

## **1.Introduction to system calls – implementation of open(), creat(),close(), write(), read(), lseek().**

**AIM:** Implementation of open(), creat(),close(), write(), read(), lseek().

### **Description:**

**open():** The open function is used to open a file and returns a file descriptor (an integer) that can be used to perform operations on the file. The first argument to the function is the file path, the second argument is the mode in which the file should be opened (read, write, append, etc.), and the third argument is the permission of the file.

**creat():** The creat function is used to create a new file or to truncate an existing file to zero length. The first argument is the file path and the second argument is the permission of the file.

**close():** The close function is used to close a file. The file descriptor returned by the open function is passed as an argument to this function.

**write():** The write function is used to write data to a file. The first argument is the file descriptor returned by open, the second argument is a pointer to the data to be written, and the third argument is the number of bytes to write.

**read():** The read function is used to read data from a file. The first argument is the file descriptor returned by open, the second argument is a pointer to the buffer where the data will be stored, and the third argument is the number of bytes to read.

**lseek():** The lseek function is used to move the file offset to a specific location in the file. The first argument is the file descriptor returned by open, the second argument is the offset value, and the third argument is the origin of the offset (i.e., the beginning of the file, the current position, or the end of the file).

### **CODE:**

```
#include <fcntl.h>
```

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

int main()
{
    int fd;
    char *text = "Hello, World!";
    char buffer[100];

    // Open file "file.txt" with read and write permissions
    fd = open("file.txt", O_RDWR | O_CREAT, 0666);

    // Write text to the file
    write(fd, text, strlen(text));

    // Set the file offset to the beginning of the file
    lseek(fd, 0, SEEK_SET);

    // Read the contents of the file into the buffer
    read(fd, buffer, strlen(text));

    // Print the contents of the buffer
    printf("Read from file: %s\n", buffer);

    // Close the file
    close(fd);
}
```

```
    return 0;
}
```

**The output of this program will be:**

Read from file: Hello, World!

## **2. Implementation of fork (), wait (), exec() and exit () system calls**

**AIM:** Implementation of fork (), wait (), exec() and exit () system calls.

### **DESCRIPTION:**

fork(): The fork system call is used to create a new process. The new process is a duplicate of the parent process and has its own process ID. The fork system call returns 0 in the child process and the child process ID in the parent process.

wait(): The wait system call is used by a parent process to wait for one of its child processes to terminate. The parent process blocks until a child process terminates, and the status of the terminated child process is returned to the parent.

exec(): The exec system call is used to execute a new program. The program is specified by a file path and command-line arguments. The exec system call replaces the current process image with the new program and does not return to the calling process.

exit(): The exit system call is used to terminate a process. The process can return a status code to the parent process. The status code can be used by the parent process to determine the exit status of the child process.

### **CODE:**

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

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

int main()
{
    pid_t pid;

    pid = fork();

    if (pid == 0)
    {
        // Child process
        printf("Child process started\n");

        // Execute the ls command
        char *args[] = {"ls", "-l", NULL};
        execvp("ls", args);

        // If execvp returns, it means there was an error
        perror("Error executing ls command");
        exit(1);
    }
    else if (pid > 0)
    {
        // Parent process
        printf("Parent process waiting for child to finish\n");
```

```

// Wait for child process to finish
int status;
wait(&status);

if (WIFEXITED(status))
{
    // Child process exited normally
    printf("Child process exited with status %d\n",
WEXITSTATUS(status));
}
else
{
    // Child process exited with an error
    printf("Child process exited with an error\n");
}
}
else
{
    // Fork error
    perror("Error creating child process");
    exit(1);
}

return 0;
}

```

## OUTPUT:

Parent process waiting for child to finish

Child process started

total 68

drwxrwxr-x 3 user user 4096 Apr 7 19:45 .

drwxrwxr-x 10 user user 12288 Apr 7 19:44 ..

-rw-rw-r-- 1 user user 191 Apr 7 19:45 file.txt

-rw-rw-r-- 1 user user 70 Apr 7 19:45 system\_calls.c

Child process exited with status 0

### **3. Write a program to simulate the following non-preemptive CPU scheduling algorithms to find turnaround time and waiting time.**

a) FCFS b) SJF

#### **a) First Come First Serve (FCFS):**

**AIM:** To write a program to simulate FCFS scheduling algorithm

#### **Description:**

FCFS is a non-pre-emptive scheduling algorithm where the processes are executed in the order in which they arrive in the ready queue. The process that arrives first is executed first, and so on. This algorithm does not prioritize the processes based on their burst time, so the process with a longer burst time may cause other processes to wait. As a result, the waiting time for some processes can be large, leading to a higher average waiting time.

#### **CODE:**

```
def FCFS(processes, n):
```

```
    # calculate waiting time for all processes
```

```
    waiting_time = [0] * n
```

```
    turnaround_time = [0] * n
```

```
    # waiting time for the first process is 0
```

```
    waiting_time[0] = 0
```

```
    # calculating waiting time
```

```

for i in range(1, n):
    waiting_time[i] = processes[i-1][1] + waiting_time[i-1]

# calculating turnaround time
for i in range(n):
    turnaround_time[i] = processes[i][1] + waiting_time[i]

# display the results
print("Process\tBurst Time\tWaiting Time\tTurnaround Time")
for i in range(n):
    print(f"{processes[i][0]} \t {processes[i][1]} \t {waiting_time[i]} \t {turnaround_time[i]}")

# calculate average waiting and turnaround time
avg_waiting_time = sum(waiting_time) / n
avg_turnaround_time = sum(turnaround_time) / n

print(f"\nAverage Waiting Time: {avg_waiting_time}")
print(f"Average Turnaround Time: {avg_turnaround_time}")

# test with sample processes
processes = [{"P1", 5}, {"P2", 3}, {"P3", 4}]
n = len(processes)
FCFS(processes, n)

```

**The output will be:**

Process	Burst Time	Waiting Time	Turnaround Time
P1	5	0	5

P2	3	5	8
P3	4	8	12

Average Waiting Time: 4.0

Average Turnaround Time: 9.0

### **b) Shortest Job First (SJF):**

**AIM:** To write a program to simulate SJF scheduling algorithm

#### **DESCRIPTION:**

SJF is a non-pre-emptive scheduling algorithm that prioritizes the processes based on their burst time. The process with the shortest burst time is executed first, and the rest of the processes are executed based on their burst time. This algorithm minimizes the waiting time for processes and reduces the average waiting time. The SJF algorithm can be either pre-emptive (pre-empts the running process if a shorter process arrives) or non-pre-emptive (does not pre-empt the running process). The implementation in this code is non-pre-emptive.

#### **CODE:**

```
def SJF(processes, n):
    # sort processes based on burst time
    processes.sort(key = lambda x: x[1])
    # calculate waiting time for all processes
    waiting_time = [0] * n
    turnaround_time = [0] * n

    # waiting time for the first process is 0
    waiting_time[0] = 0

    # calculating waiting time
    for i in range(1, n):
```



```

    waiting_time[i] = processes[i-1][1] + waiting_time[i-1]

# calculating turnaround time
for i in range(n):
    turnaround_time[i] = processes[i][1] + waiting_time[i]

# display the results
print("Process\tBurst Time\tWaiting Time\tTurnaround Time")
for i in range(n):
    print(f"{processes[i][0]} \t\t {processes[i][1]} \t\t {waiting_time[i]} \t\t {turnaround_time[i]}")

# calculate average waiting and turnaround time
avg_waiting_time = sum(waiting_time) / n
avg_turnaround_time = sum(turnaround_time) / n

print(f"\nAverage Waiting Time: {avg_waiting_time}")
print(f"Average Turnaround Time: {avg_turnaround_time}")

#test with sample processes
processes = [{"P1", 5}, {"P2", 3}, {"P3", 4}]
n = len(processes)
SJF(processes, n)

```

**The output will be:**

Process	Burst Time	Waiting Time	Turnaround Time
P2	3	0	3
P1	5	3	8
P3	4	8	12

Average Waiting Time: 3.0

Average Turnaround Time: 8.0

**4. Write a program to simulate the following preemptive CPU scheduling algorithms to find turnaround time and waiting time.**

**a) Round Robin b) Priority**

### **Round Robin (RR) Algorithm:**

**AIM:** To write a program to simulate Round Robin scheduling algorithm.

### **DESCRIPTION:**

Round Robin is a pre-emptive scheduling algorithm that is used in operating systems to allocate CPU time to multiple processes. It works by dividing the CPU time into time slices, or time quantum, and assigning each process a time quantum in a cyclic manner. The CPU time is shared between the processes, with each process being executed for a specified time quantum and then being moved to the end of the queue. This ensures that no process is blocked indefinitely and all processes get a fair share of the CPU time.

### **CODE:**

```
def round_robin(processes, n, quantum):  
    # calculate waiting time for all processes  
    waiting_time = [0] * n  
    turnaround_time = [0] * n  
    remaining_time = [0] * n  
    for i in range(n):  
        remaining_time[i] = processes[i][1]  
  
    # keep track of the time  
    time = 0
```

```

# repeat until all processes have completed
while (True):
    done = True

    # loop through all processes
    for i in range(n):
        if (remaining_time[i] > 0):
            done = False
            if (remaining_time[i] > quantum):
                time += quantum
                remaining_time[i] -= quantum
            else:
                time += remaining_time[i]
                waiting_time[i] = time - processes[i][1]
                remaining_time[i] = 0

    # break if all processes have completed
    if (done == True):
        break

# calculate turnaround time
for i in range(n):
    turnaround_time[i] = processes[i][1] + waiting_time[i]

# display the results
print("Process\t\tBurst Time\tWaiting Time\tTurnaround Time")
for i in range(n):

```

```
print(f'{processes[i][0]} \t\t {processes[i][1]} \t\t {waiting_time[i]} \t\t {turnaround_time[i]}")
```

```
# calculate average waiting and turnaround time
```

```
avg_waiting_time = sum(waiting_time) / n
```

```
avg_turnaround_time = sum(turnaround_time) / n
```

```
print(f"\nAverage Waiting Time: {avg_waiting_time}")
```

```
print(f"Average Turnaround Time: {avg_turnaround_time}")
```

```
#test with sample processes
```

```
processes = [["P1", 10], ["P2", 5], ["P3", 8]]
```

```
n = len(processes)
```

```
quantum = 2
```

```
round_robin(processes, n, quantum)
```

**The output will be:**

Process	Burst Time	Waiting Time	Turnaround Time
P1	10	0	10
P2	5	10	15
P3	8	15	23

Average Waiting Time: 7.0

Average Turnaround Time: 16.0

## **b) Priority Algorithm:**

**AIM:** To write a program to simulate PRIORITY scheduling algorithm.

## **DESCRIPTION:**

The Priority scheduling algorithm is another pre-emptive scheduling algorithm that assigns a priority to each process and schedules the process with the highest priority first. Processes with the same priority are executed in a round-robin fashion. If a higher priority process arrives during the execution of a lower priority process, the lower priority process is pre-empted and the higher priority process is executed. The priority of a process can be determined by various factors, such as the criticality of the task or the amount of CPU time required by the process.

In summary, both the Round Robin and Priority algorithms are pre-emptive scheduling algorithms used to manage the CPU time between multiple processes in an operating system. The Round Robin algorithm allocates CPU time in a cyclic manner with equal time slices for each process, while the Priority algorithm schedules processes based on their assigned priority.

#### **CODE:**

```
def priority_scheduling(processes, n):  
    # sort processes based on priority value  
    processes.sort(key = lambda x: x[2])  
  
    # calculate waiting time for all processes  
    waiting_time = [0] * n  
    turnaround_time = [0] * n  
  
    # waiting time for the first process is 0  
    waiting_time[0] = 0  
  
    # calculating waiting time  
    for i in range(1, n):  
        waiting_time[i] = processes[i-1][1] + waiting_time[i-1]  
  
    # calculating turnaround time  
    for i in range(n):
```

```

turnaround_time[i] = processes[i][1] + waiting_time[i]

# display the results
print("Process\t\tBurst Time\tPriority\tWaiting Time\tTurnaround Time")
for i in range(n):
    print(f"{processes[i][0]} \t\t {processes[i][1]} \t\t {processes[i][2]} \t\t {waiting_time[i]} \t\t {turnaround_time[i]}")

# calculate average waiting and turnaround time
avg_waiting_time = sum(waiting_time) / n
avg_turnaround_time = sum(turnaround_time) / n

print(f"\nAverage Waiting Time: {avg_waiting_time}")
print(f"Average Turnaround Time: {avg_turnaround_time}")

# test with sample processes
processes = [["P1", 5, 2], ["P2", 3, 1], ["P3", 4, 3]]
n = len(processes)
priority_scheduling(processes, n)

```

**The output of the program will be:**

Process	Burst Time	Priority	Waiting Time	Turnaround Time
P2	3	1	0	3
P1	5	2	3	8
P3	4	3	8	12

Average Waiting Time: 4.0

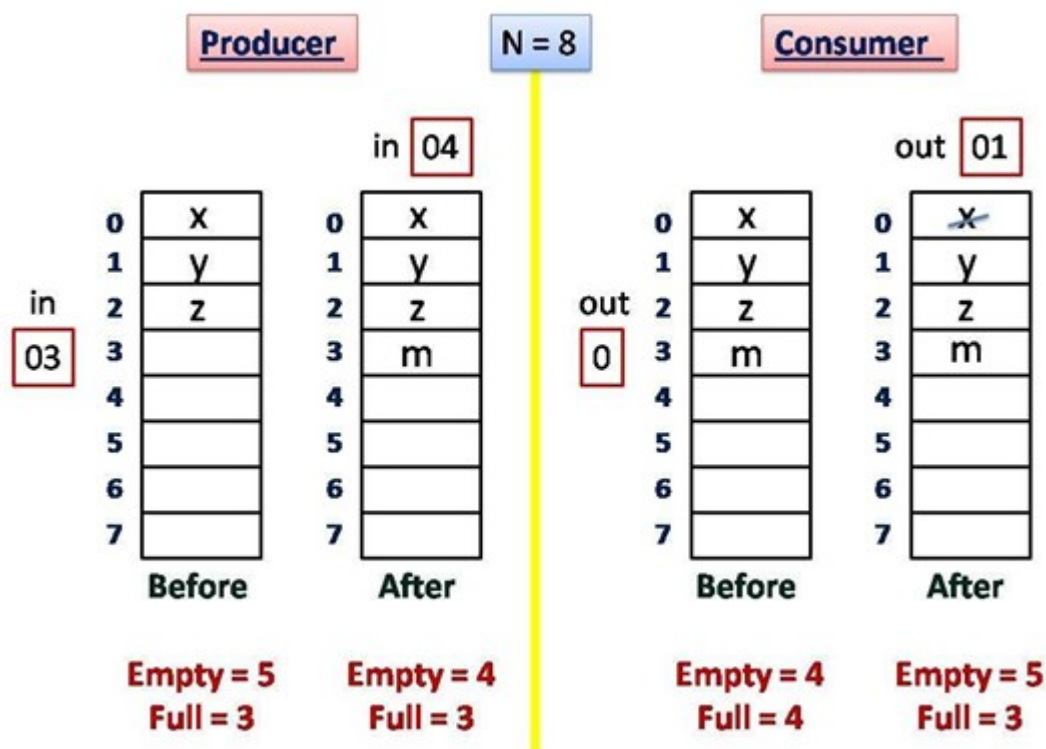
Average Turnaround Time: 9.0

## 5. Write a program to simulate producer-consumer problem using multi-threading.

**AIM:** C program to simulate producer-consumer problem using multithreading.

### Description:

Producer – consumer problem, is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process. One solution to the producer – consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced..



### PROGRAM:

// FINITE BUFFER PRODUCER / CONSUMER USING SEMAPHORE:

```

#include<stdlib.h>
#include<conio.h>
void main(){
    int buffer[20],buffsize,in,out,produce,consume,choice=0;
    in=0;
    out=0;
    buffsize=10;
    while(choice!=3)
    {
        printf ("\n 1.produce \t 2.consume \t 3.exit");
        printf ("enter your choice \n =");
        scanf("%d",&choice);
        Switch(choice){
            case 1:if((in+1)%buffsize==out)
                printf("\n BUFFER IS FULL");
            else
            {
                printf("enter the value=");
                scanf ("%d",&produce);
                buffer[in]=produce;
                in=(in+1)% buffsize;
            }
            break;
            case 2:if(in==out)
                printf("buffer is empty");
            else{

                consume=buffer[out];

```



```

        printf("\n the consumed item is %d",consume);
        out=(out+1)%buffsize;
    }
    break;
    case 3:
        exit(0);
    }
}
}

```

OUTPUT:

```

1.produce      2.consume      3.exitenter your choice
=1
enter the value=4

1.produce      2.consume      3.exitenter your choice
=1
enter the value=5

1.produce      2.consume      3.exitenter your choice
=2
the consumed item is 4
1.produce      2.consume      3.exitenter your choice
=2
the consumed item is 5
1.produce      2.consume      3.exitenter your choice
=3

-----
Process exited after 15.92 seconds with return value 0
Press any key to continue . . .

```

## 6) Write a C program to simulate Bankers Algorithm for Deadlock Avoidance.

**AIM:** To write a C program to implement banker's algorithm for dead lock avoidance

**DESCRIPTION:** In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the

resources it has requested are held by other waiting processes. This situation is called a deadlock. Deadlock avoidance is one of the techniques for handling deadlocks. This approach requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process. Banker's algorithm is a deadlock avoidance algorithm that is applicable to a system with multiple instances of each resource type.

