i = 0; i < n; i++) {
        processes[i] = i + 1;
        arrival_times[i] = (rand() % (arrival_upper - arrival_lower + 1)) +
arrival_lower;
        burst_times[i] = (rand() % (burst_upper - burst_lower + 1)) +
burst_lower;
    }
    printf("\nProcesses, Arrival Times, Burst Times\n\n");
    for (int i = 0; i < n; i++) {
        printf("P:%d ", processes[i]);
        printf("AT:%d ", arrival_times[i]);
        printf("BT:%d ", burst_times[i]);
        printf("\n");
    }
    printf("\n");
    int time = 0;

```

```

int remaining_burst_times[n];
for (int i = 0; i < n; i++) {
    remaining_burst_times[i] = burst_times[i];
}

while (1) {
    int all_processes_completed = 1;
    for (int i = 0; i < n; i++) {
        if (remaining_burst_times[i] > 0 && arrival_times[i] <= time) {
            all_processes_completed = 0;
            if (remaining_burst_times[i] > time_quantum) {
                printf("Process %d run at time %d for %d units of time\n",
processes[i], time, time_quantum);
                time += time_quantum;
                remaining_burst_times[i] -= time_quantum;
            } else {
                printf("Process %d run at time %d for %d units of time\n",
processes[i], time, remaining_burst_times[i]);
                time += remaining_burst_times[i];
                remaining_burst_times[i] = 0;
                completion_times[i] = time;
                turnaround_times[i] = completion_times[i] - arrival_times[i];
                wait_times[i] = turnaround_times[i] - burst_times[i];
            }
        }
    }
    if (all_processes_completed == 1) {
        break;
    }
}

printf("\n");
printf("Process, Arrival Time, Burst Time, Wait Time, Turn Around
Time, Completion Time\n\n");

```

```

for (int i = 0; i < n; i++) {
    printf("P:%d ", processes[i]);
    printf("AT:%d ", arrival_times[i]);
    printf("BT:%d ", burst_times[i]);
    printf("WT:%d ", wait_times[i]);
    printf("TAT:%d ", turnaround_times[i]);
    printf("CT:%d ", completion_times[i]);
    printf("\n");
}

int total_wait = 0;
for (int i = 0; i < n; i++) {
    total_wait += wait_times[i];
}

float avg_wait = (float)(total_wait) / n;
printf("\nAverage Waiting Time: %f\n\n", avg_wait);

FILE *fpt;

fpt = fopen("rr.csv", "w+");
fprintf(fpt, "Process No., Arrival Time, Burst Time, Wait Time, Turn
Around Time, Completion Time\n");
for (int i = 0; i < n; i++) {
    fprintf(fpt, "%d, %d, %d, %d, %d, %d\n", processes[i],
arrival_times[i], burst_times[i], wait_times[i], turnaround_times[i],
completion_times[i]);
}
fprintf(fpt, "\n\n\n");
fprintf(fpt, "Average Waiting Time\n%f\n", avg_wait);
fclose(fpt);
}

```

Output:

Enter no. of processes: 10

Enter time quantum: 7

Processes, Arrival Times, Burst Times

P:1 AT:6 BT:7
P:2 AT:6 BT:6
P:3 AT:1 BT:6
P:4 AT:0 BT:3
P:5 AT:3 BT:2
P:6 AT:8 BT:8
P:7 AT:5 BT:10
P:8 AT:7 BT:7
P:9 AT:9 BT:7
P:10 AT:2 BT:7

Process 4 run at time 0 for 3 units of time
Process 5 run at time 3 for 2 units of time
Process 7 run at time 5 for 7 units of time
Process 8 run at time 12 for 7 units of time
Process 9 run at time 19 for 7 units of time
Process 10 run at time 26 for 7 units of time
Process 1 run at time 33 for 7 units of time
Process 2 run at time 40 for 6 units of time
Process 3 run at time 46 for 6 units of time
Process 6 run at time 52 for 7 units of time
Process 7 run at time 59 for 3 units of time
Process 6 run at time 62 for 1 units of time

Process, Arrival Time, Burst Time, Wait Time, Turn Around Time, Completion Time

P:1 AT:6 BT:7 WT:27 TAT:34 CT:40
P:2 AT:6 BT:6 WT:34 TAT:40 CT:46
P:3 AT:1 BT:6 WT:45 TAT:51 CT:52
P:4 AT:0 BT:3 WT:0 TAT:3 CT:3
P:5 AT:3 BT:2 WT:0 TAT:2 CT:5
P:6 AT:8 BT:8 WT:47 TAT:55 CT:63
P:7 AT:5 BT:10 WT:47 TAT:57 CT:62
P:8 AT:7 BT:7 WT:5 TAT:12 CT:19
P:9 AT:9 BT:7 WT:10 TAT:17 CT:26
P:10 AT:2 BT:7 WT:24 TAT:31 CT:33

Average Waiting Time: 23.900000

	A	B	C	D	E	F	
1	Process No.	Arrival Time	Burst Time	Wait Time	Turn Around	Completion Time	
2	1	6	7	27	34	40	
3	2	6	6	34	40	46	
4	3	1	6	45	51	52	
5	4	0	3	0	3	3	
6	5	3	2	0	2	5	
7	6	8	8	47	55	63	
8	7	5	10	47	57	62	
9	8	7	7	5	12	19	
10	9	9	7	10	17	26	
11	10	2	7	24	31	33	
12							
13							
14							
15	Average Waiting Time						
16	23.9						
17							

Q) Which CPU scheduling algorithm gives the best turnaround time and waiting time?

To compare the turnaround time and waiting time for different scheduling algorithms, we need to understand their characteristics:

FCFS (First-Come, First-Served):

Turnaround Time: FCFS may result in higher turnaround time, especially if longer processes arrive first, causing shorter processes to wait longer.

Waiting Time: FCFS may lead to higher waiting times, particularly for longer processes that arrive early and need to wait for shorter processes to complete.

SJF (Shortest Job First):

Turnaround Time: SJF typically provides better turnaround time by prioritizing shorter jobs for execution, reducing the time each job spends in the system.

Waiting Time: SJF minimizes waiting time by executing shorter jobs first, thereby reducing the average waiting time for processes in the queue.

SRTF (Shortest Remaining Time First):

Turnaround Time: SRTF optimizes turnaround time by preemptively scheduling the shortest available jobs, potentially achieving the shortest possible turnaround times.

Waiting Time: SRTF also minimizes waiting time by prioritizing shorter jobs for execution, reducing the waiting time for other processes in the queue.

Priority Scheduling:

Turnaround Time: Priority scheduling can provide better turnaround time for higher priority processes, ensuring important tasks are completed promptly.

Waiting Time: Higher priority processes may experience lower waiting times, but lower priority processes could suffer from starvation if constantly preempted.

RR (Round-Robin) Scheduling:

Turnaround Time: RR scheduling aims to provide fair CPU time allocation among processes, which may result in moderate turnaround times depending on the quantum size.

Waiting Time: RR scheduling can prevent processes from waiting too long, particularly for short tasks, but longer tasks might suffer from frequent preemptions.

In summary, SJF and SRTF generally offer better turnaround time and waiting time compared to FCFS, priority scheduling, and RR scheduling. However, the performance of each algorithm can vary based on factors such as the workload characteristics, quantum size in RR scheduling, and priority assignment strategy in priority scheduling.

Continuation of the previous lab

6) Multilevel Queue Scheduling

Multilevel queue scheduling is a process scheduling method where the ready queue is divided into multiple priority levels. Each level has its own queue and scheduling algorithm. Processes move between queues based on their priority or behavior, allowing for efficient management of tasks with varying levels of urgency.

Implementation:

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

int main()
{
    int n;
    printf("Enter no. of processes: ");
    scanf("%d",&n);
    int processes[n];
    int burst_times[n];
    int queue[n];
    int waiting_times[n];
    int turnaround_times[n];
    int temp;
    float avg_wait;

    int burst_upper = 10;
```

```

int burst_lower = 1;

for(int i=0;i<n;i++)
{
    processes[i] = i+1;
    burst_times[i] = (rand() % (burst_upper - burst_lower + 1)) +
burst_lower;
    queue[i]=rand() % 2;
}

for(int i=0;i<n;i++)
    for(int k=i+1;k<n;k++)
        if(queue[i] > queue[k]){
            temp=processes[i];
            processes[i]=processes[k];
            processes[k]=temp;
            temp=burst_times[i];
            burst_times[i]=burst_times[k];
            burst_times[k]=temp;
            temp=queue[i];
            queue[i]=queue[k];
            queue[k]=temp;
        }

avg_wait = waiting_times[0] = 0;

for(int i=1;i<n;i++)
{
    waiting_times[i] = waiting_times[i-1] + burst_times[i-1];
    turnaround_times[i] = turnaround_times[i-1] + burst_times[i];
    avg_wait = avg_wait + waiting_times[i];
}

printf("\nPROCESS\t\tSYSTEM/USER PROCESS \tBURST
TIME\tWAITING TIME\tTURNAROUND TIME");