### EXAMPLE:

Consider the system with process  $\langle P_0, P_1, P_2, P_3, P_4 \rangle$  & 3 resources types A B C. The following snapshot of system has been given.

process	Allocation	MaxNeed	-Available	Remaining need
$P_0$	0 1 0	7 5 3	3 3 2 2 0 0	7 4 3 x ✓
$P_1$	2 0 0	3 2 2	5 3 2	1 2 2 ✓
$P_2$	3 0 2	9 0 2	2 1 1 7 4 3 0 0 2	6 0 0 x ✓
$P_3$	2 1 1	4 2 2	7 4 5 0 1 0	2 1 1 ✓
$P_4$	0 0 2	5 3 3	7 5 5 3 0 2	5 3 1 ✓
	7 2 5		10 5 7	
	10 5 7			
	- 7 2 5			
	3 3 2			

$$\text{Need} = \text{Max} - \text{Allocation}$$

$$\text{Available} \geq \text{need} \quad / \quad \text{need} \leq \text{Available}$$

$$\rightarrow P_0 \quad 8743 \neq 332 \quad \times$$

$$\rightarrow P_1 \quad 122 \leq 332 \quad \checkmark \quad (\text{Add to safe sequence})$$

$$122 \text{ allocation} = 200$$

$$\text{Now Available} + 200 = \text{new Available}$$

$$332 + 200 = 532$$

$$\rightarrow P_2 \quad 600 \neq 532 \quad \times$$

$$\rightarrow P_3 \quad 211 \leq 532 \quad \checkmark \quad (\text{Add to safe sequence})$$

$$211 \text{ allocation} = 211$$

$$\text{Now new available} = \text{Available} + 211$$

$$532 + 211 = 743$$

~~P<sub>4</sub> 743~~

$$\rightarrow P_4 \quad 531 \leq 743 \quad (\text{Add to safe sequence}) \quad \checkmark$$

$$531 \text{ Allocation} = 002$$

$$\text{new available} = \text{Available} + 002$$

$$743 + 002 = 745$$

$$\rightarrow P_0 \quad 743 \leq 745 \quad (\text{Add to safe sequence}) \quad \checkmark$$

$$743 \text{ Allocation} = 010$$

$$010 + 745 = 755$$

$$\rightarrow P_2 \quad 600 \leq 755 \quad (\text{Add to safe sequence}) \quad \checkmark$$

$$600 \text{ Allocation} = 302$$

$$\text{New Available} = 302 + 755$$

$$= 1057$$

$\langle P_1 P_3 P_4 P_0 P_2 \rangle$  — Safe Sequence

**PROGRAM:**

```

#include<stdio.h>

struct process
{
int allocation[3];
int max[3];
int need[3];
int finish;
}p[10];

int main()
{
int n,i,I,j,avail[3],work[3],flag,count=0,sequence[10],k=0;
printf("\nEnter the number of process:");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("\nEnter the %dth process allocated resources:",i); // allocated resource
for processess
scanf("%d%d%d",&p[i].allocation[0],&p[i].allocation[1],&p[i].allocation[2]);
printf("\nEnter the %dth process maximum resources:",i); // maximum
resources
scanf("%d%d%d",&p[i].max[0],&p[i].max[1],&p[i].max[2]);
p[i].finish=0;
p[i].need[0]=p[i].max[0]-p[i].allocation[0]; // we get need by subtracting
allocation resources from maximum resources
p[i].need[1]=p[i].max[1]-p[i].allocation[1];
p[i].need[2]=p[i].max[2]-p[i].allocation[2];
}
printf("\nEnter the available vector:"); // enter the available resources

```

```

scanf("%d%d%d",&avail[0],&avail[1],&avail[2]);
for(i=0;i<3;i++)
work[i]=avail[i]; // initialize work=available
while(count!=n)
{
count=0;
for(i=0;i<n;i++)
{
flag=1; // it indicates the process doesnot executed
if(p[i].finish==0)
if(p[i].need[0]<=work[0])
if(p[i].need[1]<=work[1])
if(p[i].need[2]<=work[2])
{
for(j=0;j<3;j++)
work[j]+=p[i].allocation[j]; // work =work+allocation
p[i].finish=1;// finish[i]=true then go to step intialize work=available
sequence[k++]=i;
flag=0;
}
if(flag==1)
count++;
}
}
count=0;
for(i=0;i<n;i++)
if(p[i].finish==1)

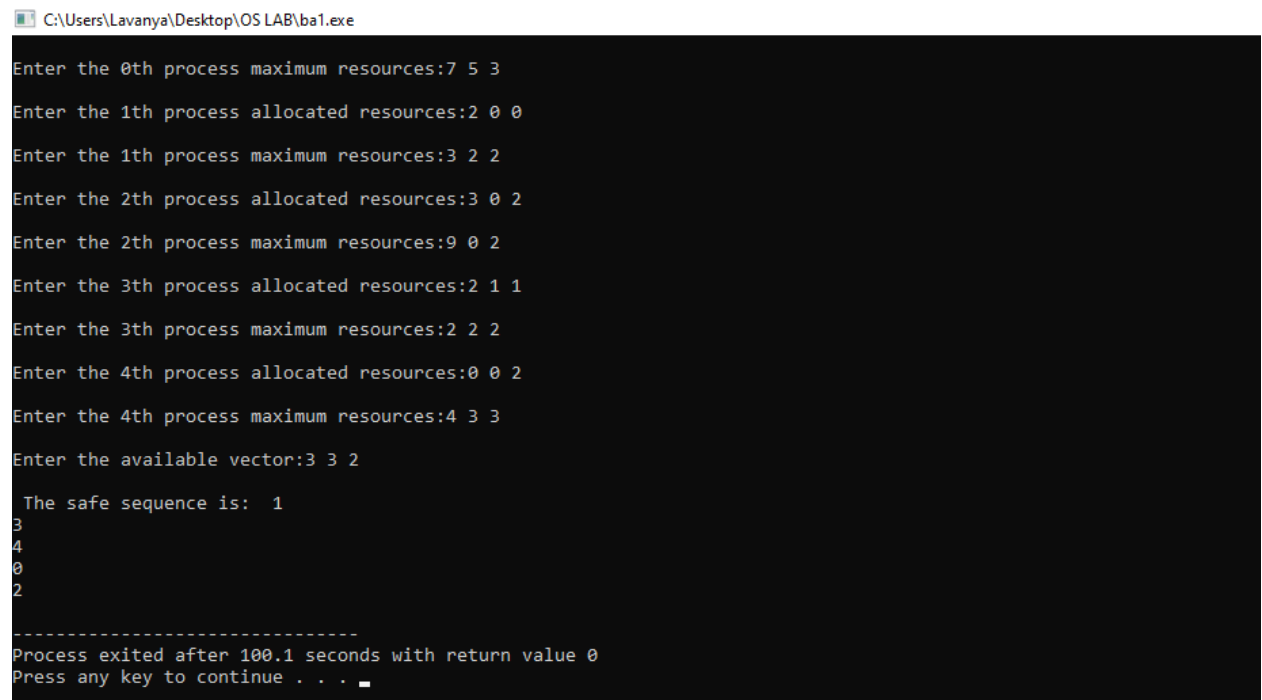
```

```

count++;
printf("\n The safe sequence is:\t");
if(count++==n)
for(i=0;i<k;i++)
printf("%d\n",sequence[i]);
else
printf("SYSTEM IS NOT IN A SAFE STATE \n\n");
return 0;
}

```

OUTPUT:



```

C:\Users\Lavanya\Desktop\OS LAB\ba1.exe
Enter the 0th process maximum resources:7 5 3
Enter the 1th process allocated resources:2 0 0
Enter the 1th process maximum resources:3 2 2
Enter the 2th process allocated resources:3 0 2
Enter the 2th process maximum resources:9 0 2
Enter the 3th process allocated resources:2 1 1
Enter the 3th process maximum resources:2 2 2
Enter the 4th process allocated resources:0 0 2
Enter the 4th process maximum resources:4 3 3
Enter the available vector:3 3 2

The safe sequence is: 1
3
4
0
2

-----
Process exited after 100.1 seconds with return value 0
Press any key to continue . . .

```

**7. Write a program to simulate the following contiguous memory allocation techniques**

**a) Worst-fit    b) Best-fit    c) First-fit**

**FIRST FIT:**

**AIM:** To Write a C program to simulate the following contiguous memory allocation techniques FIRST FIT

**DESCRIPTION:**

This method keeps the free/busy list of jobs organized by memory location, low-ordered to high-ordered memory. In this method, first job claims the first available memory with space more than or equal to it's size. The operating system doesn't search for appropriate partition but just allocate the job to the nearest memory partition available with sufficient size.

Job Number	Memory Requested
J1	20 K
J2	200 K
J3	500 K
J4	50 K

Memory location	Memory block size	Job number	Job size	Status	Internal fragmentation
10567	200 K	J1	20 K	Busy	180 K
30457	30 K			Free	30
300875	700 K	J2	200 K	Busy	500 K
809567	50 K	J4	50 K	Busy	None
Total available :	980 K	Total used :	270 K		710 K

As illustrated above, the system assigns J1 the nearest partition in the memory. As a result, there is no partition with sufficient space is available for J3 and it is placed in the waiting list.

**CODE:**

```
#include<stdio.h>

void firstFit(int blockSize[], int m, int processSize[], int n)
{
    int i, j;
    // Stores block id of the
```

```

// block allocated to a process
int allocation[n],already_allocated[n];

// Initially no block is assigned to any process
for(i = 0; i < n; i++)
{
    allocation[i] = -1; /*To store allocated block of particular process*/
    already_allocated[i]=-1; /*to indicate whether a particular block is
                                already allocated or not*/
}

// pick each process and find suitable blocks
// according to its size ad assign to it
for (i = 0; i < n; i++)      //here, n -> number of processes
{
    for (j = 0; j < m; j++)    //here, m -> number of blocks
    {
        if ((blockSize[j] >= processSize[i]) &&
already_allocated[j]==-1)
        {
            // allocating block j to the ith process
            allocation[i] = j;
            already_allocated[j]=0;
            // Reduce available memory in this block.
            blockSize[j] -= processSize[i];

            break; //go to the next process in the queue
        }
    }
}

```



```

    }
}

printf("\nProcess No.\tProcess Size\tBlock no.\n");
for (i=0; i < n; i++)
{
    printf(" %i\t\t", i+1);
    printf("%i\t\t\t", processSize[i]);
    if (allocation[i] != -1)
        printf("%i", allocation[i] + 1);
    else
        printf("Not Allocated");
    printf("\n");
}
}

```

// Driver code

```

int main()
{
    int i;
    int m; //number of blocks in the memory
    printf("enter no of blocks in memory:\n");
    scanf("%d",&m);
    int n; //number of processes in the input queue
    printf("enter no of processes in input queue:\n");
    scanf("%d",&n);
    int blockSize[m];

```

```

int processSize[n];
for(i=0;i<m;i++){
    printf("\nenter %d block size:",i+1);
    scanf("%d",&blockSize[i]);
}

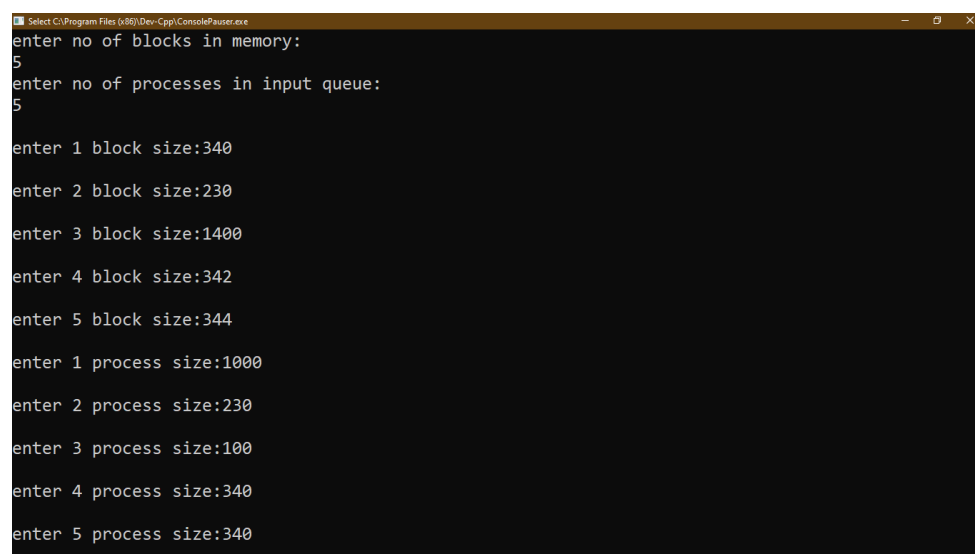
for(i=0;i<n;i++){
    printf("\nenter %d process size:",i+1);
    scanf("%d",&processSize[i]);
}

firstFit(blockSize, m, processSize, n);

return 0 ;
}

```

## INPUT:



```

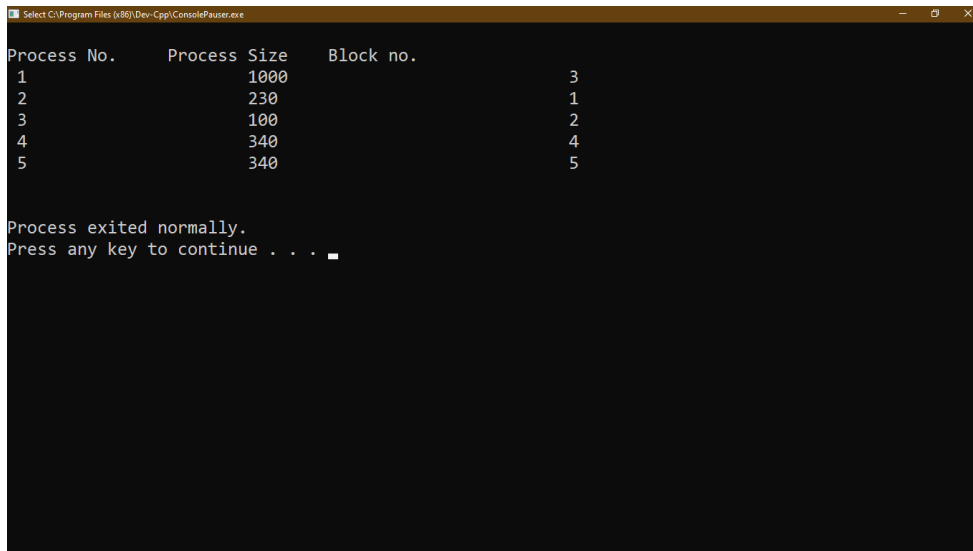
Select C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
enter no of blocks in memory:
5
enter no of processes in input queue:
5

enter 1 block size:340
enter 2 block size:230
enter 3 block size:1400
enter 4 block size:342
enter 5 block size:344

enter 1 process size:1000
enter 2 process size:230
enter 3 process size:100
enter 4 process size:340
enter 5 process size:340

```

## OUTPUT:



```
Select C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe

Process No.    Process Size    Block no.
1              1000           3
2              230            1
3              100            2
4              340            4
5              340            5

Process exited normally.
Press any key to continue . . .
```

## WORST FIT:

**AIM:** To Write a C program to simulate the following contiguous memory allocation techniques WORST FIT.

## DESCRIPTION:

In this allocation technique, the process traverses the whole memory and always search for the largest hole/partition, and then the process is placed in that hole/partition. It is a slow process because it has to traverse the entire memory to search the largest hole.

Process Number	Process Size
P1	30K
P2	100K
P3	45K

MEMORY LOCATION	MEMORY BLOCK SIZE	PROCESS NUMBER	PROCESS SIZE	STATUS	INTERNAL FRAGMENTATION
12345	50K	P3	45K	Busy	5K
45871	100K	P2	100K	Busy	None
1245	400K	P1	30K	Busy	370K
TOTAL AVAILABLE:	550K	TOTAL USED:	175K		375K

Here Process P1=30K is allocated with the Worst Fit-Allocation technique, so it traverses the entire memory and selects memory size 400K which is the largest, and hence there is an internal fragmentation of 370K which is very large and so many other processes can also utilize this leftover space.

### CODE:

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

```
void WorstFit(int blockSize[], int blocks, int processSize[], int processes)
{
    // This will store the block id of the allocated block to a process
    int allocation[processes];
    int occupied[blocks];
    int i,j;
    // initially assigning -1 to all allocation indexes
    // means nothing is allocated currently
    for(i = 0; i < processes; i++){
        allocation[i] = -1;
    }
}
```

```

for(i = 0; i < blocks; i++){
    occupied[i] = 0;
}

// pick each process and find suitable blocks
// according to its size and assign to it
for (i=0; i < processes; i++)
{
    int indexPlaced = -1;
    for(j = 0; j < blocks; j++)
    {
        // if not occupied and block size is large enough
        if(blockSize[j] >= processSize[i] && !occupied[j])
        {
            // place it at the first block fit to accommodate process
            if (indexPlaced == -1)
                indexPlaced = j;

            // if any future block is larger than the current block where
            // process is placed, change the block and thus indexPlaced
            else if (blockSize[indexPlaced] < blockSize[j])
                indexPlaced = j;
        }
    }
}

// If we were successfully able to find block for the process
if (indexPlaced != -1)
{

```

```

        // allocate this block j to process p[i]
        allocation[i] = indexPlaced;

        // make the status of the block as occupied
        occupied[indexPlaced] = 1;

        // Reduce available memory for the block
        blockSize[indexPlaced] -= processSize[i];
    }
}

printf("\nProcess No.\tProcess Size\tBlock no.\n");
for (i = 0; i < processes; i++)
{
    printf("%d \t\t\t %d \t\t\t", i+1, processSize[i]);
    if (allocation[i] != -1)
        printf("%d\n", allocation[i] + 1);
    else
        printf("Not Allocated\n");
}
}

// Driver code
int main()
{
    int i;
    int m; //number of blocks in the memory
    printf("enter no of blocks in memory:\n");

```

```

scanf("%d",&m);

int n; //number of processes in the input queue
printf("enter no of processes in input queue:\n");
scanf("%d",&n);

int blockSize[m];
int processSize[n];
for(i=0;i<m;i++){
    printf("\nenter %d block size:",i+1);
    scanf("%d",&blockSize[i]);
}

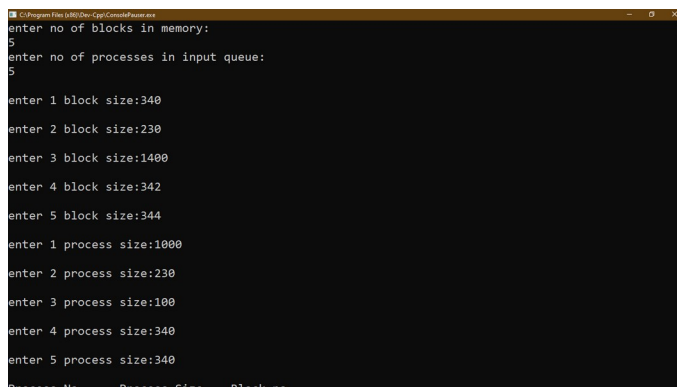
for(i=0;i<n;i++){
    printf("\nenter %d process size:",i+1);
    scanf("%d",&processSize[i]);
}

WorstFit(blockSize, m, processSize, n);

return 0 ;
}

```

## INPUT:



```

C:\Program Files (x86)\Dev-Cpp\bin\gcc.exe
enter no of blocks in memory:
5
enter no of processes in input queue:
5
enter 1 block size:340
enter 2 block size:230
enter 3 block size:1400
enter 4 block size:342
enter 5 block size:344
enter 1 process size:1000
enter 2 process size:230
enter 3 process size:100
enter 4 process size:340
enter 5 process size:340
Process No.    Process Size    Block no.

```

## OUTPUT:

```

C:\Program Files (x86)\Dev-Cpp\Conio\Pause.exe
Process No.   Process Size   Block no.
1             1000           3
2             230           5
3             100           4
4             340           1
5             340           Not Allocated

Process exited normally.
Press any key to continue . . .

```

## BEST FIT:

**AIM:** To Write a C program to simulate the following contiguous memory allocation techniques BEST FIT.

## DESCRIPTION:

This method keeps the free/busy list in order by size – smallest to largest. In this method, the operating system first searches the whole of the memory according to the size of the given job and allocates it to the closest-fitting free partition in the memory, making it able to use memory efficiently. Here the jobs are in the order from smallest job to largest job.

Job Number	Memory Requested
J1	20 K
J2	200 K
J3	500 K
J4	50 K

Memory location	Memory block size	Job number	Job size	Status	Internal fragmentation
10567	30 K	J1	20 K	Busy	10 K
30457	50 K	J4	50 K	Busy	None
300875	200 K	J2	200 K	Busy	None
809567	700 K	J3	500 K	Busy	200 K
Total available :	980 K	Total used :	770 K		210 K

As illustrated in above figure, the operating system first search throughout the memory and allocates the job to the minimum possible memory partition, making the memory allocation efficient.

## CODE:

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

```
void BestFit(int blockSize[], int blocks, int processSize[], int processes)
```



```

{
    // This will store the block id of the allocated block to a process
    int allocation[processes];
    int occupied[blocks];
    int i,j;
    // initially assigning -1 to all allocation indexes
    // means nothing is allocated currently
    for(i = 0; i < processes; i++){
        allocation[i] = -1;
    }

    for(i = 0; i < blocks; i++){
        occupied[i] = 0;
    }

    // pick each process and find suitable blocks
    // according to its size and assign to it
    for (i=0; i<processes; i++)
    {

        int indexPlaced = -1;
        for (j=0; j<blocks; j++)
        {
            if (blockSize[j] >= processSize[i] && !occupied[j])
            {
                // place it at the first block fit to accommodate process
                if (indexPlaced == -1)
                    indexPlaced = j;
            }
        }
    }
}

```

```

        // if any future block is smaller than the current block where
        // process is placed, change the block and thus indexPlaced
        else if (blockSize[indexPlaced] >= blockSize[j])
            indexPlaced = j;
    }
}

// If we were successfully able to find block for the process
if (indexPlaced != -1)
{
    // allocate this block j to process p[i]
    allocation[i] = indexPlaced;

    // make the status of the block as occupied
    occupied[indexPlaced] = 1;

    // Reduce available memory for the block
    blockSize[indexPlaced] -= processSize[i];
}
}

printf("\nProcess No.\tProcess Size\tBlock no.\n");
for (i = 0; i < processes; i++)
{
    printf("%d \t\t\t %d \t\t\t", i+1, processSize[i]);
    if (allocation[i] != -1)
        printf("%d\n", allocation[i] + 1);
}

```

```

        else
            printf("Not Allocated\n");
    }
}

// Driver code
int main()
{
    int i;
    int m; //number of blocks in the memory
    printf("enter no of blocks in memory:\n");
    scanf("%d",&m);
    int n; //number of processes in the input queue
    printf("enter no of processes in input queue:\n");
    scanf("%d",&n);
    int blockSize[m];
    int processSize[n];
    for(i=0;i<m;i++){
        printf("\nenter %d block size:",i+1);
        scanf("%d",&blockSize[i]);
    }

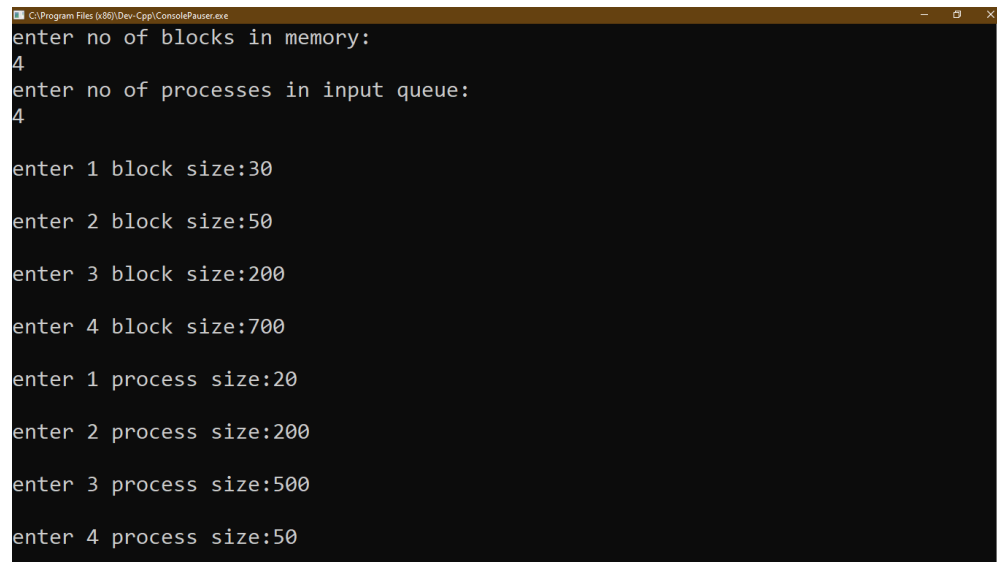
    for(i=0;i<n;i++){
        printf("\nenter %d process size:",i+1);
        scanf("%d",&processSize[i]);
    }

    BestFit(blockSize, m, processSize, n);
}

```

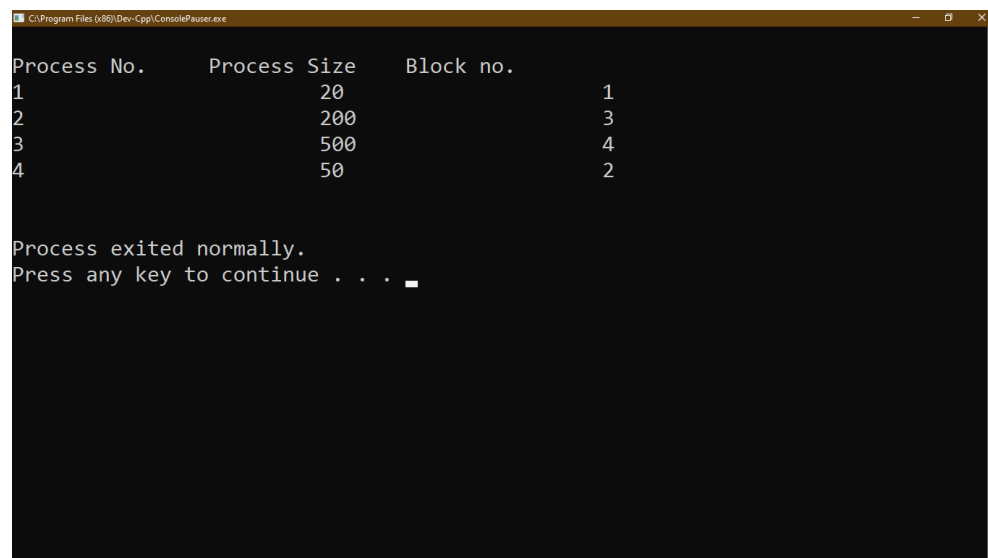
```
    return 0 ;  
}
```

### INPUT:



```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe  
enter no of blocks in memory:  
4  
enter no of processes in input queue:  
4  
  
enter 1 block size:30  
enter 2 block size:50  
enter 3 block size:200  
enter 4 block size:700  
  
enter 1 process size:20  
enter 2 process size:200  
enter 3 process size:500  
enter 4 process size:50
```

### OUTPUT:



```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe  
Process No.      Process Size      Block no.  
1                20               1  
2                200             3  
3                500             4  
4                50              2  
  
Process exited normally.  
Press any key to continue . . .
```

## 8. Write a program to simulate paging technique of memory management.

**AIM:** write a c program to simulate paging technique of memory management.

**DESCRIPTION:** In computer operating systems, paging is one of the memory management schemes by which a computer stores and retrieves data from the secondary storage for use in main memory. In the paging memory-management scheme, the operating system retrieves data from secondary storage in same-

size blocks called pages. Paging is a memory-management scheme that permits the physical address space a process to be noncontiguous. The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from their source.

EXAMPLE:

**PROGRAM:**

```
#include<stdio.h>
#include<conio.h>
main()
{
int ms, ps, nop, np, rempages, i, j, x, y, pa, offset;
int s[10], fno[10][20];
printf("\nEnter the memory size -- ");
scanf("%d",&ms);
printf("\nEnter the page size -- ");
scanf("%d",&ps);
nop = ms/ps;
printf("\nThe no. of pages available in memory are -- %d ",nop);
printf("\nEnter number of processes -- ");
scanf("%d",&np);
rempages = nop;
for(i=1;i<=np;i++)
{
printf("\nEnter no. of pages required for p[%d]-- ",i);
```

```

scanf("%d",&s[i]);
if(s[i] > rempages)
{
printf("\nMemory is Full");
break;
}
rempages = rempages - s[i];
printf("\nEnter pagetable for p[%d] --- ",i);
for(j=0;j<s[i];j++)
scanf("%d",&fno[i][j]);
}
printf("\nEnter Logical Address to find Physical Address ");
printf("\nEnter process no. and pagenumber and offset -- ");
scanf("%d %d %d",&x,&y, &offset);
if(x>np || y>=s[i] || offset>=ps)
printf("\ninvalid Process or Page Number or offset");
else
{
pa=fno[x][y]*ps+offset;
printf("\nThe Physical Address is -- %d",pa);
}
getch();
}

```

OUTPUT:

C:\Users\Lavanya\Desktop\OS LAB\paging technique.exe

```
Enter the memory size -- 1000
Enter the page size -- 100
The no. of pages available in memory are -- 10
Enter number of processes -- 3
Enter no. of pages required for p[1]-- 4
Enter pagetable for p[1] --- 8 6 9 5
Enter no. of pages required for p[2]-- 5
Enter pagetable for p[2] --- 1 4 5 7 3
Enter no. of pages required for p[3]-- 5
Memory is Full
Enter Logical Address to find Physical Address
Enter process no. and pagenumber and offset --
```

## 9. Write a C program to simulate page replacement algorithms

a) FIFO b) LRU

**FIFO:**

**AIM:** C program to simulate page replacement algorithm FIFO.

**DESCRIPTION:**

This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

**Example:** Consider page reference string 1, 3, 0, 3, 5, 6, 3 with 3 page frames. Find the number of page faults.

Initially, all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> **3 Page Faults.**

When 3 comes, it is already in memory so —> **0 Page Faults.** Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —> **1**

**Page Fault.** 6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —> **1 Page Fault.** Finally, when 3 come it is not available so it replaces 0 **1 page fault.**

Page  
reference

1, 3, 0, 3, 5, 6, 3

1	3	0	3	5	6	3
		0	0	0	0	3
	3	3	3	3	6	6
1	1	1	1	5	5	5
Miss	Miss	Miss	Hit	Miss	Miss	Miss

Total Page Fault = 6

## CODE:

```
#include<stdio.h>

int main()
{
    int i,j,n,a[50],frame[10],no,k,avail,count=0;

    printf("\n ENTER THE NUMBER OF PAGES:\n");

    scanf("%d",&n);

    printf("\n ENTER THE PAGE NUMBER :\n");
    for(i=1;i<=n;i++)
        scanf("%d",&a[i]);

    printf("\n ENTER THE NUMBER OF FRAMES :");
    scanf("%d",&no);

    for(i=0;i<no;i++)
        frame[i]= -1;

    j=0;

    printf("\ntref string\t page frames\n");

    for(i=1;i<=n;i++){

        printf("%d\t\t",a[i]);  /* print element which is going to be inserted to
                                   page table currently*/

        avail=0;                /*flag to whether page is already present or not*/

        for(k=0;k<no;k++)
```

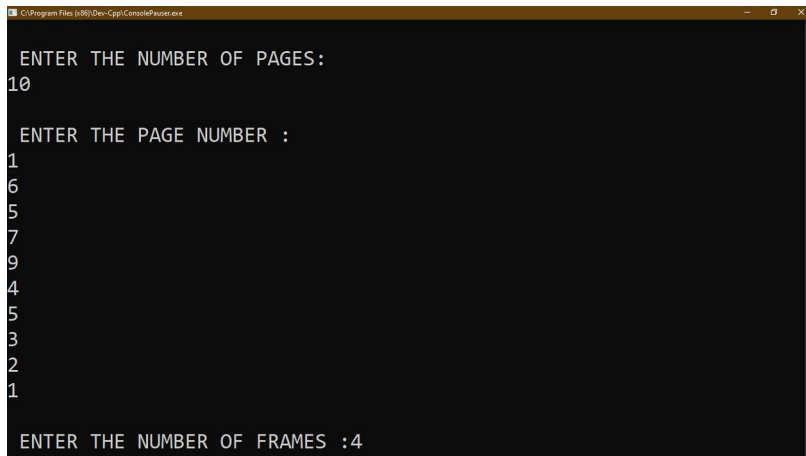


```

        if(frame[k]==a[i])
            avail=1;    /* set flag to 1 if it is already present*/
        if (avail==0){    /*If page is not present in the frames insert it*/
            frame[j]=a[i];
            j=(j+1)%no;    /* this expression helps in maintaining FIFO order*/
            count++;    /* count no of page faults */
            for(k=0;k<no;k++) /* prints frames data as there is alternation
occured*/
                printf("%d\t",frame[k]);
        }
        printf("\n");
    }
    printf("Page Fault Is %d",count);
    return 0;
}

```

### Input:



```

C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
ENTER THE NUMBER OF PAGES:
10
ENTER THE PAGE NUMBER :
1
6
5
7
9
4
5
3
2
1
ENTER THE NUMBER OF FRAMES :4

```

### Output:

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
ENTER THE NUMBER OF FRAMES :4
      ref string      page frames
1         1        -1        -1        -1
6         1         6        -1        -1
5         1         6         5        -1
7         1         6         5         7
9         9         6         5         7
4         9         4         5         7
5
3         9         4         3         7
2         9         4         3         2
1         1         4         3         2
Page Fault Is 9

Process exited normally.
Press any key to continue . . .
```

## LRU

**AIM:** C program to simulate page replacement algorithm FIFO

### DESCRIPTION:

In this algorithm, page will be replaced which is least recently used.

**Example:** Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 3 with 4 page frames. Find number of page faults.

Initially, all slots are empty, so when 7 0 1 2 are allocated to the empty slots

—> **4 Page faults**

0 is already there so —> **0 Page fault**. When 3 came it will take the place of 7 because it is least recently used —> **1 Page fault**

0 is already in memory so —> **0 Page fault**.

4 will takes place of 1 —> **1 Page Fault**

Now for the further page reference string —> **0 Page fault** because they are already available in the memory.

Page  
reference

7,0,1,2,0,3,0,4,2,3,0,3,2,3

No. of Page frame - 4

7	0	1	2	0	3	0	4	2	3	0	3	2	3
			2	2	2	2	2	2	2	2	2	2	2
		1	1	1	1	1	4	4	4	4	4	4	4
	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3	3	3	3	3	3

Miss Miss Miss Miss Hit Miss Hit Miss Hit Hit Hit Hit Hit Hit

Total Page Fault = 6

Here LRU has same number of page fault as optimal but it may differ according to question.

## CODE:

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

```
int findLRU(int time[], int n){/* function to find least recently used frame position*/
```

```
int i, minimum = time[0], pos = 0;
```

```
for(i = 1; i < n; ++i){
```

```
if(time[i] < minimum){
```

```
minimum = time[i];
```

```
pos = i;
```

```
}
```

```
}
```

```
return pos;
```

```
}
```

```
int main()
```

```
{
```

```

    int no_of_frames, no_of_pages, frames[10], pages[30], counter = 0, time[10],
    flag1, flag2, i, j, pos, faults = 0;
    printf("Enter number of frames: ");
    scanf("%d", &no_of_frames);
    printf("Enter number of pages: ");
    scanf("%d", &no_of_pages);
    printf("Enter reference string: "); /* sequence of pages to be accessed */
    for(i = 0; i < no_of_pages; ++i){
        scanf("%d", &pages[i]);
    }

    for(i = 0; i < no_of_frames; ++i){// initially load all frames with -1
        frames[i] = -1;
    }

    for(i = 0; i < no_of_pages; ++i){ // set flags to 0
        flag1 = flag2 = 0;

        for(j = 0; j < no_of_frames; ++j){ /* check whether particular page is
        already available, if founds increment its hit*/
            if(frames[j] == pages[i]){
                counter++;
                time[j] = counter;
                flag1 = flag2 = 1; // forbids execution of next 2 loops
                break;
            }
        }
    }

```

```

    if(flag1 == 0){ // take page to the frame if any frame is free
for(j = 0; j < no_of_frames; ++j){
    if(frames[j] == -1){
        counter++;
        faults++;
        frames[j] = pages[i];
        time[j] = counter;
        flag2 = 1; // forbids execution of next loop
        break;
    }
}
}

if(flag2 == 0){ // finds least recently used page and replaces it
    pos = findLRU(time, no_of_frames); // function returns position of LRU
page
    counter++;
    faults++;
    frames[pos] = pages[i];
    time[pos] = counter;
}

printf("\n");

for(j = 0; j < no_of_frames; ++j){
    printf("%d\t", frames[j]);
}
}

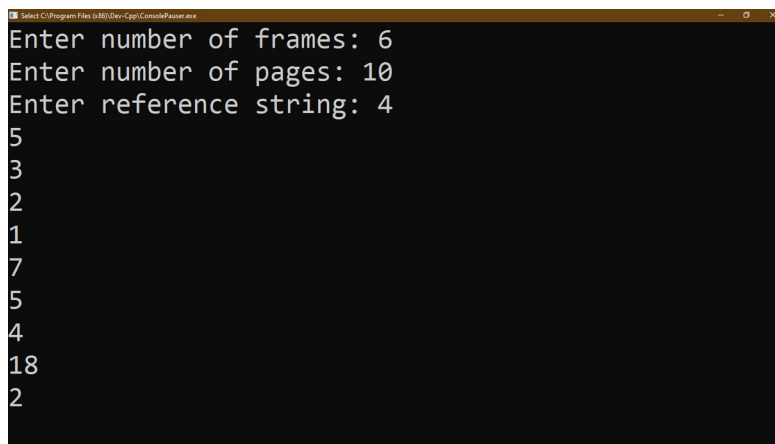
```

```
printf("\n\nTotal Page Faults = %d", faults);
```

```
    return 0;
```

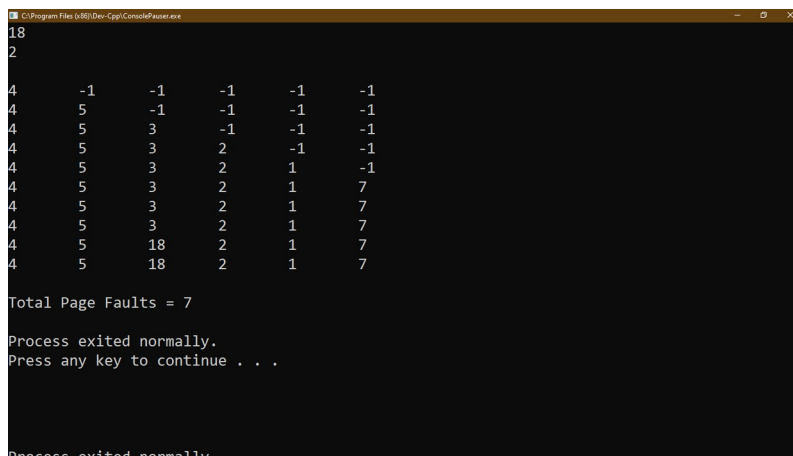
```
}
```

### INPUT:



```
C:\Program Files (x86)\Dev-Cpp\ConsolePause.exe
Enter number of frames: 6
Enter number of pages: 10
Enter reference string: 4
5
3
2
1
7
5
4
18
2
```

### OUTPUT:



```
C:\Program Files (x86)\Dev-Cpp\ConsolePause.exe
18
2

4      -1      -1      -1      -1      -1
4      5      -1      -1      -1      -1
4      5      3      -1      -1      -1
4      5      3      2      -1      -1
4      5      3      2      1      -1
4      5      3      2      1      7
4      5      3      2      1      7
4      5      3      2      1      7
4      5      18     2      1      7
4      5      18     2      1      7

Total Page Faults = 7
Process exited normally.
Press any key to continue . . .
Process exited normally.
```

## 10. Write a C program to simulate disk scheduling algorithms

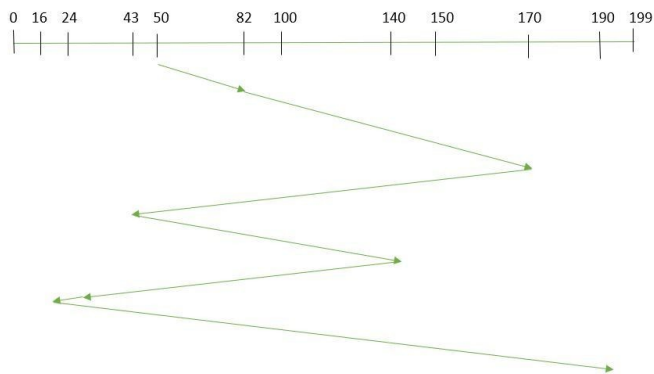
a) FCFS b) SCAN c) C-SCAN d) LOOK e) C-LOOK

### FCFS:

AIM: C program to simulate FCFS disk scheduling algorithm

### DESCRIPTION:

FCFS is the simplest of all the 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.



### **Example:**

1. Suppose the order of request is- (82,170,43,140,24,16,190)

And current position of Read/Write head is : 50

1. So, total seek time:

$$= (82-50) + (170-82) + (170-43) + (140-43) + (140-24) + (24-16) + (190-16) \\ = 642$$

### **CODE:**

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

```
int main(){
```

```
    int n,header_moves,first,last,cur_pos;
```

```
    printf("enter first cylinder number:");
```

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

```
    printf("enter last cylinder number");
```

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

```
    printf("Enter current position of header");
```

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

```
    printf("enter no of requests:");
```

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

```
    int request[n],service_seq[n];
```

```
    int i;
```

```

header_moves=0; /* To count no of cylinders traversed by header*/
int count=0;
for(i=0;i<n;i++){
    printf("enter request %d :",i+1);
    scanf("%d",&request[i]);
    if(request[i]<=last && request[i]>=first){
        /* validate whether requested cylinder is
            present or not */
        service_seq[i]=++count; /* n th time serviced*/
        if(cur_pos>=request[i]){ /* increment header moves*/
            header_moves+=(cur_pos-request[i]);
            /*current position is grater than present request
            so no of cylinders traversed is cur_pos-request[i] */
        }
        else{
            header_moves+=(request[i]-cur_pos);
            /*current position is less than present request
            so no of cylinders traversed is request[i]-cur_pos */
        }
        cur_pos=request[i];
        /*present request now becomes current position of header*/
    }
    else{
        service_seq[i]=0;
        /* request cant be serviced if it is out of range*/
    }
}

```

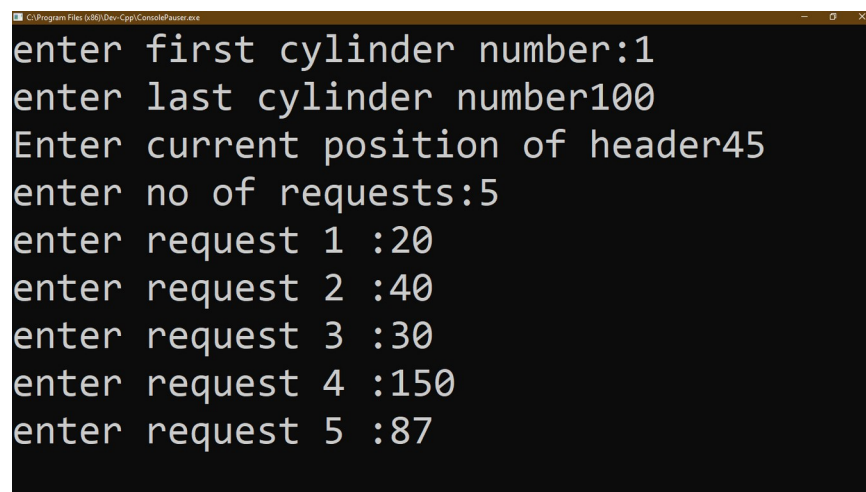


```

printf("\n no of head movements is:\t %d",header_moves);
printf("\nrequest\t service_sequence\n");
for(i=0;i<n;i++){
    printf("%d\t %d",request[i],service_seq[i]);
    printf("\n");
}
}

```

### INPUT:

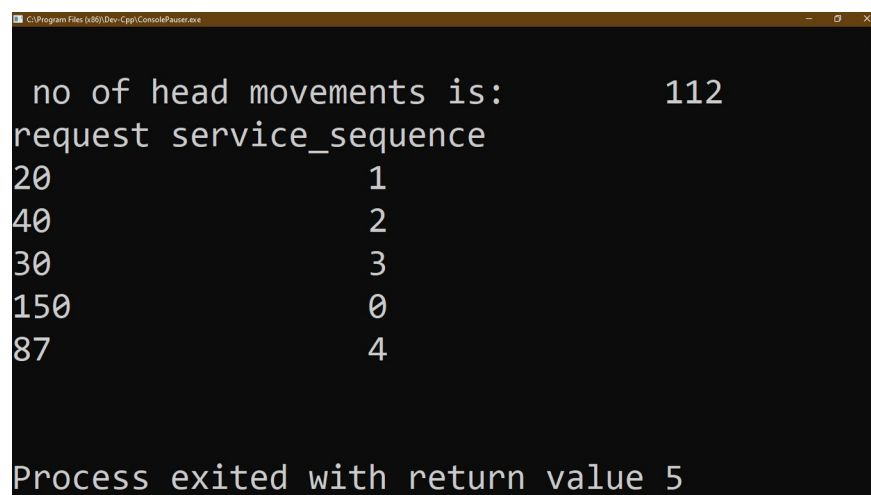


```

C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
enter first cylinder number:1
enter last cylinder number:100
Enter current position of header:45
enter no of requests:5
enter request 1 :20
enter request 2 :40
enter request 3 :30
enter request 4 :150
enter request 5 :87

```

### OUTPUT:



```

C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
no of head movements is:      112
request service_sequence
20      1
40      2
30      3
150     0
87      4

Process exited with return value 5

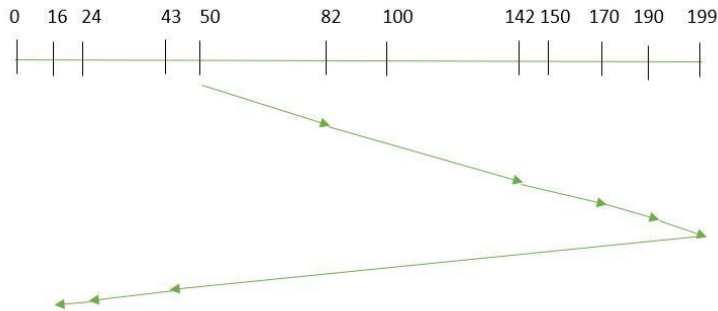
```

### SCAN:

**AIM:** C program to simulate SCAN disk scheduling algorithm

### DESCRIPTION:

In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works as an elevator and hence also known as **elevator algorithm**. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.



### **Example:**

Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **“towards the larger value”**.

Therefore, the seek time is calculated as:

$$\begin{aligned}
 &= (199 - 50) + (199 - 16) \\
 &= 332
 \end{aligned}$$

### **CODE:**

```

#include<stdio.h>

int main(){
    int n,header_moves,first,last,cur_pos,direction;
    printf("enter first cylinder number:");
    scanf("%d",&first);
    printf("enter last cylinder number");
    scanf("%d",&last);
    printf("Enter current position of header");
    scanf("%d",&cur_pos);
    printf("Enter direction 0->left 1->right");

```

```

        /* current direction of header*/
scanf("%d",&direction);
printf("enter no of requests:");
scanf("%d",&n);
int request[n];
int i;
header_moves=0;
int count=0;
for(i=0;i<n;i++){
    printf("enter request %d :",i+1);
    scanf("%d",&request[i]);
}
/* sorting the requests */
int min_pos,j,temp;
for(i=0;i<n;i++){
    min_pos=i;
    for(j=i+1;j<n;j++){
        if(request[min_pos]>request[j]){
            min_pos=j;
        }
    }
    temp=request[i];
    request[i]=request[min_pos];
    request[min_pos]=temp;
}
printf("\n");

/* print sorted requests*/

```

```

for(i=0;i<n;i++){
    printf("%d",request[i]);
}
printf("\n");

```

*/\* if header is headed toward left go to first cylinder and after that check whether all requests are serviced or not*

*ex: if the sequence is 5 7 8 23 70*

*header at 80 & direction is left*

*if it reaches first cylinder now it means it serviced all the requests*

*if the sequence is 5 7 8 23 70*

*header is at 50 & direction is left*

*now if it reaches first cylinder from 50 still it need to cover other requests*

*If it unable to cover any one request it will start from that first cylinder and*

*reaches last cylinder in the request[] array.*

*\*/*

```

if(direction==0){
    header_moves+=(cur_pos-first); // moving toward first cylinder
    if(request[n-1]>cur_pos){ // check if there are any un covered
cylinders in the request
        header_moves+=(request[n-1]-first);
    }
}

```

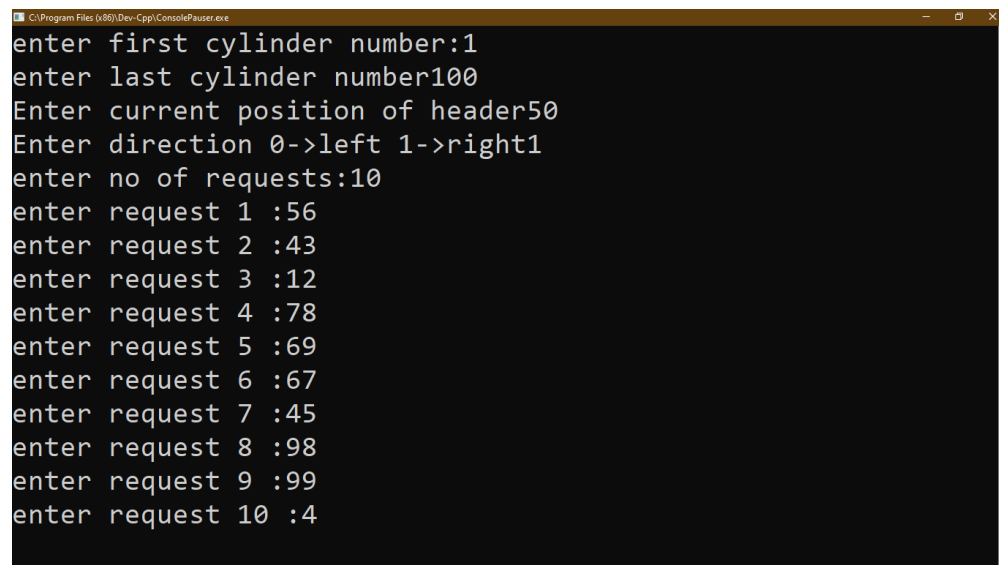
*/\* same as above but header is headed toward right*

*header first reaches last cylinder and check whether all the requests are serviced or not\*/*

```
else{
    header_moves+=(last-cur_pos);
    if(request[0]<cur_pos){
        header_moves+=(last-request[0]);
    }
}

printf("\n no of head movements is:\t %d",header_moves);
}
```

### INPUT:



```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
enter first cylinder number:1
enter last cylinder number:100
Enter current position of header:50
Enter direction 0->left 1->right:1
enter no of requests:10
enter request 1 :56
enter request 2 :43
enter request 3 :12
enter request 4 :78
enter request 5 :69
enter request 6 :67
enter request 7 :45
enter request 8 :98
enter request 9 :99
enter request 10 :4
```

### OUTPUT:

```
Select C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe

4,12,43,45,56,67,69,78,98,99,

no of head movements is:      146

Process exited with return value 31
Press any key to continue . . .
```

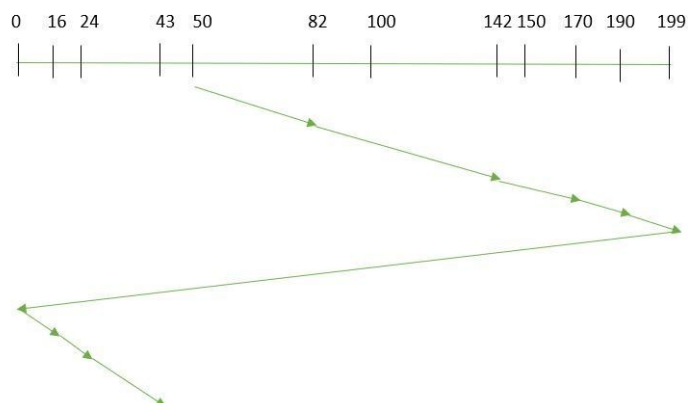
## C SCAN:

**AIM:** C program to simulate SCAN disk scheduling algorithm

### DESCRIPTION:

In 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 *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 SCAN algorithm and hence it is known as C-SCAN (Circular SCAN).



### Example:

Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move **“towards the larger value”**.

Seek time is calculated as:

$$\begin{aligned}
 &=(199-50)+(199-0)+(43-0) \\
 &=391
 \end{aligned}$$

## CODE:

```

#include<stdio.h>

/* C SCAN is same as scan but after reaching first or last cylinder based
upon direction of header it will moves from current end (first or last cylinder)
to opposite end(last or first cylinder).
now service the remaining requests*/

int main(){
    int n,header_moves,first,last,cur_pos,direction,last_visit;
    printf("enter first cylinder number:");
    scanf("%d",&first);
    printf("enter last cylinder number");
    scanf("%d",&last);
    printf("Enter current position of header");
    scanf("%d",&cur_pos);
    printf("Enter direction 0->left 1->right");
    scanf("%d",&direction);
    printf("enter no of requests:");
    scanf("%d",&n);
    int request[n];
    int i;
    header_moves=0;
    int count=0;
    for(i=0;i<n;i++){
        printf("enter request %d :",i+1);
        scanf("%d",&request[i]);
    }
}

```

```
}
```

```
/* sorting the requests */
```

```
int min_pos,j,temp;
```

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

```
    min_pos=i;
```

```
    for(j=i+1;j<n;j++){
```

```
        if(request[min_pos]>request[j]){
```

```
            min_pos=j;
```

```
        }
```

```
    }
```

```
    temp=request[i];
```

```
    request[i]=request[min_pos];
```

```
    request[min_pos]=temp;
```

```
}
```

```
printf("\n");
```

```
/* print sorted requests*/
```

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

```
    printf("%d",request[i]);
```

```
}
```

```
printf("\n");
```

```
if(direction==0){ //for left direction
```

```
    header_moves+=(cur_pos-first); // moving toward first cylinder
```

```
    if(request[n-1]>cur_pos){
```

```
        /*check if there are any un covered cylinders in the request
```

```
        if yes go to last cylinder to service the remaining requests
```



```

    if last number in request array is grater than current
position
    means we will not service it while moving toward left end*/
    header_moves+=(last-first)
    for(i=0;i<n;i++){    /* findout highest cylinder number
among
    cylinders left to be serviced (we are directly finding it since
    we stop our header at that postion)*/
        if(request[i]>cur_pos){
            last_visit=request[i];
            break;
        }
    }
    header_moves+=(last-last_visit);
}

/* same scenario but header is headed toward right*/
else{
    header_moves+=(last-cur_pos);
    if(request[0]<cur_pos){
        header_moves+=(last-first);
        for(i=0;i<n;i++){
            if(request[i]>cur_pos){
                last_visit=request[i-1];
                break;
            }
        }
    }
}

```

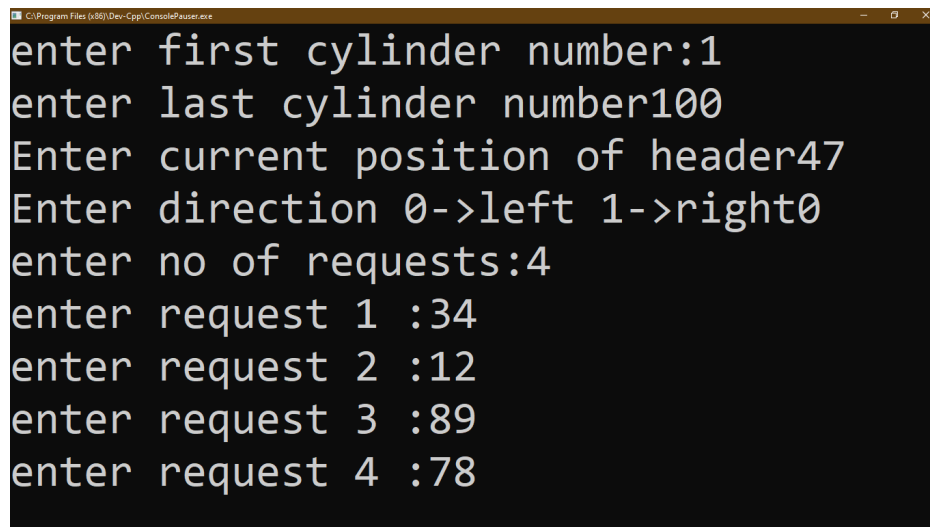
```

        header_moves+=(last_visit-first);
    }
}

printf("\n no of head movements is:\t %d",header_moves);
}

```

### Input:

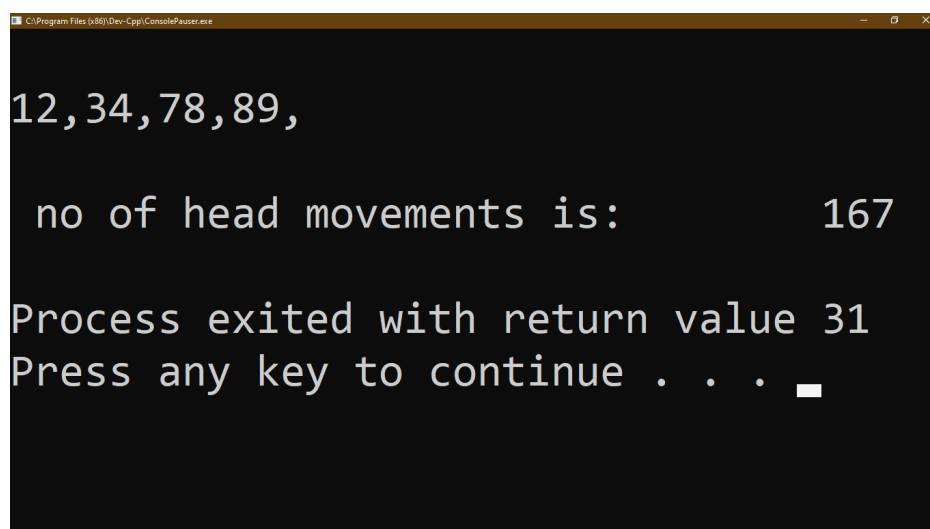


```

C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
enter first cylinder number:1
enter last cylinder number:100
Enter current position of header:47
Enter direction 0->left 1->right:0
enter no of requests:4
enter request 1 :34
enter request 2 :12
enter request 3 :89
enter request 4 :78

```

### Output:



```

C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe
12,34,78,89,

no of head movements is:          167

Process exited with return value 31
Press any key to continue . . .