```

```

for(int i=0;i<n;i++)
    printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d",
processes[i],queue[i],burst_times[i],waiting_times[i],turnaround_times
[i]);
printf("\nAverage Waiting Time: %f",avg_wait/n);

FILE *fpt;

fpt = fopen("mlq.csv", "w+");
fprintf(fpt, "Process No., Queue No., Burst Time, Wait Time, Turn
Around Time\n");
for (int i = 0; i < n; i++) {
    fprintf(fpt, "%d, %d, %d, %d, %d\n", processes[i], queue[i],
burst_times[i], waiting_times[i], turnaround_times[i]);
}
fprintf(fpt, "\n\n\n");
fprintf(fpt, "Average Waiting Time\n%f\n", avg_wait);
fclose(fpt);
}

```

Output:

PROCESS	SYSTEM/USER	PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
2	0	5	0	0	
3	0	10	5	10	
4	0	9	15	19	
5	0	3	24	22	
9	0	6	27	28	
10	0	8	33	36	
7	1	2	41	38	
8	1	2	43	40	
1	1	2	45	42	
6	1	6	47	48	
Average Waiting Time: 28.000000					

	A	B	C	D	E	F
1	Process No	Queue No	Burst Time	Wait Time	Turn Around Time	
2	2	0	5	0	0	
3	3	0	10	5	10	
4	4	0	9	15	19	
5	5	0	3	24	22	
6	9	0	6	27	28	
7	10	0	8	33	36	
8	7	1	2	41	38	
9	8	1	2	43	40	
10	1	1	2	45	42	
11	6	1	6	47	48	
12						
13						
14						
15	Average Waiting Time					
16	280					
17						

7) Multilevel Feedback Queue Scheduling

A Multi-Level Feedback Queue (MLFQ) is a scheduling algorithm used in operating systems for managing the execution of processes on a CPU. It is designed to provide efficient scheduling by dynamically adjusting the priority of processes based on their behavior and resource requirements.

Implementation:

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

struct process{
    char name;
    int AT,BT,WT,TAT,RT,CT;
}Q1[10],Q2[10],Q3[10];
```

```

int n;
void sortByArrival(){
    struct process temp;
    int i,j;
    for(int i=0;i<n;i++){
        for(int j=i+1;j<n;j++){
            if(Q1[i].AT>Q1[j].AT){
                temp=Q1[i];
                Q1[i]=Q1[j];
                Q1[j]=temp;
            }
        }
    }
}

int main(){
    int j,k=0;
    int r=0;
    int time=0;
    int tq1,tq2,flag=0;
    char c;
    tq1 = (rand() % (10 - 2 + 1)) + 2;
    tq2 = (rand() % (10 - 2 + 1)) + 2;
    printf("Enter the no of processes: ");
    scanf("%d",&n);

    int arrival_upper = 10;
    int arrival_lower = 1;
    int burst_upper = 10;
    int burst_lower = 1;

    for(int i=0;i<n;i++){
        Q1[i].name=i;

```

```

        Q1[i].AT=(rand() % (arrival_upper - arrival_lower + 1)) +
arrival_lower;
        Q1[i].BT=(rand() % (burst_upper - burst_lower + 1)) + burst_lower;
        Q1[i].RT=Q1[i].BT;
    }