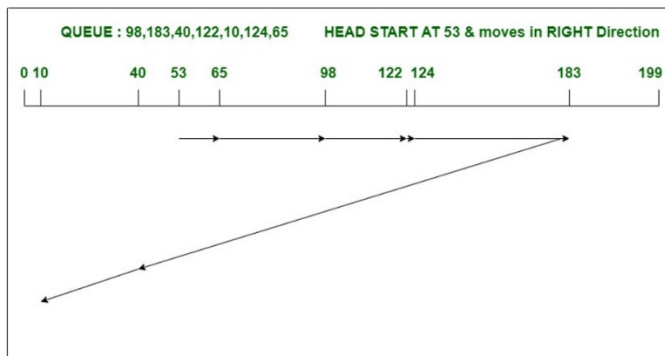
```

### LOOK:

**AIM:** C program to simulate LOOK disk scheduling algorithm

### DESCRIPTION:

Look Algorithm is actually an improved version of SCAN Algorithm. In this algorithm, the head starts from first request at one side of disk and moves towards the other end by serving all requests in between. After reaching the last request of one end, the head reverse its direction and returns to first request, servicing all requests in between. Unlike SCAN, in this the head instead of going till last track, it goes till last request and then direction is changed.



### Example –

Consider a disk with 200 tracks (0-199) and the disk queue having I/O requests in the following order as follows: 98, 183, 40, 122, 10, 124, 65. The current head position of the Read/Write head is 53 and will move in Right direction. Calculate the total number of track movements of Read/Write head using LOOK algorithm.

$$\begin{aligned}
 &= (65-53) + (98-65) + (122-98) \\
 &\quad + (124-122) + (183-124) + (183-40) + (40-10) \\
 &= 303
 \end{aligned}$$

### CODE:

```

#include<stdio.h>

int main(){
    int n,header_moves,first,last,cur_pos,direction;
    printf("enter first cylinder number:");
    scanf("%d",&first);
    printf("enter last cylinder number");
    scanf("%d",&last);
    printf("Enter current position of header");
    scanf("%d",&cur_pos);
    printf("Enter direction 0->left 1->right");

```

```

        /* current direction of header*/
scanf("%d",&direction);
printf("enter no of requests:");
scanf("%d",&n);
int request[n];
int i;
header_moves=0;
int count=0;
for(i=0;i<n;i++){
    printf("enter request %d :",i+1);
    scanf("%d",&request[i]);
}
/* sorting the requests */
int min_pos,j,temp;
for(i=0;i<n;i++){
    min_pos=i;
    for(j=i+1;j<n;j++){
        if(request[min_pos]>request[j]){
            min_pos=j;
        }
    }
    temp=request[i];
    request[i]=request[min_pos];
    request[min_pos]=temp;
}
printf("\n");

/* print sorted requests*/
for(i=0;i<n;i++){
    printf("%d,",request[i]);

```

```
}  
printf("\n");
```

*/\* if header is headed toward left go to smallest cylinder that need to be serviced and after that check whether all requests are serviced or not  
ex: if the sequence is 5 7 8 23 70*

*header at 80 & direction is left  
if it reaches smallest cylinder in request(5) now it means it serviced all the requests*

*if the sequence is 5 7 8 23 70  
header is at 50 & direction is left  
now if it reaches smallest cylinder(5) from 50 still it need to cover other requests*

*If it unable to cover any one request it will start from that first cylinder and*

*reaches last cylinder in the request[] array.*

*\*/*

```
if(direction==0){  
    header_moves+=(cur_pos-request[0]);// moving toward smallest cylinder  
    if(request[n-1]>cur_pos){ // check if there are any un covered cylinders in the request  
        header_moves+=(request[n-1]-request[0]);  
    }  
}
```

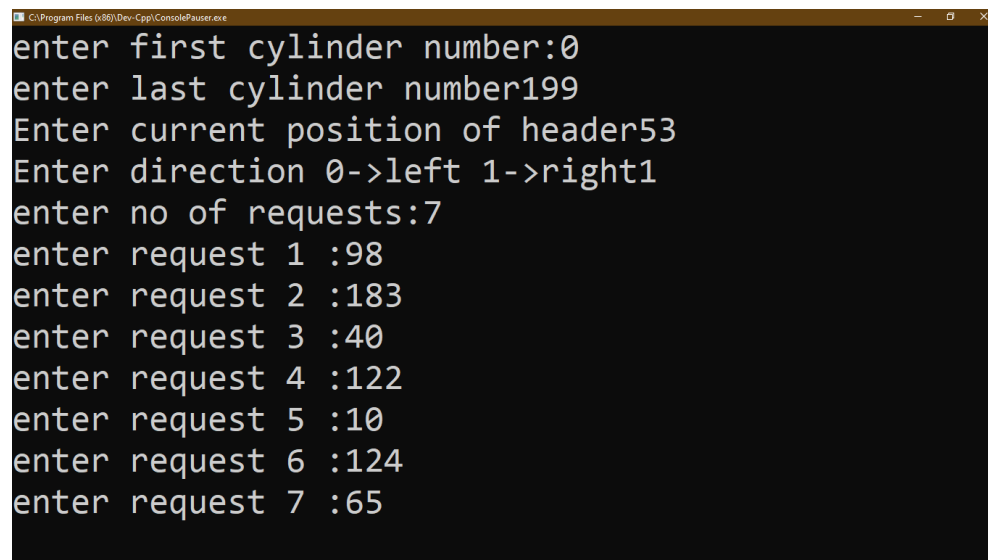
*/\* same as above but header is headed toward right  
header first reaches largest cylinder and check whether all the requests are*

*serviced or not\*/*

```
else{
    header_moves+=(request[n-1]-cur_pos);
    if(request[0]<cur_pos){
        header_moves+=(request[n-1]-request[0]);
    }
}

printf("\n no of head movements is:\t %d",header_moves);
}
```

### INPUT:

A screenshot of a Windows console window titled "C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe". The window has a black background with white text. It shows the following input sequence:

```
enter first cylinder number:0
enter last cylinder number199
Enter current position of header53
Enter direction 0->left 1->right1
enter no of requests:7
enter request 1 :98
enter request 2 :183
enter request 3 :40
enter request 4 :122
enter request 5 :10
enter request 6 :124
enter request 7 :65
```

### OUTPUT:

```
C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe

10,40,65,98,122,124,183,

no of head movements is:      303

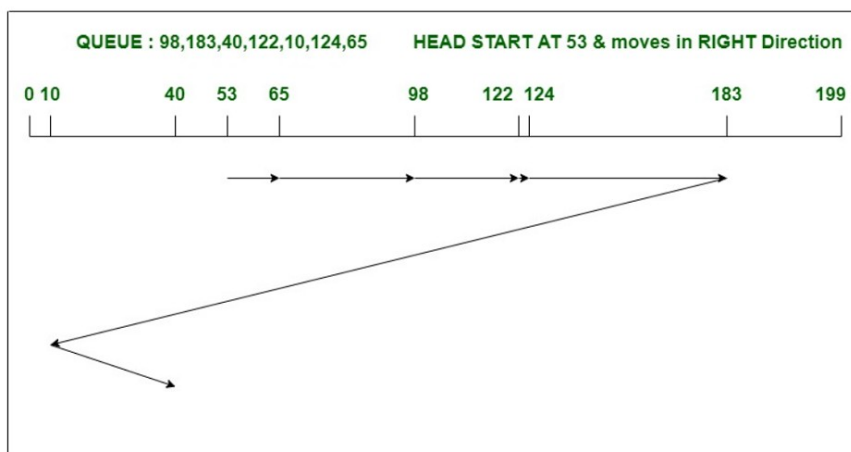
Process exited with return value 31
Press any key to continue . . .
```

## C-LOOK:

**AIM:** C program to simulate C-LOOK disk scheduling algorithm

## DESCRIPTION:

C-LOOK is the modified version of both LOOK and SCAN algorithms. In this algorithm, the head starts from first request in one direction and moves towards the last request at other end, serving all request in between. After reaching last request in one end, the head jumps in other direction and move towards the remaining requests and then satisfies them in same direction as before. Unlike LOOK, it satisfies requests only in one direction.



## Example –

Consider a disk with 200 tracks (0-199) and the disk queue having I/O requests in the following order as follows: 98, 183, 40, 122, 10, 124, 65. The current head position of the Read/Write head is 53 and will move in Right direction . Calculate the total number of track movements of Read/Write head using C-LOOK algorithm.

$$= (65-53)+(98-65)+(122-98)$$

$$+(124-122)+(183-124)+(183-10)+(40-10)$$

$$= 333$$

## CODE:

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

```
/* C LOOK is same as scan but after reaching smallest or largest cylinder based
```

```
upon direction of header it will moves from current end (largest or smallest cylinder)
```

```
to opposite end(smallest or largest cylinder).
```

```
smallest or largest cylinders are cylinders which are small / large in servicing cylinders not in cylinders of platter
```

```
now service the remaining requests*/
```

```
int main(){
```

```
    int n,header_moves,first,last,cur_pos,direction,last_visit;
```

```
    printf("enter first cylinder number:");
```

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

```
    printf("enter last cylinder number");
```

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

```
    printf("Enter current position of header");
```

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

```
    printf("Enter direction 0->left 1->right");
```

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

```
    printf("enter no of requests:");
```

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

```
    int request[n];
```

```
    int i;
```

```
    header_moves=0;
```



```

int count=0;
for(i=0;i<n;i++){
    printf("enter request %d :",i+1);
    scanf("%d",&request[i]);
}

/* sorting the requests */
int min_pos,j,temp;
for(i=0;i<n;i++){
    min_pos=i;
    for(j=i+1;j<n;j++){
        if(request[min_pos]>request[j]){
            min_pos=j;
        }
    }
    temp=request[i];
    request[i]=request[min_pos];
    request[min_pos]=temp;
}
printf("\n");

/* print sorted requests */
for(i=0;i<n;i++){
    printf("%d",request[i]);
}
printf("\n");

if(direction==0){    //for left direction

```

```

header_moves+=(cur_pos-request[0]);// moving toward smallest
cylinder
if(request[n-1]>cur_pos){
    /*check if there are any un covered cylinders in the request
    if yes go to last cylinder to service the remaining requests

    if last number in request array is grater than current
    position
    means we will not service it while moving toward left end*/
    header_moves+=(request[n-1]-request[0]);
    for(i=0;i<n;i++){ /* findout highest cylinder number
    among
    cylinders left to be serviced (we are directly finding it since
    we stop our header at that postion)*/
        if(request[i]>cur_pos){
            last_visit=request[i];
            break;
        }
    }
    header_moves+=(request[n-1]-last_visit);
}
}
/* same scenario but header is headed toward right*/
else{
    header_moves+=(request[n-1]-cur_pos);
    if(request[0]<cur_pos){
        header_moves+=(request[n-1]-request[0]);
        for(i=0;i<n;i++){

```

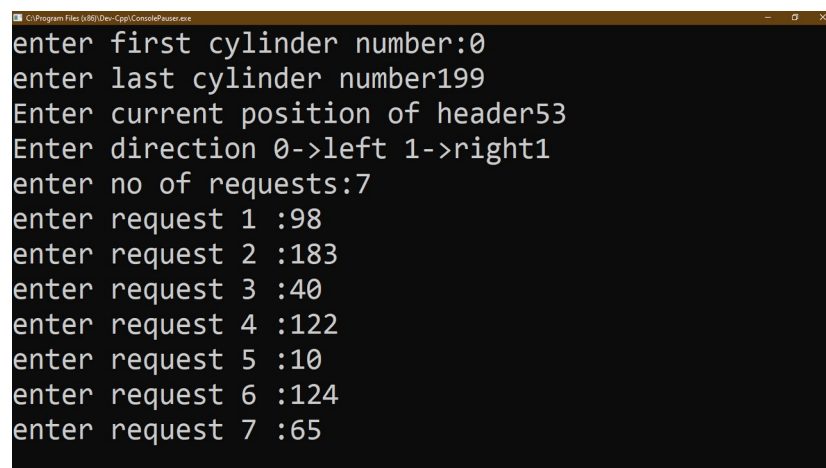
```

        if(request[i]>cur_pos){
            last_visit=request[i-1];
            break;
        }
    }
    header_moves+=(last_visit-request[0]);
}

printf("\n no of head movements is:\t %d",header_moves);
}

```

### INPUT:



A screenshot of a C++ console application window titled "C:\Program Files (x86)\Dev-Cpp\ConsolePauser.exe". The window has a black background with white text. The text shows the following input sequence:

```

enter first cylinder number:0
enter last cylinder number199
Enter current position of header53
Enter direction 0->left 1->right1
enter no of requests:7
enter request 1 :98
enter request 2 :183
enter request 3 :40
enter request 4 :122
enter request 5 :10
enter request 6 :124
enter request 7 :65

```

### OUTPUT:

```

10,40,65,98,122,124,183,

no of head movements is:      333

Process exited with return value 31
Press any key to continue . . .

```

# 11. Study and practice of Unix/Linux general purpose utility command list man,who,cad, cd, cp, ps, ls, mv, rm, mkdir, rmdir, echo, more, date, time, kill, history, chmod, chown, finger, pwd, cal, logout, shutdown.

AIM : To study and practice of Unix/Linux general purpose utility command  
list man,who,cad, cd, cp, ps, ls, mv, rm, mkdir, rmdir, echo, more, date, time,  
kill, history, chmod, chown, finger, pwd, cal, logout, shutdown.

Description:

MAN:

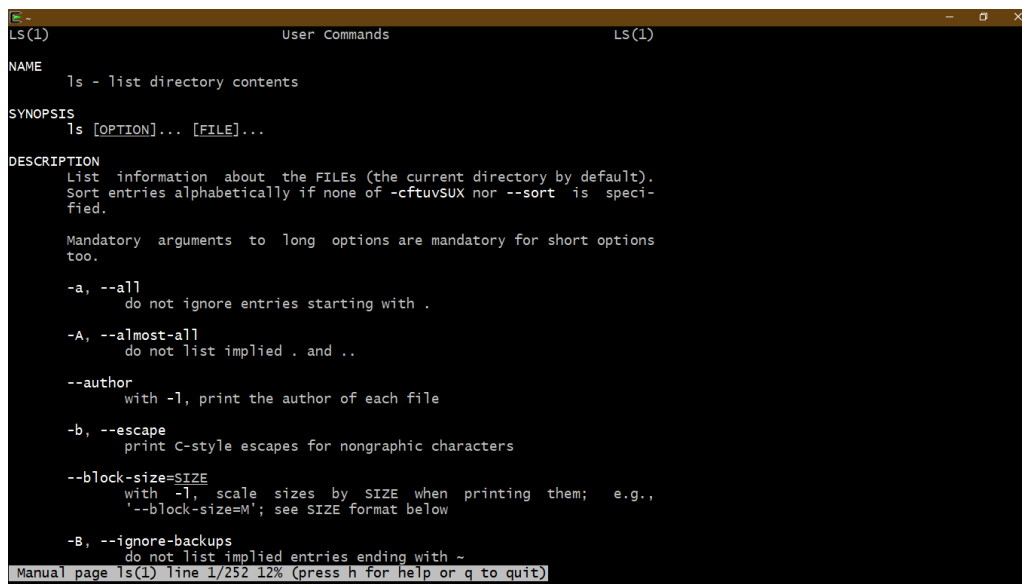
**man** : man command in Linux is used to display the user manual of any  
command that we can run on the terminal. It provides a detailed view of the  
command which includes NAME, SYNOPSIS, DESCRIPTION, OPTIONS,  
EXIT STATUS, RETURN VALUES, ERRORS, FILES, VERSIONS,  
EXAMPLES, AUTHORS .

**Syntax** : *man [OPTION]... [COMMAND NAME]*

Command name	Description	Syntax	Example
No Option	It displays the whole manual of the command	\$ man[COMMAND NAME]	\$ man printf
Section-	Since a	\$ man [SECTION-	\$ man 2 intro

Command name	Description	Syntax	Example
<b>num</b>	manual is divided into multiple sections so this option is used to display only a specific section of a manual.	NUM] [COMMAND NAME]	
<b>-a option</b>	This option helps us to display all the available intro manual pages in succession.	\$ man -a [COMMAND NAME]	\$ man -a [COMMAND NAME]
<b>-w option</b>	This option returns the location in which the manual page of a given command is present.	\$ man -w [COMMAND NAME]	\$ man -w ls
<b>-I option</b>	It considers the command as case sensitive.	\$ man -I [COMMAND NAME]	\$ man -I printf

## Example : man ls

A screenshot of a terminal window titled "User Commands" showing the output of the 'man ls' command. The output is in a dark-themed terminal with white text. It includes sections for NAME, SYNOPSIS, and DESCRIPTION. The DESCRIPTION section lists various options like -a, -A, --author, -b, --block-size, and -B, along with their functions. At the bottom, it says "Manual page ls(1) line 1/252 12% (press h for help or q to quit)".

```
LS(1) User Commands LS(1)
NAME
  ls - list directory contents
SYNOPSIS
  ls [OPTION]... [FILE]...
DESCRIPTION
  List information about the FILES (the current directory by default).
  Sort entries alphabetically if none of -cftuvSUX nor --sort is speci-
  fied.

  Mandatory arguments to long options are mandatory for short options
  too.

  -a, --all
      do not ignore entries starting with .
  -A, --almost-all
      do not list implied . and ..
  --author
      with -l, print the author of each file
  -b, --escape
      print C-style escapes for nongraphic characters
  --block-size=SIZE
      with -l, scale sizes by SIZE when printing them; e.g.,
      '--block-size=M'; see SIZE format below
  -B, --ignore-backups
      do not list implied entries ending with ~
Manual page ls(1) line 1/252 12% (press h for help or q to quit)
```

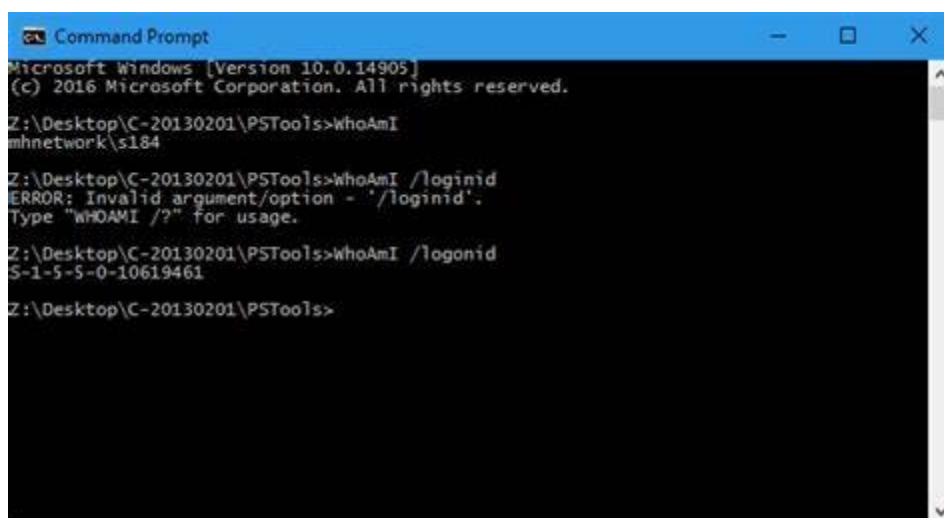
## WHO:

identifies the users currently logged in The "who" command lets you display the users that are currently logged into your UNIX computer system.

The following information is displayed: login name, workstation name, date and time of login. Entering who am i or who am I displays your login name, workstation name, date and time you logged in.

Synopsys who [OPTION]... [ FILE | ARG1 ARG2 ]

Example: who am i

A screenshot of a Windows Command Prompt window. The title bar says "Command Prompt". The text inside shows the command 'whoami' being executed, which returns 'mhnetwork\s184'. Then, the command 'whoami /loginid' is executed, which returns an error: 'ERROR: Invalid argument/option - '/loginid''. Finally, the command 'whoami /logonid' is executed, which returns 'S-1-5-S-0-10619461'.

```
Microsoft Windows [Version 10.0.14905]
(c) 2016 Microsoft Corporation. All rights reserved.

Z:\Desktop\C-20130201\PSTools>whoami
mhnetwork\s184

Z:\Desktop\C-20130201\PSTools>whoami /loginid
ERROR: Invalid argument/option - '/loginid'.
Type "WHOAMI /?" for usage.

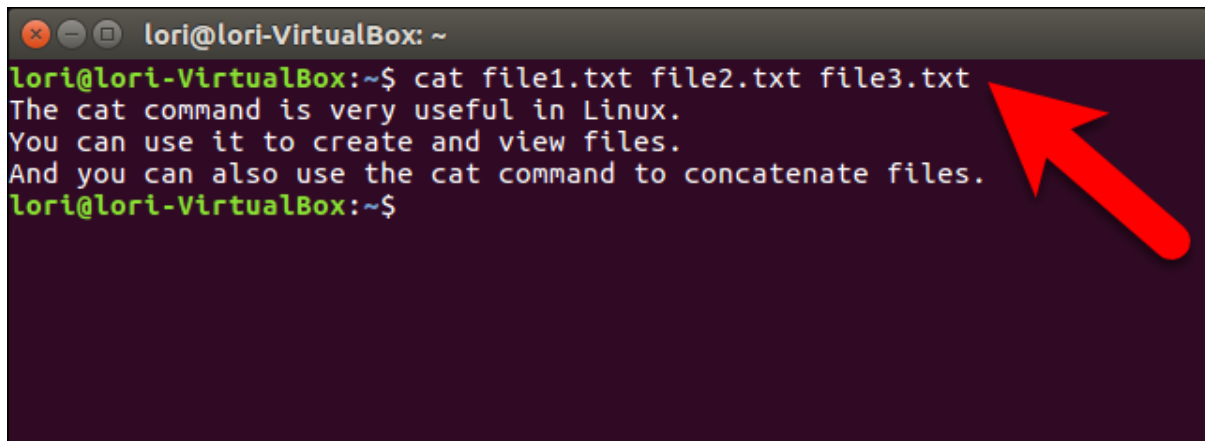
Z:\Desktop\C-20130201\PSTools>whoami /logonid
S-1-5-S-0-10619461

Z:\Desktop\C-20130201\PSTools>
```

**cat** : It is generally used to concatenate the files. It gives the output on the standard output.

**syntax :** `cat filename1 filename2`

**Example :**



```
lori@lori-VirtualBox: ~  
lori@lori-VirtualBox:~$ cat file1.txt file2.txt file3.txt  
The cat command is very useful in Linux.  
You can use it to create and view files.  
And you can also use the cat command to concatenate files.  
lori@lori-VirtualBox:~$
```

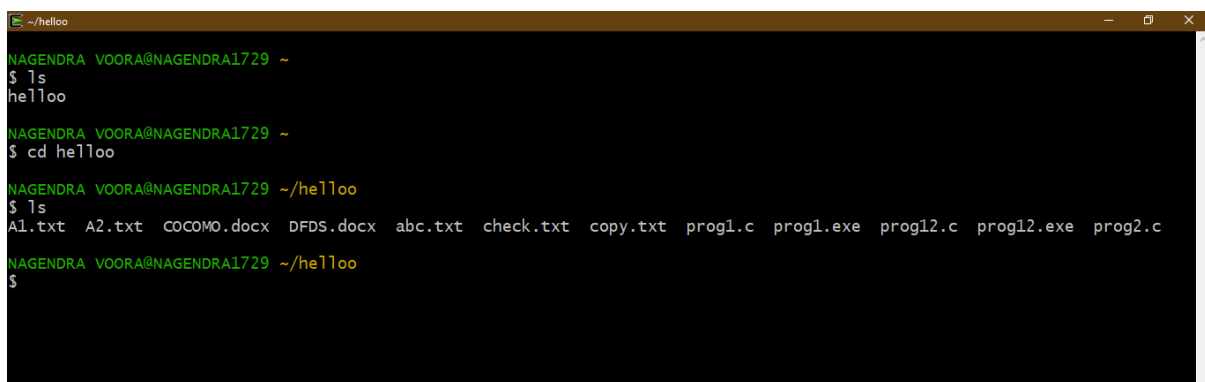
**cd :** Use the "cd" command to change directory. For example, if you are in the home folder, and you want to go to the downloads folder, then you can type in "cd Downloads".

**syntax :** `cd /Directory_path`

**example:**

*\$cd Desktop*

**ls :** If you want to see the list of files on your UNIX or Linux system, use the 'ls' command. It shows the files /directories in your current directory. The ls command will show you the list of files in your current directory.



```
NAGENDRA VOORA@NAGENDRA1729 ~  
$ ls  
helloo  
  
NAGENDRA VOORA@NAGENDRA1729 ~  
$ cd helloo  
  
NAGENDRA VOORA@NAGENDRA1729 ~/helloo  
$ ls  
A1.txt A2.txt COCOMO.docx DFDS.docx abc.txt check.txt copy.txt prog1.c prog1.exe prog12.c prog12.exe prog2.c  
NAGENDRA VOORA@NAGENDRA1729 ~/helloo  
$
```

**pwd :** When you first open the terminal, you are in the home directory of your user. To know which directory you are in, you can use the "pwd" command. It gives us the absolute path, which means the path that starts from the root. The root is the base of the Linux file system. It is denoted by a forward slash( / ).

**Syntax:** `$pwd`

```
NAGENDRA VOORA@NAGENDRA1729 ~
$ ls
helloo

NAGENDRA VOORA@NAGENDRA1729 ~
$ cd helloo

NAGENDRA VOORA@NAGENDRA1729 ~/helloo
$ ls
A1.txt  A2.txt  COCOMO.docx  DFDS.docx  abc.txt  check.txt  copy.txt  prog1.c  prog1.exe  prog12.c  prog12.exe  prog2.c

NAGENDRA VOORA@NAGENDRA1729 ~/helloo
$

NAGENDRA VOORA@NAGENDRA1729 ~/helloo
$ pwd
/home/NAGENDRA VOORA/helloo

NAGENDRA VOORA@NAGENDRA1729 ~/helloo
$
```

**mkdir & rmdir** : Use the mkdir command when you need to create a folder or a directory. Use rmdir to delete a directory. But rmdir can only be used to delete an empty directory.

Syntax : mkdir foldername

Rmdir foldername

```
NAGENDRA VOORA@NAGENDRA1729 ~/helloo
$ mkdir new_folder

NAGENDRA VOORA@NAGENDRA1729 ~/helloo
$ ls
A1.txt  COCOMO.docx  abc.txt  copy.txt  prog1.c  prog12.c  prog2.c
A2.txt  DFDS.docx  check.txt  new_folder  prog1.exe  prog12.exe

NAGENDRA VOORA@NAGENDRA1729 ~/helloo
$ rmdir new_folder

NAGENDRA VOORA@NAGENDRA1729 ~/helloo
$ ls
A1.txt  A2.txt  COCOMO.docx  DFDS.docx  abc.txt  check.txt  copy.txt  prog1.c  prog1.exe  prog12.c  prog12.exe  prog2.c

NAGENDRA VOORA@NAGENDRA1729 ~/helloo
$ |
```

**rm** : Use the rm command to delete files and directories.

*rm file or directory name*

```
shovon@linuxhint: ~
File Edit View Search Terminal Help
shovon@linuxhint:~$ rm -rfv files/
removed 'files/c-test-1'
removed directory 'files/c-test'
removed 'files/a-test-1'
removed 'files/c-test-3'
removed 'files/b-test-3'
removed 'files/b-test-2'
removed directory 'files/b-test'
removed directory 'files/a-test'
removed 'files/c-test-2'
removed 'files/a-test-2'
removed 'files/a-test-3'
removed 'files/b-test-1'
removed directory 'files/'
shovon@linuxhint:~$
```



**cp** : cp stands for copy. This command is used to copy files or group of files or directory. It creates an exact image of a file on a disk with different file name. cp command require at least two filenames in its arguments.

**Syntax** : *cp Src file Dest file*

```

[ bilwadal@localhost ~ ]$ pwd
/home/bilwadal
[ bilwadal@localhost ~ ]$ ls
accounts  bill      buc       letter.txt  office     pune      shop.txt   travel
a_jay    billo    Chennai  linux      patna      raipur    sonan      video
appln.txt bilwadal documents lottery   personal  resume.txt statement.txt
audio    biodata.txt letter    mohan     profile    ship      ticket
[ bilwadal@localhost ~ ]$ ls *.txt
letter.txt  resume.txt  shop.txt  statement.txt
[ bilwadal@localhost ~ ]$ cp *.txt raipur/
[ bilwadal@localhost ~ ]$ ls
accounts  bill      buc       letter.txt  office     pune      shop.txt   travel
a_jay    billo    Chennai  linux      patna      raipur    sonan      video
appln.txt bilwadal documents lottery   personal  resume.txt statement.txt
audio    biodata.txt letter    mohan     profile    ship      ticket
[ bilwadal@localhost ~ ]$ _
  
```

Copy command applied to copy all the text files into folder 'Raipur'

Checking how many txt files are in the present working Directory

**ps** : The ps command reports information on current running processes, outputting to standard output. It is frequently used to find process identifier numbers. It supports searching for processes by user, group, process id or executable name.

syntax: \$ps

```

NAGENDRA VOORA@NAGENDRA1729 ~/helloo
$ ps
  PID  PPID  PGID  WINPID  TTY      UID    STIME  COMMAND
 1158   1157  1158    9556   pts/0    197609 15:56:48 /usr/bin/bash
 1171   1158  1171   10440   pts/0    197609 16:07:16 /usr/bin/ps
 1157      1  1157    9060    ?        197609 15:56:48 /usr/bin/mintty
  
```

**mv** : mv stands for move. mv is used to move one or more files or directories from one place to another in file system like UNIX. It has two distinct functions:

- (i) It rename a file or folder.
- (ii) It moves group of files to different directory.

Syntax : mv [options] source destination

```

NAGENDRA VOORA@NAGENDRA1729 ~/hello
$ ls
A1.txt  COCOMO.docx  abc.txt  copyy.txt  prog1.exe  prog12.exe
A2.txt  DFDS.docx    check.txt  prog1.c    prog12.c   prog2.c

NAGENDRA VOORA@NAGENDRA1729 ~/hello
$ mv abc.txt xyz.txt

NAGENDRA VOORA@NAGENDRA1729 ~/hello
$ ls
A1.txt  COCOMO.docx  check.txt  prog1.c  prog12.c  prog2.c
A2.txt  DFDS.docx    copyy.txt  prog1.exe  prog12.exe  xyz.txt

NAGENDRA VOORA@NAGENDRA1729 ~/hello
$

```

**echo** : echo command in linux is used to display line of text/string that are passed as an argument . This is a built in command that is mostly used in shell scripts and batch files to output status text to the screen or a file.

Syntax: echo “enter text”

```

NAGENDRA VOORA@NAGENDRA1729 ~/hello
$ echo hello
hello

NAGENDRA VOORA@NAGENDRA1729 ~/hello
$ |

```

**more** : more command is used to view the text files in the command prompt, displaying one screen at a time in case the file is large (For example log files). The more command also allows the user do scroll up and down through the page.

**-d** : Use this command in order to help the user to navigate. It displays “[Press space to continue, ‘q’ to quit.]” and displays “[Press ‘h’ for instructions.]” when wrong key is pressed.

Syntax: more **-d** filename

```

NAGENDRA VOORA@NAGENDRA1729 ~/hello
$ more check.txt
hello ifocus institute

NAGENDRA VOORA@NAGENDRA1729 ~/hello
$ |

```

**date** : date command is used to display the system date and time. date command is also used to set date and time of the system.

Syntax : \$date

```

NAGENDRA VOORA@NAGENDRA1729 ~/hello
$ date
Mon Jan  9 16:14:38 IST 2023

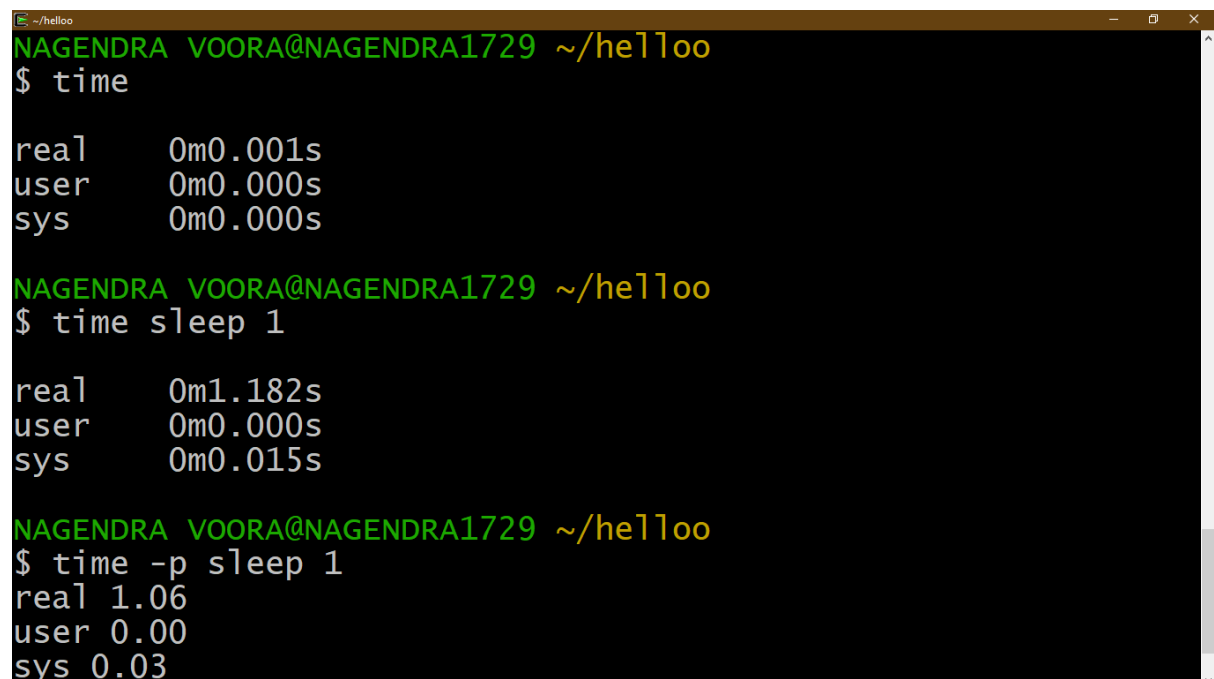
NAGENDRA VOORA@NAGENDRA1729 ~/hello
$

```

**time** : time command in Linux is used to execute a command and prints a summary of real-time, user CPU time and system CPU time spent by executing a command when it terminates. 'real' time is the time elapsed wall clock time taken by a command to get executed, while 'user' and 'sys' time are the number of CPU seconds that command uses in user and kernel mode respectively.

Syntax: time [options ] command

- **time -p** : This option is used to print time in POSIX format.
- **help time** : it displays help information.
- **sleep 1** is used to create a dummy job which lasts 1 seconds.

A terminal window titled '~/.helloo' showing the execution of the 'time' command. The prompt is 'NAGENDRA VOORA@NAGENDRA1729 ~/.helloo'. The first command is '\$ time', which outputs 'real 0m0.001s', 'user 0m0.000s', and 'sys 0m0.000s'. The second command is '\$ time sleep 1', which outputs 'real 0m1.182s', 'user 0m0.000s', and 'sys 0m0.015s'. The third command is '\$ time -p sleep 1', which outputs 'real 1.06', 'user 0.00', and 'sys 0.03'.