    sortByArrival();
    time=Q1[0].AT;
    printf("Process in first queue following RR with qt=%d",tq1);
    printf("\nProcess\t\tRT\t\tWT\t\tTAT\t\t");
    for(int i=0;i<n;i++){
        if(Q1[i].RT<=tq1){
            time+=Q1[i].RT;
            Q1[i].RT=0;
            Q1[i].WT=time-Q1[i].AT-Q1[i].BT;
            Q1[i].TAT=time-Q1[i].AT;
            printf("\n%d\t\t%d\t\t%d\t\t%d",Q1[i].name,Q1[i].BT,Q1[i].W
T,Q1[i].TAT);
        }
        else{
            Q2[k].WT=time;
            time+=tq1;
            Q1[i].RT-=tq1;
            Q2[k].BT=Q1[i].RT;
            Q2[k].RT=Q2[k].BT;
            Q2[k].name=Q1[i].name;
            k=k+1;
            flag=1;
        }
    }
    if(flag==1){
        printf("\nProcess in second queue following RR with qt %d",tq2);
        printf("\nProcess\t\tRT\t\tWT\t\tTAT\t\t");
    }

```



```

for(int i=0;i<k;i++){
    if(Q2[i].RT<=tq2){
        time+=Q2[i].RT;
        Q2[i].RT=0;
        Q2[i].WT=time-tq1-Q2[i].BT;
        Q2[i].TAT=time-Q2[i].AT;
        printf("\n%d\t\t%d\t\t%d\t\t%d",Q2[i].name,Q2[i].BT,Q2[i].W
T,Q2[i].TAT);
    }
    else{
        Q3[r].AT=time;
        time+=tq2;
        Q2[i].RT-=tq2;
        Q3[r].BT=Q2[i].RT;
        Q3[r].RT=Q3[r].BT;
        Q3[r].name=Q2[i].name;
        r=r+1;
        flag=2;
    }
}
if(flag==2){
    printf("\nProcess in third queue following FCFS ");
}
for(int i=0;i<r;i++){
    if(i==0){
        Q3[i].CT=Q3[i].BT+time-tq1-tq2;
    }
    else{
        Q3[i].CT=Q3[i-1].CT+Q3[i].BT;
    }
}
for(int i=0;i<r;i++){
    Q3[i].TAT=Q3[i].CT;
    Q3[i].WT=Q3[i].TAT-Q3[i].BT;
}

```

```

        printf("\n%c\t\t%d\t\t%d\t\t%d\t\t",Q3[i].name,Q3[i].BT,Q3[i]
.WT,Q3[i].TAT);
    }
}

```

Output:

```

Enter the no of processes: 10
Process in first queue following RR with qt=7
Process      RT      WT      TAT
6            2       7       9
9            5       9      14
3            5      13      18
0            1      16      17
7            3      16      19
4            6      19      25
8            7      23      30
1            5      35      40
Process in second queue following RR with qt 10
Process      RT      WT      TAT
5            1      43      51
2            2      44      53

```

Deadlocks

Q) What is a Deadlock?

A deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process. It is a situation where two or more processes are stuck in a circular waiting state.

Deadlock can arise if the following four conditions hold simultaneously:

- i) **Mutual Exclusion:** Two or more resources are non-shareable.
- ii) **Hold and Wait:** A process is holding at least one resource and waiting for resources.
- iii) **No Preemption:** A resource cannot be taken from a process unless the process releases the resource.
- iv) **Circular Wait:** A set of processes waiting for each other in circular form.

Detecting and correcting deadlocks are essential facets of operating system design to prevent system-wide disruptions and resource wastage.

Q) What are some methods of Deadlock Detection?

To fix a deadlock, we first need to identify it. Here are some ways operating systems can detect a deadlock:

1) Resource Allocation Graph (RAG):

- One of the primary methods for deadlock detection is the Resource Allocation Graph (RAG).
- Processes are represented as nodes, and resource instances are represented as edges.
- Deadlocks are detected by identifying cycles in the graph. If a cycle exists, it indicates a potential deadlock.

2) Wait-for Graph:

- Similar to RAG, the Wait-for Graph is another technique for deadlock detection.
- Processes are represented as nodes, and edges represent processes waiting for resources held by other processes.
- Cycles in this graph indicate potential deadlocks.

3) Banker's Algorithm:

- While primarily a deadlock avoidance technique, the Banker's algorithm can also be adapted for deadlock detection.
- It simulates resource allocation to determine if a system state could lead to deadlock.

4) Timeout Mechanisms:

- Systems may employ timeout mechanisms to detect potential deadlocks.
- If a process waits for a resource for longer than a specified timeout period, it may be considered deadlocked.

5) Resource-Request Algorithm:

- Periodically, the system checks the status of processes and their resource requests.

- If a process is unable to acquire all resources it needs, the system investigates whether this could lead to deadlock.

Q) What is Deadlock correction? Give some methods.

Deadlock correction refers to the actions taken by an operating system or system administrator to resolve a deadlock situation once it has been detected.

Some methods for deadlock correction are:

1) Process Termination:

- One approach to correcting deadlocks is to terminate one or more processes involved in the deadlock.
- Processes may be terminated based on criteria such as priority or resource usage.

2) Resource Preemption:

- Resources may be preempted from one or more processes to break the deadlock.
- The preempted resources are then allocated to other processes to allow progress.

3) Rollback:

- In systems with transactional processing, a rollback mechanism can be used to revert the state of processes to a point before the deadlock occurred.
- This allows processes to retry their operations without encountering the deadlock.

4) Recovery and Restart:

- In distributed systems, recovery mechanisms may involve restarting failed processes or reallocating resources to restore system functionality.
- This may require coordination among multiple nodes in the system.

5) Deadlock Avoidance:

- While not a direct correction mechanism, designing systems to avoid deadlocks in the first place is often preferred.
- Techniques such as resource allocation policies, deadlock prevention algorithms, and careful system design can help minimize the occurrence of deadlocks.

Q) What is Banker's Algorithm?

The Banker's Algorithm is a resource allocation and deadlock avoidance algorithm used in operating systems to manage the allocation of multiple resources among multiple processes in a way that prevents deadlock and ensures system safety. It is also used to detect deadlocks in a system.

Implementation:

```
#include<stdio.h>

static int mark[20];
int i, j, np, nr;

int main() {
    int alloc[10][10], request[10][10], avail[10], r[10], w[10];

    printf("\nEnter the number of processes: ");
```

```

scanf("%d", &np);
printf("\nEnter the number of resources: ");
scanf("%d", &nr);

for(i = 0; i < nr; i++) {
    printf("\nTotal Amount of the Resource R%d: ", i + 1);
    scanf("%d", &r[i]);
}

printf("\nEnter the request matrix:");
for(i = 0; i < np; i++)
    for(j = 0; j < nr; j++)
        scanf("%d", &request[i][j]);

printf("\nEnter the allocation matrix:");
for(i = 0; i < np; i++)
    for(j = 0; j < nr; j++)
        scanf("%d", &alloc[i][j]);

for(j = 0; j < nr; j++) {
    avail[j] = r[j];
    for(i = 0; i < np; i++) {
        avail[j] -= alloc[i][j];
    }
}

for(i = 0; i < np; i++) {
    int count = 0;
    for(j = 0; j < nr; j++) {
        if(alloc[i][j] == 0)
            count++;
        else
            break;
    }
}

```

```

        if(count == nr)
            mark[i] = 1;
    }

    for(j = 0; j < nr; j++)
        w[j] = avail[j];

    for(i = 0; i < np; i++) {
        int canbeprocessed = 0;
        if(mark[i] != 1) {
            for(j = 0; j < nr; j++) {
                if(request[i][j] <= w[j])
                    canbeprocessed = 1;
                else {
                    canbeprocessed = 0;
                    break;
                }
            }
            if(canbeprocessed) {
                mark[i] = 1;
                for(j = 0; j < nr; j++)
                    w[j] += alloc[i][j];
            }
        }
    }

    int deadlock = 0;
    for(i = 0; i < np; i++)
        if(mark[i] != 1)
            deadlock = 1;

    if(deadlock)
        printf("\nDeadlock detected");
    else

```



```
        printf("\nNo Deadlock possible");  
  
    return 0;  
}
```

Output:

```
Enter the number of processes: 3  
Enter the number of resources: 3  
Total Amount of the Resource R1: 3  
Total Amount of the Resource R2: 2  
Total Amount of the Resource R3: 1  
  
Enter the request matrix:1 0 2  
3 2 1  
1 2 4  
  
Enter the allocation matrix:1 0 0  
0 1 0  
0 0 0  
  
Deadlock detected
```

Critical Section Problem

Q) What is the Critical Section Problem?

The Critical Section Problem (CSP) is a fundamental challenge in concurrent programming where multiple processes or threads share a common resource and need to ensure that only one process can access the resource at a time.

The critical section refers to the segment of code or region of execution where a process accesses or modifies shared resources. The goal of solving the Critical Section Problem is to guarantee mutual exclusion, ensuring that only one process executes its critical section at any given time, thereby preventing data corruption or inconsistency.

Q) What is Peterson's solution to CSP?

Peterson's solution is a classic algorithm used to address the Critical Section Problem (CSP) in concurrent programming. It is applicable for only two processes.

Peterson's solution provides a mechanism for mutual exclusion, ensuring that only one process can execute its critical section at any given time.

It uses two key concepts:

- i) Flags:** Each process maintains a boolean flag indicating its intention to enter the critical section. If a process wants to enter the critical section, it sets its flag to true.

ii) Turn Variable: This variable indicates whose turn it is to enter the critical section. It is usually an integer that alternates between the IDs of the processes.

Implementation:

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

bool flag[2] = {false, false};
int turn = 0;

void process(int id){
    int other = 1 - id;
    flag[id] = true;
    turn = id;
    while (flag[other] && turn == id);
    // Critical Section
    printf("Process %d is in the critical section.\n", id);
    flag[id] = false;
    // Remainder Section
    printf("Process %d is in the remainder section.\n", id);
}

int main(){
    process(0);
    process(1);
    return 0;
}
```

Output:

```
Process 0 is in the critical section.  
Process 0 is in the remainder section.  
Process 1 is in the critical section.  
Process 1 is in the remainder section.
```

Q) What is a Semaphore? How is it used to solve CSP?

A semaphore is a synchronization primitive used in concurrent programming to control access to a shared resource by multiple processes or threads. It essentially acts as a counter that helps regulate access to a critical section of code or a shared resource. Semaphores can be thought of as a variable that is modified through two atomic operations: wait (decrement) and signal (increment).

Wait Operation (P operation): Decrements the semaphore's value. If the semaphore's value becomes negative, the process or thread attempting the wait operation is blocked until the semaphore's value becomes non-negative.

Signal Operation (V operation): Increments the semaphore's value. If there are processes or threads waiting on the semaphore, one of them is unblocked.

Binary Semaphore – Implementation:

```
#include <stdio.h>  
#include <pthread.h>  
  
typedef struct {  
    pthread_mutex_t mutex;  
    int value;  
} BinarySemaphore;
```

```

void binary_semaphore_init(BinarySemaphore *semaphore) {
    pthread_mutex_init(&semaphore->mutex, NULL);
    semaphore->value = 1;
}

void binary_semaphore_acquire(BinarySemaphore *semaphore) {
    pthread_mutex_lock(&semaphore->mutex);
    while (semaphore->value == 0) {
        pthread_mutex_unlock(&semaphore->mutex);
        pthread_mutex_lock(&semaphore->mutex);
    }
    semaphore->value = 0;
    pthread_mutex_unlock(&semaphore->mutex);
}

void binary_semaphore_release(BinarySemaphore *semaphore) {
    pthread_mutex_lock(&semaphore->mutex);
    semaphore->value = 1;
    pthread_mutex_unlock(&semaphore->mutex);
}

int main() {
    BinarySemaphore semaphore;
    binary_semaphore_init(&semaphore);
    printf("Acquiring semaphore...\n");
    binary_semaphore_acquire(&semaphore);
    printf("Semaphore acquired. Performing some work...\n");
    printf("Releasing semaphore...\n");
    binary_semaphore_release(&semaphore);
    printf("Semaphore released.\n");
    return 0;
}

```

Output:

```
Acquiring semaphore...
Semaphore acquired. Performing some work...
Releasing semaphore...
Semaphore released.
```

Counting Semaphore – Implementation:

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

typedef struct {
    int value;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} CountingSemaphore;

void counting_semaphore_init(CountingSemaphore *sem, int
initial_value) {
    sem->value = initial_value;
    pthread_mutex_init(&sem->mutex, NULL);
    pthread_cond_init(&sem->cond, NULL);
}

void counting_semaphore_wait(CountingSemaphore *sem) {
    pthread_mutex_lock(&sem->mutex);
    while (sem->value <= 0) {
        pthread_cond_wait(&sem->cond, &sem->mutex);
    }
    sem->value--;
    pthread_mutex_unlock(&sem->mutex);
}
```

```

void counting_semaphore_signal(CountingSemaphore *sem) {
    pthread_mutex_lock(&sem->mutex);
    sem->value++;
    pthread_cond_signal(&sem->cond);
    pthread_mutex_unlock(&sem->mutex);
}

#define wait(counting_semaphore) counting_semaphore_wait
(&counting_semaphore)
#define signal(counting_semaphore) counting_semaphore_signal
(&counting_semaphore)

void *thread_function(void *arg) {
    CountingSemaphore *sem = (CountingSemaphore *)arg;
    int thread_id = *((int *)arg + 1);

    wait(*sem);
    printf("Thread %d entered critical section\n", thread_id);
    // critical section
    printf("Thread %d exited critical section\n", thread_id);
    signal(*sem);

    free(arg);
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[5];
    CountingSemaphore semaphore;
    counting_semaphore_init(&semaphore, 2);

    for (int i = 0; i < 5; ++i) {
        int *arg = malloc(2 * sizeof(int));
        arg[0] = (int)&semaphore;
    }
}

```

```
    arg[1] = i;
    pthread_create(&threads[i], NULL, thread_function, arg);
}

for (int i = 0; i < 5; ++i) {
    pthread_join(threads[i], NULL);
}
return 0;
}
```

Output:

```
Thread 0 entered critical section
Thread 0 exited critical section
Thread 1 entered critical section
Thread 1 exited critical section
Thread 3 entered critical section
Thread 3 exited critical section
Thread 2 entered critical section
Thread 2 exited critical section
Thread 4 entered critical section
Thread 4 exited critical section
```

Q) Explain the Dining Philosophers Problem.

The Dining Philosophers Problem is a synchronization problem that illustrates challenges with resource allocation and deadlock avoidance in concurrent systems. The problem is framed around a scenario where a group of philosophers sits around a dining table, with a bowl of rice in front of each philosopher and a single chopstick placed between each pair of adjacent philosophers.

The rules for the Dining Philosophers Problem are:

- Each philosopher must alternately think and eat.

- To eat, a philosopher needs to pick up both chopsticks adjacent to them.
- Only one philosopher can use a chopstick at a time.
- A philosopher cannot eat if they are holding only one chopstick.
- Philosophers are free to think as long as they wish, regardless of whether they have both chopsticks.

Implementation:

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

#define NUM_PHILOSOPHERS 3
#define NUM_ITERATIONS 3

int chopsticks[NUM_PHILOSOPHERS] = {1, 1, 1};

void *philosopher_function(void *arg) {
    int philosopher_id = *((int *)arg);

    for (int iteration = 0; iteration < NUM_ITERATIONS; ++iteration) {
        printf("Philosopher %d: Iteration %d\n", philosopher_id, iteration +
1);
        int left = philosopher_id;
        int right = (philosopher_id + 1) % NUM_PHILOSOPHERS;

        printf("Philosopher %d is thinking\n", philosopher_id);

        while (!(chopsticks[left] && chopsticks[right])) {
```

```

        printf("Philosopher %d cannot eat, continuing to think\n",
philosopher_id);
        usleep(1000000);
    }

    chopsticks[left] = 0;
    chopsticks[right] = 0;

    printf("Philosopher %d is eating\n", philosopher_id);

    chopsticks[left] = 1;
    chopsticks[right] = 1;

    usleep(1000000);
}

pthread_exit(NULL);
}

int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int philosopher_ids[NUM_PHILOSOPHERS];

    for (int i = 0; i < NUM_PHILOSOPHERS; ++i) {
        philosopher_ids[i] = i;
        pthread_create(&philosophers[i],    NULL,    philosopher_function,
&philosopher_ids[i]);
    }

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

    return 0;
}

```

}

Output:

```
Philosopher 1: Iteration 1
Philosopher 1 is thinking
Philosopher 1 is eating
Philosopher 0: Iteration 1
Philosopher 0 is thinking
Philosopher 0 is eating
Philosopher 2: Iteration 1
Philosopher 2 is thinking
Philosopher 2 is eating
Philosopher 1: Iteration 2
Philosopher 1 is thinking
Philosopher 1 is eating
Philosopher 0: Iteration 2
Philosopher 0 is thinking
Philosopher 0 is eating
Philosopher 2: Iteration 2
Philosopher 2 is thinking
Philosopher 2 is eating
Philosopher 1: Iteration 3
Philosopher 1 is thinking
Philosopher 1 is eating
Philosopher 2: Iteration 3
Philosopher 2 is thinking
Philosopher 2 is eating
Philosopher 0: Iteration 3
Philosopher 0 is thinking
Philosopher 0 is eating
```

Disk Scheduling :

Disk scheduling is done by operating systems to schedule I/O requests arriving for the disk. Disk scheduling is also known as I/O Scheduling.

Importance of Disk Scheduling in Operating System

- Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by the disk controller. Thus other I/O requests need to wait in the waiting queue and need to be scheduled.
- Two or more requests may be far from each other so this can result in greater disk arm movement.
- Hard drives are one of the slowest parts of the computer system and thus need to be accessed in an efficient manner.