```
NAGENDRA VOORA@NAGENDRA1729 ~/.helloo
$ time
real    0m0.001s
user    0m0.000s
sys     0m0.000s

NAGENDRA VOORA@NAGENDRA1729 ~/.helloo
$ time sleep 1
real    0m1.182s
user    0m0.000s
sys     0m0.015s

NAGENDRA VOORA@NAGENDRA1729 ~/.helloo
$ time -p sleep 1
real 1.06
user 0.00
sys 0.03
```

**kill** : kill command in Linux (located in /bin/kill), is a built-in command which is used to terminate processes manually. kill command sends a signal to a process which terminates the process. If the user doesn't specify any signal which is to be sent along with kill command then default TERM signal is sent that terminates the process.

**kill -l** :To display all the available signals you can use below command option

```
~/helloo
$ kill -l
1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL
5) SIGTRAP     6) SIGABRT     7) SIGEMT      8) SIGFPE      9) SIGKILL     10) SIGBUS
11) SIGSEGV    12) SIGSYS     13) SIGPIPE    14) SIGALRM     15) SIGTERM
16) SIGURG     17) SIGSTOP    18) SIGTSTP    19) SIGCONT     20) SIGCHLD
21) SIGTTIN    22) SIGTTOU    23) SIGIO      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF    28) SIGWINCH   29) SIGPWR      30) SIGUSR1
31) SIGUSR2    32) SIGRTMIN   33) SIGRTMIN+1 34) SIGRTMIN+2
35) SIGRTMIN+3 36) SIGRTMIN+4 37) SIGRTMIN+5 38) SIGRTMIN+6 39) SIGRTMIN+7
40) SIGRTMIN+8 41) SIGRTMIN+9 42) SIGRTMIN+10 43) SIGRTMIN+11 44) SIGRTMIN+12
45) SIGRTMIN+13 46) SIGRTMIN+14 47) SIGRTMIN+15 48) SIGRTMIN+16 49) SIGRTMAX-15
50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6  59) SIGRTMAX-5
60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2  63) SIGRTMAX-1  64) SIGRTMAX
```

**history** : history command is used to view the previously executed command. This feature was not available in the Bourne shell. Bash and Korn support this feature in which every command executed is treated as the event and is associated with an event number using which they can be recalled and changed if required. These commands are saved in a history file. In Bash shell history command shows the whole list of the command.

Syntax: \$history

```
~/helloo
99  chmod u=r COCOMO.docx
100 chmod u=r COCOMO.docx
101 ls -l
102 ifconfig
103 man ls
104 who a i
105 who am i
106 who
107 man who
108 who -a
109 who -l
110 who am i
111 ls
112 cd helloo
113 ls
114 pwd
115 mkdir new_folder
116 ls
117 rmdir new_folder
118 ls
119 ps
120 mv copy.txt copyy.txt
121 ls
122 ls
123 mv abc.txt xyz.txt
124 ls
125 echo hello
126 more check.txt
127 date
128 time
129 time sleep 1
130 time -p sleep 1
131 kill -l
132 history
```

**cal** : cal command is a calendar command in Linux which is used to see the calendar of a specific month or a whole year.

Syntax: \$cal [ [ month ] year ]

```
~/helloo
131 kill -l
132 history

NAGENDRA VOORA@NAGENDRA1729 ~/helloo
$ cal
    January 2023
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31

NAGENDRA VOORA@NAGENDRA1729 ~/helloo
$ cal july 2022
    July 2022
Su Mo Tu We Th Fr Sa
                 1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31

NAGENDRA VOORA@NAGENDRA1729 ~/helloo
$ |
```

**logout :** logout command allows you to programmatically logout from your session. causes the session manager to take the requested action immediately.

Syntax: \$logout

→ closes the cmd

## CHMOD:

In Unix-like operating systems, the **chmod** command is used to change the access mode of a file.

The name is an abbreviation of **change mode**.

Syntax: chmod [reference][operator][mode] file...

The references are used to distinguish the users to whom the permissions apply i.e. they are list of letters that specifies whom to give permissions. The references are represented by one or more of the following letters:

Reference	Class	Description
u	owner	file's owner
g	group	users who are members of the file's group
o	others	users who are neither the

file's owner nor members of  
the file's group

a      all      All three of the above, same as ugo

The operator is used to specify how the modes of a file should be adjusted.  
The following operators are accepted:

Operator    Description

- +      Adds the specified modes to the  
         specified classes
- Removes the specified modes from  
         the specified classes
- =      The modes specified are to be made  
         the exact modes for the specified  
         classes

**Note :** Putting blank space(s) around operator would make the command fail.  
The modes indicate which permissions are to be granted or removed from the  
specified classes. There are three basic modes which correspond to the basic  
permissions:

- r      Permission to read the file.
- w      Permission to write (or delete) the file.
- x      Permission to execute the file, or, in  
         the case of a directory, search it.

BEFORE:

```
rwxr-xr-x 1 NAGENDRA VOORA None    10 Dec 17 16:13 A1.txt
```

AFTER:

```
-r--r-xr-x 1 NAGENDRA VOORA None    10 Dec 17 16:13 A1.txt
```

```
~ /helloo
NAGENDRA VOORA@NAGENDRA1729 ~/helloo
$ ls -l
total 402
-rwxr-xr-x 1 NAGENDRA VOORA None 10 Dec 17 16:13 A1.txt
-rwxr-xr-x 1 NAGENDRA VOORA None 37 Jan 2 08:19 A2.txt
-r--r-xr-x 1 NAGENDRA VOORA None 19854 Nov 28 16:10 CCOMO.docx
-rwxr-xr-x 1 NAGENDRA VOORA None 128408 Dec 5 15:39 DFDS.docx
-rwxr-xr-x 1 NAGENDRA VOORA None 50 Dec 18 20:41 check.txt
-rwxr-xr-x 1 NAGENDRA VOORA None 10 Dec 17 16:20 copyy.txt
-rwxr-xr-x 1 NAGENDRA VOORA None 538 Dec 18 20:43 prog1.c
-rwxr-xr-x 1 NAGENDRA VOORA None 123053 Dec 18 20:41 prog1.exe
-rwxr-xr-x 1 NAGENDRA VOORA None 628 Jan 2 07:41 prog12.c
-rwxr-xr-x 1 NAGENDRA VOORA None 122568 Jan 2 08:20 prog12.exe
-rwxr-xr-x 1 NAGENDRA VOORA None 2246 Dec 18 21:00 prog2.c
-r-xr-xr-x 1 NAGENDRA VOORA None 0 Dec 18 20:41 xyz.txt

NAGENDRA VOORA@NAGENDRA1729 ~/helloo
$ chmod u=r A1.txt

NAGENDRA VOORA@NAGENDRA1729 ~/helloo
$ ls -l
total 402
-r--r-xr-x 1 NAGENDRA VOORA None 10 Dec 17 16:13 A1.txt
-rwxr-xr-x 1 NAGENDRA VOORA None 37 Jan 2 08:19 A2.txt
-r--r-xr-x 1 NAGENDRA VOORA None 19854 Nov 28 16:10 CCOMO.docx
-rwxr-xr-x 1 NAGENDRA VOORA None 128408 Dec 5 15:39 DFDS.docx
-rwxr-xr-x 1 NAGENDRA VOORA None 50 Dec 18 20:41 check.txt
-rwxr-xr-x 1 NAGENDRA VOORA None 10 Dec 17 16:20 copyy.txt
-rwxr-xr-x 1 NAGENDRA VOORA None 538 Dec 18 20:43 prog1.c
-rwxr-xr-x 1 NAGENDRA VOORA None 123053 Dec 18 20:41 prog1.exe
-rwxr-xr-x 1 NAGENDRA VOORA None 628 Jan 2 07:41 prog12.c
-rwxr-xr-x 1 NAGENDRA VOORA None 122568 Jan 2 08:20 prog12.exe
-rwxr-xr-x 1 NAGENDRA VOORA None 2246 Dec 18 21:00 prog2.c
-r-xr-xr-x 1 NAGENDRA VOORA None 0 Dec 18 20:41 xyz.txt
```

**chown** command is used to change the file Owner or group. Whenever you want to change ownership you can use chown command.

### Syntax:

chown [OPTION]... [OWNER][:[GROUP]] FILE...

chown [OPTION]... --reference=RFILE FILE...

**Example:** To change owner of the file:

chown owner\_name file\_name

```
root@kali:~# ls -l
total 80
drwxr-xr-x 2 root root 4096 Feb 2 05:29 Desktop
drwxr-xr-x 2 root root 4096 Feb 2 05:05 Documents
drwxr-xr-x 2 root root 4096 Feb 3 06:41 Downloads
-rw-r--r-- 1 root root 202 Feb 4 12:08 example.a
-rw-r--r-- 1 master group1 12 Feb 4 12:04 file1.txt
-rw-r--r-- 1 master group1 61 Feb 4 12:06 file2.txt
-rw-r--r-- 1 root group1 7 Feb 4 12:09 greek1
-rw-r--r-- 1 master group1 6 Feb 4 12:10 greek2
-rw-r--r-- 1 master group1 6 Feb 4 12:21 greek3
-rw-r--r-- 1 root root 208 Feb 4 12:23 greeks.a
drwxr-xr-x 2 root root 4096 Feb 2 05:05 Music
drwxr-xr-x 2 root root 4096 Feb 2 05:05 Pictures
drwxr-xr-x 2 root root 4096 Feb 2 05:05 Public
-rw-r--r-- 1 root root 6 Feb 5 14:04 star1.txt
-rw-r--r-- 1 root root 7 Feb 5 14:01 star2.txt
-rw-r--r-- 1 root root 6 Feb 5 14:01 star3.txt
-rw-r--r-- 1 root root 8 Feb 5 12:33 star.a
-rw-r--r-- 1 root root 208 Feb 5 13:12 super.a
drwxr-xr-x 2 root root 4096 Feb 2 05:05 Templates
drwxr-xr-x 2 root root 4096 Feb 2 05:05 Videos
root@kali:~#
```

```
root : bash — Konsole
root@kali:~# ls -l file1.txt
-rw-r--r-- 1 root root 12 Feb  4 12:04 file1.txt
root@kali:~# chown master file1.txt
root@kali:~# ls -l file1.txt
-rw-r--r-- 1 master root 12 Feb  4 12:04 file1.txt
root@kali:~#
```

**NOTE:** here master is another user of the system

## Finger:

The finger command in Linux is basically a user information lookup program. Following is its syntax:

The finger command in Linux is basically a user information lookup program. Following is its syntax:

```
finger [-lmsp] [user ...] [user@host ...]
```

And here's how the tool's man page explains it:

The finger displays information about the system users.

Basic usage is simple, just execute 'finger' with name of a user as input. Here's an example:

```
finger himanshu
```

And here's the output the above command produced on my system:

```
Login: himanshu           Name: Himanshu
Directory: /home/himanshu  Shell: /bin/bash
On since Sat Nov 24 10:16 (IST) on :0 from :0 (messages off)
No mail.
No Plan.
```

## Shutdown:



The **shutdown** command in Linux is used to shutdown the system in a safe way. You can shutdown the machine immediately, or schedule a shutdown using 24 hour format. It brings the system down in a secure way. When the shutdown is initiated, all logged-in users and processes are notified that the system is going down, and no further logins are allowed.

Only root user can execute shutdown command.

### **Syntax of shutdown Command**

shutdown [OPTIONS] [TIME] [MESSAGE]

⌘ options – Shutdown options such as halt, power-off (the default option) or reboot the system.

⌘ time – The time argument specifies when to perform the shutdown process.

⌘ message – The message argument specifies a message which will be broadcast to all users.

### **Options**

-r : Requests that the system be rebooted after it has been brought down.

-h : Requests that the system be either halted or powered off after it has been brought down, with the choice as to which left up to the system.

-H : Requests that the system be halted after it has been brought down.

-P : Requests that the system be powered off after it has been brought down.

-c : Cancels a running shutdown. TIME is not specified with this option, the first argument is MESSAGE.

-k : Only send out the warning messages and disable logins, do not actually bring the system down.

## **12. Write a C program that makes a copy of a file using standard I/O, and system calls**

**AIM:** To write a C program to make a copy of file using standard I/O and system calls

### **DESCRIPTION:**

#### **Syntax in C language**

size\_t read (int fd, void\* buf, size\_t cnt);

#### **Parameters:**

- **fd:** file descriptor
- **buf:** buffer to read data from
- **cnt:** length of buffer

#### **Returns: How many bytes were actually read**

- return Number of bytes read on success
- return 0 on reaching end of file
- return -1 on error
- return -1 on signal interrupt

**write:** Writes cnt bytes from buf to the file or socket associated with fd. cnt should not be greater than INT\_MAX (defined in the limits.h header file). If cnt is zero, write() simply returns 0 without attempting any other action.

- #include <fcntl.h>
- size\_t write (int fd, void\* buf, size\_t cnt);

## CODE:

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

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

```
#include <fcntl.h>
```

```
void typefile (char *filename)
```

```
{
```

```
    int fd, nread;
```

```
    char buf[1024];           // buffer to store data copied from the source
```

```
    fd = open (filename, O_RDONLY); // to open in read only format
```

```
    if (fd == -1) {           // fd returns -1 if file is unavailable
```

```
        perror (filename);
```

```
        return;
```

```
    }                        // if available read the file
```

```
    while ((nread = read (fd, buf, sizeof (buf))) > 0)
```

```
        write (1, buf, nread);
```

```
        close (fd);
```

```
}
```

```
int main ()
```

```
{
```

```
    char *files={"A2.txt"};
```

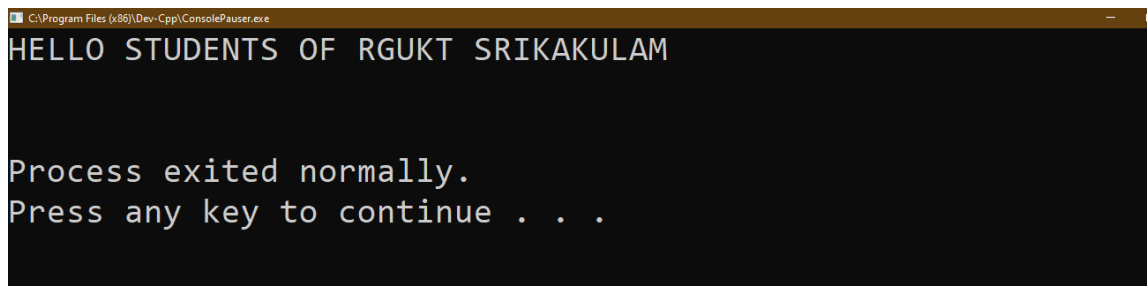
```
    typefile (files);
```

```
    exit (0);  
}
```

## FILE:



## OUTPUT:



## 13. Write a C program to emulate the UNIX ls -l command.

**Aim:** - C program to emulate the UNIX ls -l command.

**Theory:** -

### UNIX ls -l command

ls is a Linux shell command that lists directory contents of files and directories

\$ ls -l : To show long listing information about the file/directory.

**Example:** -

```
-rwxrwxrwx 1 sivasai sivasai 310 Jan 9 00:16 ls.c  
  ↑      ↑      ↑      ↑      ↑      ↑      ↑  
Field 1   2      3      4      5      6      7
```

- **Field 1 – File Permissions:** First 9 character specifies the files permission. The every 3 characters specifies read, write, execute permissions for user(root), group and others respectively in order. Taking above example, -rw-rw-r- indicates read-write permission for user(root) , read permission for group, and no permission for others respectively. If all three permissions are given to user(root), group and others, the format looks like -rwxrwxrwx
- **Field 2 – Number of links:** Second field specifies the number of links for that file. In this example, 1 indicates only one link to this file.
- **Field 3 – Owner:** Third field specifies owner of the file. In this example, this file is owned by username 'sivasai'.
- **Field 4 – Group:** Fourth field specifies the group of the file. In this example, this file belongs to "sivasai group.
- **Field 5 – Size:** Fifth field specifies the size of file in bytes. In this example, '310 indicates the file size in bytes.
- **Field 6 – Last modified date and time:** Sixth field specifies the date and time of the last modification of the file. In this example, 'Jan 9 00:16' specifies the last modification time of the file.
- **Field 7 – File name:** The last field is the name of the file. In this example, the file name is ls.c.

### Expected Output in UNIX CMD:-

```
sivasai@Siva: /mnt/c/users/siva/desktop/os/ubuntu
sivasai@Siva:/mnt/c/users/siva/desktop/os/ubuntu$ ls -l
total 20
-rwxrwxrwx 1 sivasai sivasai 16128 Jan  9 00:16 a.out
-rwxrwxrwx 1 sivasai sivasai  150 Dec 23 16:36 fork1.c
-rwxrwxrwx 1 sivasai sivasai  310 Jan  9 00:16 ls.c
-rwxrwxrwx 1 sivasai sivasai  289 Jan  9 00:10 ls1.c
-rwxrwxrwx 1 sivasai sivasai  536 Dec 22 23:31 open.c
-rwxrwxrwx 1 sivasai sivasai  305 Dec 22 23:36 open2.c
-rwxrwxrwx 1 sivasai sivasai  232 Dec 22 22:44 read.c
-rwxrwxrwx 1 sivasai sivasai   25 Dec 22 23:17 text1.txt
-rwxrwxrwx 1 sivasai sivasai   10 Dec 23 10:15 towrite.txt
-rwxrwxrwx 1 sivasai sivasai  151 Dec 22 22:31 write.c
-rwxrwxrwx 1 sivasai sivasai  140 Dec 22 22:31 write2.c
sivasai@Siva:/mnt/c/users/siva/desktop/os/ubuntu$
```

### C Programming:-

Input:

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

```

#include<unistd.h>
void main()
{
    int pid;
    pid=fork();
    if(pid==0)
    {
        execlp("/bin/ls","ls","-l",NULL);
    }
    else
    {
        wait(NULL);
        printf("child complete\n");
    }
}

```

## Output:-

```

sivasai@Siva: /mnt/c/users/siva/desktop/os/ubuntu
sivasai@Siva:/mnt/c/users/siva/desktop/os/ubuntu$ gcc ls.c
ls.c: In function 'main':
ls.c:17:9: warning: implicit declaration of function 'wait' [-Wimplicit-function-declaration]
   17 |         wait(NULL);
      |         ^~~~~
sivasai@Siva:/mnt/c/users/siva/desktop/os/ubuntu$ ./a.out
total 20
-rwxrwxrwx 1 sivasai sivasai 16128 Jan  9 00:16 a.out
-rwxrwxrwx 1 sivasai sivasai  150 Dec 23 16:36 fork1.c
-rwxrwxrwx 1 sivasai sivasai  310 Jan  9 00:16 ls.c
-rwxrwxrwx 1 sivasai sivasai  289 Jan  9 00:10 ls1.c
-rwxrwxrwx 1 sivasai sivasai  536 Dec 22 23:31 open.c
-rwxrwxrwx 1 sivasai sivasai  305 Dec 22 23:36 open2.c
-rwxrwxrwx 1 sivasai sivasai  232 Dec 22 22:44 read.c
-rwxrwxrwx 1 sivasai sivasai   25 Dec 22 23:17 text1.txt
-rwxrwxrwx 1 sivasai sivasai   10 Dec 23 10:15 towrite.txt
-rwxrwxrwx 1 sivasai sivasai  151 Dec 22 22:31 write.c
-rwxrwxrwx 1 sivasai sivasai  140 Dec 22 22:31 write2.c
child complete
sivasai@Siva:/mnt/c/users/siva/desktop/os/ubuntu$ _

```

## Code Explanation: -

The `execlp` system call duplicates the actions of the shell in searching for an executable file if the specified file name does not contain a slash (/) character. The search path is the path specified in the environment by the `PATH` variable. If this variable isn't specified, the default path `"/bin:/usr/bin"` is used.

```
execlp("/bin/ls","-ls","-l", NULL );
```

first argument is location

second argument is command

third argument is passing argument for command

NULL is default

### **Main Code:-**

- 1.create a system call fork with pid variable
2. if child Process is created then run ls -l command using execlp
- 3.if parent process is created then print "child Completed"

## **14. Write a C program that illustrates how to execute two commands concurrently with a command pipe. Ex: - ls -l | sort**

### **Aim: -**

C program that illustrates how to execute two commands concurrently with a command pipe

### **Theory: -**

### **PIPE | in UNIX**

The 'pipe' command is used in both UNIX and Linux operating systems. Pipes help combine two or more commands and are used as input/output concepts in a command. In the Linux operating system, we use more than one pipe in command so that the output of one command before a pipe acts as input for the other command after the pipe

### **Syntax; -**

**Command 1 | Command 2 | Command 3 | Command 4 | .....**

```

sivasai@Siva:/mnt/c/users/siva/desktop/os/ubuntu$ ls -l|sort
-rwxrwxrwx 1 sivasai sivasai 10 Dec 23 10:15 towrite.txt
-rwxrwxrwx 1 sivasai sivasai 25 Dec 22 23:17 text1.txt
-rwxrwxrwx 1 sivasai sivasai 140 Dec 22 22:31 write2.c
-rwxrwxrwx 1 sivasai sivasai 150 Dec 23 16:36 fork1.c
-rwxrwxrwx 1 sivasai sivasai 151 Dec 22 22:31 write.c
-rwxrwxrwx 1 sivasai sivasai 232 Dec 22 22:44 read.c
-rwxrwxrwx 1 sivasai sivasai 244 Jan 9 00:56 ls.c
-rwxrwxrwx 1 sivasai sivasai 289 Jan 9 00:10 ls1.c
-rwxrwxrwx 1 sivasai sivasai 305 Dec 22 23:36 open2.c
-rwxrwxrwx 1 sivasai sivasai 536 Dec 22 23:31 open.c
-rwxrwxrwx 1 sivasai sivasai 16080 Jan 9 00:56 a.out
total 20
sivasai@Siva:/mnt/c/users/siva/desktop/os/ubuntu$ _

```

### Explanation: -

Output of the ls -l command is the Input for the sort

So, its consider all lines as individual words and sort it by ascending order

### Input: -

```

#include<stdio.h>
#include<unistd.h>
void main()
{
    int pfd[2],p;
    pipe(pfd);
    p=fork();
    if(p==0)//for child
    {
        close(pfd[0]);
        close(1);
        dup(pfd[1]);
        execlp("/bin/ls","ls","-l",(char *)0);
    }
    else
    {

```

```

close(pfd[1]);
close(0);
dup(pfd[0]);
execlp("sort","sort ",(char *)0);
}
}

```

Output:-

```

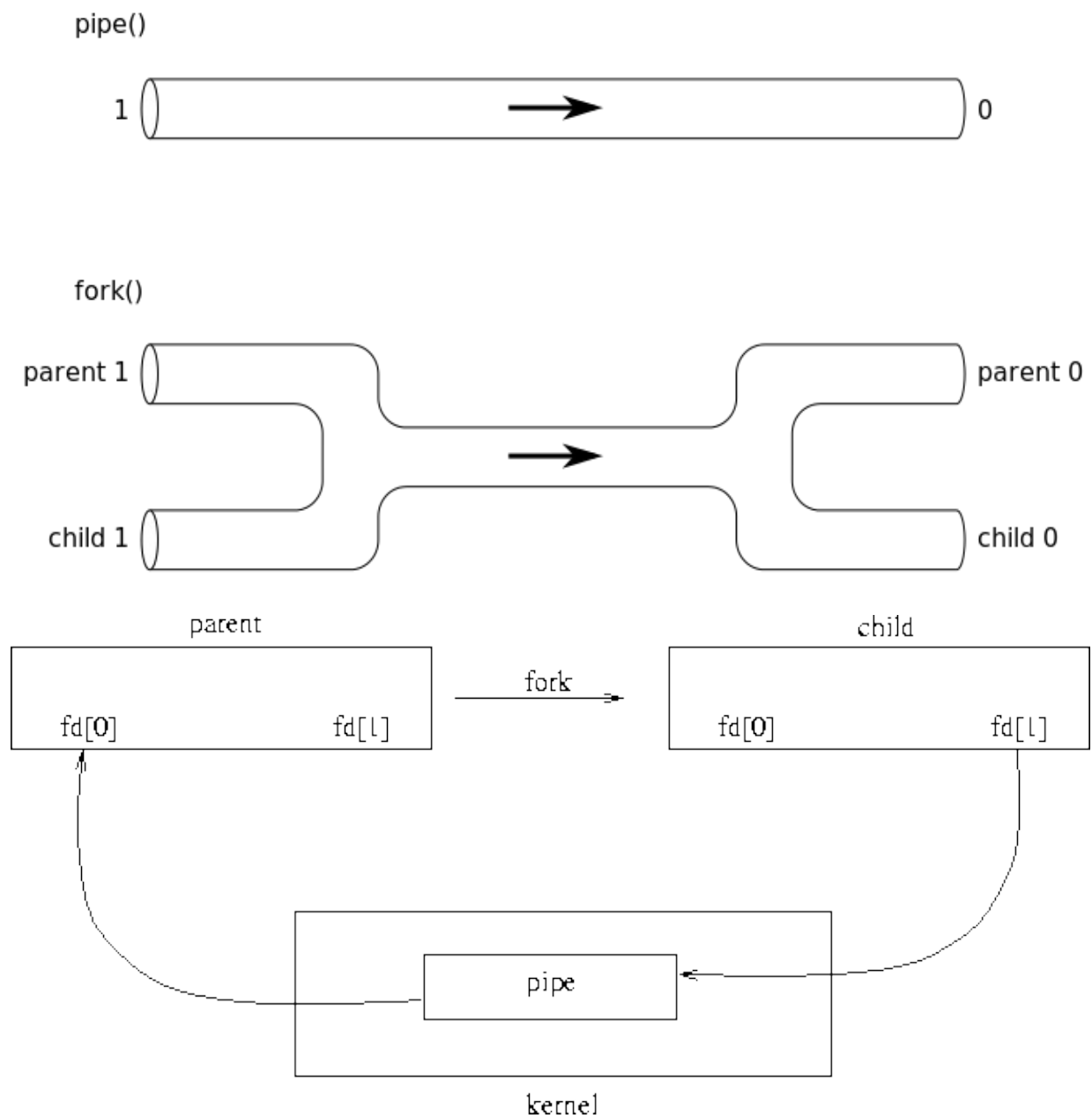
sivasai@Siva:/mnt/c/users/siva/desktop/os/ubuntu$ gcc ls1.c
sivasai@Siva:/mnt/c/users/siva/desktop/os/ubuntu$ ./a.out
-rwxrwxrwx 1 sivasai sivasai 10 Dec 23 10:15 towrite.txt
-rwxrwxrwx 1 sivasai sivasai 25 Dec 22 23:17 text1.txt
-rwxrwxrwx 1 sivasai sivasai 140 Dec 22 22:31 write2.c
-rwxrwxrwx 1 sivasai sivasai 150 Dec 23 16:36 fork1.c
-rwxrwxrwx 1 sivasai sivasai 151 Dec 22 22:31 write.c
-rwxrwxrwx 1 sivasai sivasai 232 Dec 22 22:44 read.c
-rwxrwxrwx 1 sivasai sivasai 244 Jan 9 00:56 ls.c
-rwxrwxrwx 1 sivasai sivasai 305 Dec 22 23:36 open2.c
-rwxrwxrwx 1 sivasai sivasai 390 Jan 9 14:06 ls1.c
-rwxrwxrwx 1 sivasai sivasai 536 Dec 22 23:31 open.c
-rwxrwxrwx 1 sivasai sivasai 16176 Jan 9 14:06 a.out
total 20
sivasai@Siva:/mnt/c/users/siva/desktop/os/ubuntu$

```

### Explanation: -

- Pipe is one-way communication only i.e. we can use a pipe such that One process writes to the pipe, and the other process reads from the pipe. It opens a pipe, which is an area of main memory that is treated as a “*virtual file*”.
- The pipe can be used by the creating process, as well as all its child processes, for reading and writing. One process can write to this “virtual file” or pipe and another related process can read from it.
- If a process tries to read before something is written to the pipe, the process is suspended until something is written.
- The pipe system call finds the first two available positions in the process’s open file table and allocates them for the read and write ends of the pipe.





15.

a) Study of Bash shell, Bourne shell and C shell in Unix/Linux operating system.

b) Study of Unix/Linux file system (tree structure).

c) Study of .bashrc, /etc/bashrc and Environment variables.

**Aim:** - To study

- a) Study of Bash shell, Bourne shell and C shell in Unix/Linux operating system.
- b) Study of Unix/Linux file system (tree structure).
- c) Study of .bashrc, /etc/bashrc and Environment variables.

**Theory:** -

**a) Study of Bash shell, Bourne shell and C shell in Unix/Linux operating system.**

**Shell :** SHELL is a program which provides the interface between the user and an operating system. When the user logs in OS starts a shell for user.

### **Types of Shells in Linux**

- Bourne shell
- Bash shell
- C shell
- Korn shell

**Bourne shell :** The Bourne shell, called "sh," is one of the original shells, developed for Unix computers by Stephen Bourne at AT&T's Bell Labs in 1977. Its long history of use means many software developers are familiar with it. It offers features such as input and output redirection, shell scripting with string and integer variables, and condition testing and looping.

**Bash shell :** The popularity of sh motivated programmers to develop a shell that was compatible with it, but with several enhancements. Linux systems still offer the sh shell, but "bash" -- the "Bourne-again Shell," based on sh -- has become the new default standard. One attractive feature of bash is its ability to run sh shell scripts unchanged. Shell scripts are complex sets of commands that automate programming and maintenance chores; being able to reuse these scripts saves programmers time. Conveniences not present with the original Bourne shell include command completion and a command history.

**C shell :** Developers have written large parts of the Linux operating system in the C and C++ languages. Using C syntax as a model, Bill Joy at Berkeley University developed the "C-shell," csh, in 1978. Ken Greer, working at Carnegie-Mellon University, took csh concepts a step forward with a new shell, tcsh, which Linux systems now offer. Tcsh fixed problems in csh and added command completion, in which the shell makes educated "guesses" as you type,

based on your system's directory structure and files. Tcsh does not run bash scripts, as the two have substantial differences.

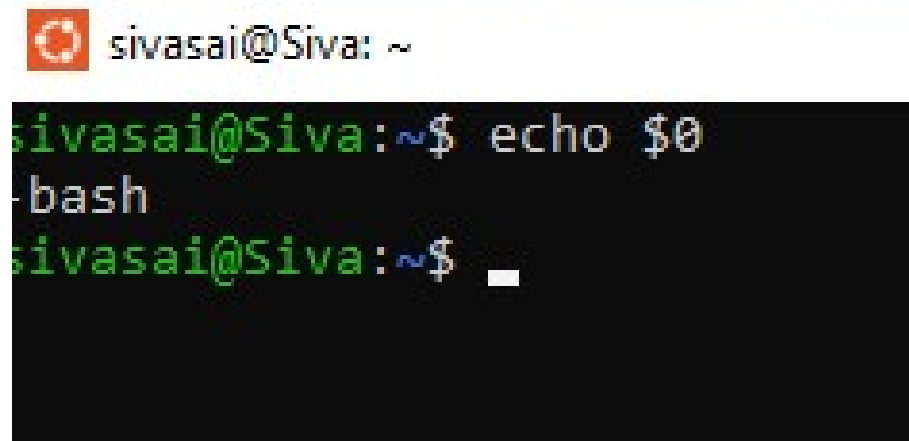
### **How do I check which shell I am using?**

You can type the following command in your terminal to see which shell you are using:

```
echo $0
```

The result will look something similar to the below if you are using the *bash* (**B**ourne **A**gain **S**hell) terminal:

```
-bash
```

A screenshot of a terminal window. At the top, there is a status bar with a red circular icon containing a white 'C' and the text 'sivasai@Siva: ~'. Below this, the terminal shows a command prompt 'sivasai@Siva:~\$' followed by the command 'echo \$0'. The output of the command is '-bash', which is displayed on the next line. The prompt 'sivasai@Siva:~\$' appears again on the following line, followed by a white cursor bar.

### **b) Study of Unix/Linux file system (tree structure).**

A file system is a logical collection of files on a partition or disk. UNIX uses a hierarchical file system structure, much like an upside-down tree, with root (/) at the base of the file system and all other directories spreading from there.

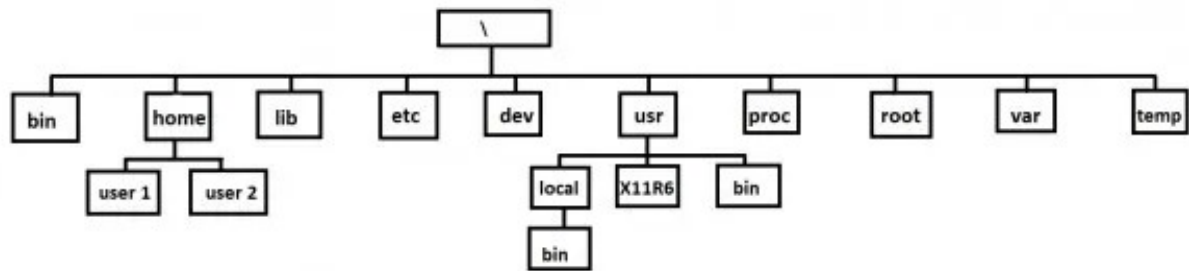


figure-1  
to show linux is a hierarchical structure with upside down tree .

A UNIX filesystem is a collection of files and directories that has the following properties :

1. It has a root directory (/) that contains other files and directories.
  2. Each file or directory is uniquely identified by its name, the directory in which it resides, and a unique identifier, typically called an inode.
- By convention, the root directory has an inode number of 2 and the lost+found directory has an inode number of 3. Inode numbers 0 and 1 are not used. File inode numbers can be seen by specifying the -i option to ls command.
  - It is self contained. There are no dependencies between one filesystem and any other.

The directories have specific purposes and generally hold the same types of information for easily locating files. Following are the directories that exist on the major versions of Unix:

Directory	Description
/	This is the root directory which should contain only the directories needed at the top level of the file structure
/bin	This is where the executable files are located. They are available to all user.
/dev	These are device drivers.

Directory	Description
/etc	Supervisor directory commands, configuration files, disk configuration files, valid user lists, groups, ethernet, hosts, where to send critical messages
/lib	Contains shared library files and sometimes other kernel-related files
/boot	Contains files for booting the system.
/home	Contains the home directory for users and other accounts
/mnt	Used to mount other temporary file systems, such as cdrom and floppy for the CDROM drive and floppy diskette drive, respectively
/proc	Contains all processes marked as a file by process number or other information that is dynamic to the system
/tmp	Holds temporary files used between system boots
/user	Used for miscellaneous purposes, or can be used by many users. Includes administrative commands, shared files, library files, and others
/var	Typically contains variable-length files such as log and print files and any other type of file that may contain a variable amount of data
/sbin	Contains binary (executable) files, usually for system administration. For example fdisk and ifconfig utilities.

Directory	Description
/kernel	Contains kernel files

### c). Study of .bashrc, /etc/bashrc and Environment variables

**Environment variables** or **ENVs** basically define the behavior of the environment. They can affect the processes ongoing or the programs that are executed in the environment.

Scope of an environment variable

Scope of any variable is the region from which it can be accessed or over which it is defined. An environment variable in Linux can have **global** or **local** scope.

#### **Global**

A globally scoped ENV that is defined in a terminal can be accessed from anywhere in that particular environment which exists in the terminal. That means it can be used in all kind of scripts, programs or processes running in the environment bound by that terminal.

#### **Local**

A locally scoped ENV that is defined in a terminal cannot be accessed by any program or process running in the terminal. It can only be accessed by the terminal( in which it was defined) itself.

Following is the partial list of important environment variables :-

1. **DISPLAY** : Contains the identifier for the display that X11 programs should use by default.
2. **HOME**: Indicates the home directory of the current user: the default argument for the cd built-in command.
3. **IFS**: Indicates the Internal Field Separator that is used by the parser for word splitting after expansion.
4. **LANG**: LANG expands to the default system locale; LC\_ALL can be used to override this. For example, if its value is pt\_BR, then the language is set to (Brazilian) Portuguese and the locale to Brazil.

5. **LD\_LIBRARY\_PATH:** On many Unix systems with a dynamic linker, contains a colon-separated list of directories that the dynamic linker should search for shared objects when building a process image after exec, before searching in any other directories.
6. **PATH:** Indicates search path for commands. It is a colon-separated list of directories in which the shell looks for commands.
7. **PWD:** Indicates the current working directory as set by the cd command.
8. **RANDOM:** Generates a random integer between 0 and 32,767 each time it is referenced.
9. **SHLVL:** Increments by one each time an instance of bash is started. This variable is useful for determining whether the built-in exit command ends the current session.
10. **TERM:** Refers to the display type
11. **VZ:** Refers to Time zone. It can take values like GMT, AST, etc.
12. **UID:** Expands to the numeric user ID of the current user, initialized at shell startup

### **.bashrc**

The **.bashrc** file is a script file that's executed when a user logs in. The file itself contains a series of configurations for the terminal session. This includes setting up or enabling: coloring, completion, shell history, command aliases, and more.

bashrc can be used to define functions that reduce redundant efforts. These functions can be a collection of basic commands. These functions can even use arguments from the terminal.

Aliases are different names for the same command. Consider them as shortcuts to a longer form command. The .bashrc file already has a set of predefined aliases.

### **/etc/bashrc**

Same as .bashrc

The /etc/bashrc is referred to as the /etc/bash.bashrc on some Linux distros. It contains system-wide functions and aliases including other configurations that apply to all system users.

But in `.bashrc` its up to the one user only