Disk Scheduling Algorithms

FCFS (First Come First Serve)

SSTF (Shortest Seek Time First)

SCAN (Elevator Algorithm)

C-SCAN (Circular SCAN)

LOOK

C-LOOK

FCFS :

FCFS is the simplest of all Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue. Let us understand this with the help of an example.

```
#include <iostream>
#include <vector>
using namespace std;
int FCFS(vector<int> requests, int head){
    int n = requests.size();
    int dist = 0;
    int noOfTurns = 0;
    int totalDist = 0;
    int start = head;
    for(int i=0;i<n;i++){
        dist = requests[i] - start;
        if(dist<0){
            dist *= -1;
            noOfTurns++;
        }
        start = requests[i];
        totalDist += dist;
    }
    return totalDist;
}

int main(){
    vector<int> v{ 176, 79, 34, 60, 92, 11, 41, 114 };
    int head = 50;
    printf("%d ", FCFS(v, head));
}
```

Output:

```

510

```

SSTF:

In SSTF (Shortest Seek Time First), requests having the shortest seek time are executed first. So, the seek time of every request is calculated in advance in the queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of the system. Let us understand this with the help of an example.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

```
int absv(int x){
    if(x<0){
        x = -1*x;
    }
    return x;
}
```

```
int return_min_req(int head, vector<int> requests, vector<bool> &fulfilled){
    int min_req = 0;
    int min_dist = absv(head-requests[min_req]);
    int dist = 0;
    int flag = 0;
    for(int i=0;i<requests.size();i++){
```

```

        if(fulfilled[i]==false){
            flag = 1;
            dist = absv(head - requests[i]);
            if(dist<min_dist){
                min_req = i;
                min_dist = dist;
            }
        }
    }
    if(flag == 0){
        return -1; //all requests are fulfilled
    }
    else{
        fulfilled[min_req] = true;
        return min_req;
    }
}

void SSTF(int head, vector<int> requests){
    int curr_req;
    vector<bool> fulfilled(requests.size(), false);
    int dist = 0;
    int no_of_dir_changes = 0;
    for(int i=0;i<requests.size();i++){
        curr_req = return_min_req(head, requests, fulfilled);
        dist += absv(head - requests[curr_req]);
        head = requests[curr_req];
        cout<<"CURR HEAD: "<<head<<" ";
        cout<<"REQUEST FULFILLED: "<<curr_req<<" "<<requests[curr_req]<<"
"<<dist<<endl;
    }
    cout<<"DISTANCE: "<<dist<<endl;
}

```

```

int main(){
    vector<int> requests;
    int head;
    cout<<"Enter the position of the head: ";
    cin>>head;
    cout<<"Enter the number of requests: ";
    int n;
    cin>>n;
    cout<<"Enter the requests: ";
    int x;
    for(int i=0;i<n;i++){
        cout<<"Enter: ";
        cin>>x;
        requests.push_back(x);
    }
    SSTF(head, requests);
}

```

Output:

```

orithm\" ; if ($?) { g++ Shortest_Seek_Time_Schediling.cpp -o Shortest_Seek_Time_Schediling } ;
Enter the position of the head: 50
Enter the number of requests: 3
Enter the requests: Enter: 100
Enter: 40
Enter: 130
CURR HEAD: 40 REQUEST FULFILLED: 1 40 10
CURR HEAD: 100 REQUEST FULFILLED: 0 100 70
CURR HEAD: 100 REQUEST FULFILLED: 0 100 70
DISTANCE: 70
PS C:\Users\Archit Chhajer\OneDrive\Desktop\Operating_System\lab\Disk_Scheduling_Algorithm>

```

SCAN:

In the SCAN algorithm the disk arm moves in a particular direction and services the requests coming in its path and after reaching the end of the disk, it reverses

its direction and again services the request arriving in its path. So, this algorithm works as an elevator and is hence also known as an elevator algorithm. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int direction(int head, int x){
    return head - x;
}

int absv(int x){
    if(x<0){
        return -1*x;
    }
    return x;
}

void procedure_SCAN(int head, vector<int> &first, vector<int> &second);

void printArr(vector<int> x){
    for(int i=0;i<x.size();i++){
        cout<<x[i]<<" ";
    }
    cout<<endl;
}

void SCAN(int head, int dir, vector<int> &req){
    vector<bool> vis(req.size(), false);
```

```

sort(req.begin(), req.end() );
cout<<"REQUEST ARRAY: ";
printArr(req);

int start_req = 0;
int start_req_index = 0;
for(int i=0;i<req.size();i++){
    if(req[i]>head){

        start_req = i;
        start_req_index = req[i];
        break;
    }
}
if(dir<0){
    start_req -= 1;
}

int index = start_req;
int total_dist = 0;
while(index>=0 && index<req.size()){
    total_dist += absv(head - req[index]);
    vis[index] = true;
    cout<<"HEAD POS: "<<head<<" REQ NO FULFILLED: "<<index<<"
REQUEST VALUE: "<<req[index]<<" DISTANCE: "<<absv(head-
req[index])<<"TOTAL DIST: "<<total_dist<<endl;
    head = req[index];
    index += dir;
}

if(dir>0){
    total_dist += head - 199;
    cout<<"HEAD IS NOT AT 199: "<<" DIST: "<< (head-199) << "TOTAL DIST:
"<<total_dist<<endl;

```

```

        head = 199;
    }
    else{
        total_dist += head - 0;
        cout<<"HEAD IS NOT AT 0: "<<" DIST: "<< (head-0) << "TOTAL DIST:
"<<total_dist<<endl;
        head = 0;
    }

    index = start_req;
    if(dir<0){
        index = index+1;
    }

    // 176 79 34 60 92 11 41 114
    while(index>=0 && index<req.size()){
        total_dist += absv(head-req[index]);
        vis[index] = true;
        cout<<"HEAD POS: "<<head<<" REQ NO FULFILLED: "<<index<<"
REQUEST VALUE: "<<req[index]<<" DISTANCE: "<<absv(head-
req[index])<<"TOTAL DISTANCE: "<<total_dist<<endl;
        head = req[index];
        index += (-1)*dir;
    }

    cout<<"Total distance covered by the disk pointer on the track
:"<<total_dist<<endl;

}

```

```

int main(){
    int head;
    vector<int> requests;

```

```

cout<<"Enter the position of the head: ";
cin>>head;
cout<<"Enter the initial direction (1 for positive, -1 for negative): "<<endl;
int dir;
cin>>dir;
cout<<"Enter the number of requests:";
int n;
cin>>n;
int x;
for(int i=0;i<n;i++){
    cout<<"Enter: ";
    cin>>x;
    requests.push_back(x);
}
SCAN(head, dir, requests);
}

```

Output:

```

CAN.cpp -o SCAN } ; if ($?) { ./SCAN }
Enter the position of the head: 50
Enter the initial direction (1 for positive, -1 for negative):
1
Enter the number of requests:3
Enter: 100
Enter: 40
Enter: 130
REQUEST ARRAY: 40 100 130
HEAD POS: 50 REQ NO FULFILLED: 1 REQUEST VALUE: 100 DISTANCE: 50TOTAL DIST: 50
HEAD POS: 100 REQ NO FULFILLED: 2 REQUEST VALUE: 130 DISTANCE: 30TOTAL DIST: 80
HEAD IS NOT AT 199: DIST: -69TOTAL DIST: 11
HEAD POS: 199 REQ NO FULFILLED: 1 REQUEST VALUE: 100 DISTANCE: 99TOTAL DISTANCE: 110
HEAD POS: 100 REQ NO FULFILLED: 0 REQUEST VALUE: 40 DISTANCE: 60TOTAL DISTANCE: 170
Total distance covered by the disk pointer on the track :170

```

C-SCAN:

In the SCAN algorithm, the disk arm again scans the path that has been scanned, after reversing its direction. So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area.

These situations are avoided in the CSCAN algorithm in which the disk arm instead of reversing its direction goes to the other end of the disk and starts servicing the requests from there. So, the disk arm moves in a circular fashion and this algorithm is also similar to the SCAN algorithm hence it is known as C-SCAN (Circular SCAN).

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int absv(int x){
    if(x<0){
        return -1*x;
    }
    return x;
}

void printArr(vector<int> a){
    cout<<"ARR: ";
    for(int i=0;i<a.size();i++){
        cout<<a[i]<<" ";
    }
    cout<<endl;
}

int binSearch(vector<int> a, int x){
    int start = 0;
    int mid;
    int end = a.size()-1;
    while(start<=end){
        mid = (start+end)/2;
        if(a[mid]==x){
```

```

        return mid;
    }
    else if(a[mid]>x){
        end = mid-1;
    }
    else{
        start = mid + 1;
    }
}
return -1;
}

```

```

int CSCAN(int head, int dir, vector<int> req){
    req.push_back(head);
    sort(req.begin(), req.end());
    printArr(req);
    int startPos = binSearch(req, head);
    cout<<"START POS: "<<startPos<<endl;

```

```

    int index = startPos;
    index = index + dir;
    int total_dist = 0;
    cout<<"HEAD INITALLY AT: "<<head<<endl;
    while(index>=0 && index<req.size()){
        total_dist += absv(head - req[index]);
        cout<<"HEAD AT: "<<head<<"NODE VISITED: "<<req[index]<<" DIST:
"<<absv(head-req[index])<<endl;
        head = req[index];
        index += dir;
    }

```

```

    if(dir>0){
        total_dist += absv(head - 199);
    }

```

```

        cout<<"HEAD IS NOT AT 199: "<<" DIST: "<<(head-199) <<"TOTAL DIST:
"<<total_dist<<endl;
        head = 199;
        total_dist += 199;
        cout<<"HEAD HAS NOW MOVED TO 0"<<"DIST: "<<(199)<<"TOTAL DIST:
"<<total_dist<<endl;
        head = 0;
        index = 0;
    }
    else{
        total_dist += head - 0;
        cout<<"HEAD IS NOT AT 0: "<<" DIST: "<<(head-0) <<"TOTAL DIST:
"<<total_dist<<endl;
        head = 0;
        total_dist += 199;
        cout<<"HEAD HAS NOW MOVED TO 199"<<"DIST: "<<(199)<<"TOTAL
DIST: "<<total_dist<<endl;
        head = 199;
        index = req.size()-1;
    }

    while(index!=startPos){
        total_dist += absv(head - req[index]);
        cout<<"HEAD AT: "<<head<<"NODE VISITED: "<<req[index]<<" DIST
COVERED: "<<absv(head - req[index])<<endl;
        head = req[index];
        index += dir;
    }

    cout<<"TOTAL DISTANCE COVERED: "<<total_dist<<endl;
    return total_dist;
}

int main(){
    vector<int> requests;

```

```

cout<<"Enter the head position: "<<endl;
int head;
cin>>head;
cout<<"Enter the initial direction: "<<endl;
int dir;
cin>>dir;
cout<<"Enter the number of requests:"<<endl;
int n;
cin>>n;
cout<<"Enter the elements: "<<endl;
int x;
for(int i=0;i<n;i++){
    cout<<"Enter: "<<endl;
    cin>>x;
    requests.push_back(x);
}
CSCAN(head, dir, requests);
}

```

Output:

```

orithm\" ; if ($?) { g++ CSCAN.cpp -o CSCAN } ; if ($?) { .\CSCAN }
Enter the head position:
50
Enter the initial direction:
1
Enter the number of requests:
3
Enter the elements:
Enter:
40
Enter:
100
Enter:
130
ARR: 40 50 100 130
START POS: 1
HEAD INITIALLY AT: 50
HEAD AT: 50 NODE VISITED: 100 DIST: 50
HEAD AT: 100 NODE VISITED: 130 DIST: 30
HEAD IS NOT AT 199: DIST: -69 TOTAL DIST: 149
HEAD HAS NOW MOVED TO 0 DIST: 199 TOTAL DIST: 348
HEAD AT: 0 NODE VISITED: 40 DIST COVERED: 40
TOTAL DISTANCE COVERED: 388

```

LOOK:

LOOK Algorithm is similar to the SCAN disk scheduling algorithm except for the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

```
int absv(int x){
    if(x<0){
        return -1*x;
    }
    return x;
}
```

```
int binSearch(vector<int>& a, int x){
    int start = 0;
    int mid;
    int end = a.size()-1;
    while(start<=end){
        mid = (start+end)/2;
        if(a[mid]==x){
            return mid;
        }
        else if(a[mid]>x){
            end = mid-1;
        }
        else{
            start = mid + 1;
        }
    }
}
```

```

    }
}
return -1;
}

```

```

void printArr(vector<int> a){
    cout<<"ARR: ";
    for(int i=0;i<a.size();i++){
        cout<<a[i]<<" ";
    }
    cout<<endl;
}

```

```

int LOOK(int head,int dir, vector<int> req){
    req.push_back(head);
    sort(req.begin(), req.end());
    printArr(req);
    int startPos = binSearch(req, head);
    cout<<"START POS: "<<startPos<<endl;

```

```

    int index = startPos;
    index = index + dir;
    int total_dist = 0;
    cout<<"HEAD INITALLY AT: "<<head<<endl;
    while(index>=0 && index<req.size()){
        total_dist += absv(head - req[index]);
        cout<<"HEAD AT: "<<head<<"NODE VISITED: "<<req[index]<<" DIST:
"<<absv(head-req[index])<<endl;
        head = req[index];
        index += dir;
    }

    index = startPos;

```

```

    index += (-1)*dir;
    while(index>=0 && index<req.size()){
        total_dist += absv(head - req[index]);
        cout<<"HEAD AT: "<<head<<"NODE VISITED: "<<req[index]<<" DIST
COVERED: "<<absv(head - req[index])<<endl;
        head = req[index];
        index += (-1)*dir;
    }

    cout<<"TOTAL DISTANCE COVERED: "<<total_dist<<endl;
    return total_dist;

}

int main(){
    vector<int> requests;
    cout<<"Enter the head position: "<<endl;
    int head;
    cin>>head;
    cout<<"Enter the initial direction: "<<endl;
    int dir;
    cin>>dir;
    cout<<"Enter the number of requests:"<<endl;
    int n;
    cin>>n;
    cout<<"Enter the elements: "<<endl;
    int x;
    for(int i=0;i<n;i++){
        cout<<"Enter: "<<endl;
        cin>>x;
        requests.push_back(x);
    }
    LOOK(head, dir, requests);
}

```

Output:

```
Enter the position of the head: 50
Enter the initial direction (1 for positive, -1 for negative):
1
Enter the number of requests:3
Enter: 100
Enter: 40
Enter: 130
REQUEST ARRAY: 40 100 130
HEAD POS: 50 REQ NO FULFILLED: 1 REQUEST VALUE: 100 DISTANCE: 50TOTAL DIST: 50
HEAD POS: 100 REQ NO FULFILLED: 2 REQUEST VALUE: 130 DISTANCE: 30TOTAL DIST: 80
HEAD IS NOT AT 199: DIST: -69TOTAL DIST: 11
HEAD POS: 199 REQ NO FULFILLED: 1 REQUEST VALUE: 100 DISTANCE: 99TOTAL DISTANCE: 110
HEAD POS: 100 REQ NO FULFILLED: 0 REQUEST VALUE: 40 DISTANCE: 60TOTAL DISTANCE: 170
Total distance covered by the disk pointer on the track :170
```

CLOOK:

As LOOK is similar to the SCAN algorithm, in a similar way, C-LOOK is similar to the CSCAN disk scheduling algorithm. In CLOOK, the disk arm in spite of going to the end goes only to the last request to be serviced in front of the head and then from there goes to the other end's last request. Thus, it also prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

```
int absv(int x)
{
    if (x < 0)
    {
        return -1 * x;
    }
}
```

```

    return x;
}

void printArr(vector<int> a)
{
    cout << "ARR: ";
    for (int i = 0; i < a.size(); i++)
    {
        cout << a[i] << " ";
    }
    cout << endl;
}

int binSearch(vector<int> a, int x)
{
    int start = 0;
    int mid;
    int end = a.size() - 1;
    while (start <= end)
    {
        mid = (start + end) / 2;
        if (a[mid] == x)
        {
            return mid;
        }
        else if (a[mid] > x)
        {
            end = mid - 1;
        }
        else
        {
            start = mid + 1;
        }
    }
}

```

```

    return -1;
}

int CSCAN(int head, int dir, vector<int> req)
{
    req.push_back(head);
    sort(req.begin(), req.end());
    printArr(req);
    int startPos = binSearch(req, head);
    cout << "START POS: " << startPos << endl;

    int index = startPos;
    index = index + dir;
    int total_dist = 0;
    cout << "HEAD INITALLY AT: " << head << endl;
    while (index >= 0 && index < req.size())
    {
        total_dist += absv(head - req[index]);
        cout << "HEAD AT: " << head << "NODE VISITED: " << req[index] << " DIST: "
        << absv(head - req[index]) << endl;
        head = req[index];
        index += dir;
    }

    if (dir > 0)
    {
        total_dist += req[req.size() - 1] - req[0];
        cout << "HEAD HAS NOW MOVED TO :" << req[0] << " DIST: " <<
        (req[req.size() - 1] - req[0]) << " TOTAL DIST: " << total_dist;

        index = 0;
    }
    else

```

```

    {
        total_dist += req[req.size() - 1] - req[0];
        cout << "HEAD HAS NOW MOVED TO :" << req[req.size() - 1] << " DIST: "
<< (req[req.size() - 1] - req[0]) << " TOTAL DIST: " << total_dist;

        index = req.size() - 1;
    }

    while (index != startPos)
    {
        total_dist += absv(head - req[index]);
        cout << "HEAD AT: " << head << "NODE VISITED: " << req[index] << " DIST
COVERED: " << absv(head - req[index]) << "TOTAL DIST: " << total_dist << endl;
        head = req[index];
        index += dir;
    }

    cout << "TOTAL DISTANCE COVERED: " << total_dist << endl;
    return total_dist;
}

int main()
{
    vector<int> requests;
    cout << "Enter the head position: " << endl;
    int head;
    cin >> head;
    cout << "Enter the initial direction: " << endl;
    int dir;
    cin >> dir;
    cout << "Enter the number of requests:" << endl;
    int n;
    cin >> n;

```

```

    cout << "Enter the elements: " << endl;
    int x;
    for (int i = 0; i < n; i++)
    {
        cout << "Enter: " << endl;
        cin >> x;
        requests.push_back(x);
    }
    CSCAN(head, dir, requests);
}

```

Output:

```

Enter the head position:
50
Enter the initial direction:
1
Enter the number of requests:
3
Enter the elements:
Enter:
40
Enter:
100
Enter:
130
ARR: 40 50 100 130
START POS: 1
HEAD INITALLY AT: 50
HEAD AT: 50 NODE VISITED: 100 DIST: 50
HEAD AT: 100 NODE VISITED: 130 DIST: 30
HEAD HAS NOW MOVED TO :40 DIST: 90 TOTAL DIST: 170 HEAD AT: 130 NODE VISITED: 40 DIST COVERED: 90 TOTAL DIST: 260
TOTAL DISTANCE COVERED: 260

```

THANK YOU...