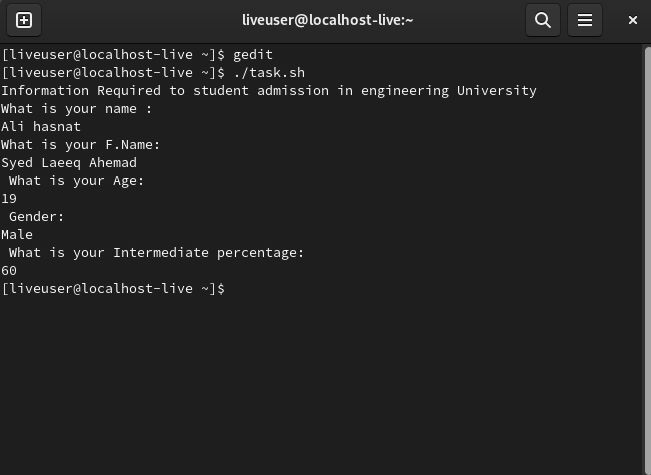


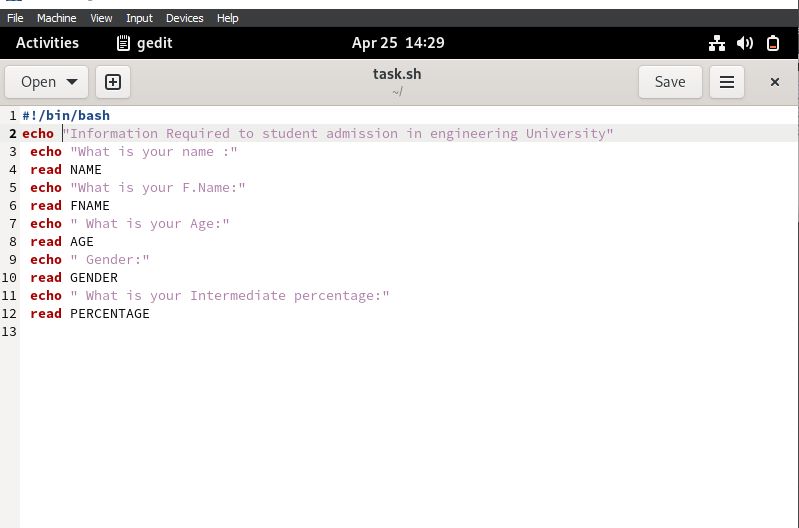
**Name:** **Syed Ali Hasnat Danyal**

**Id: 59750**

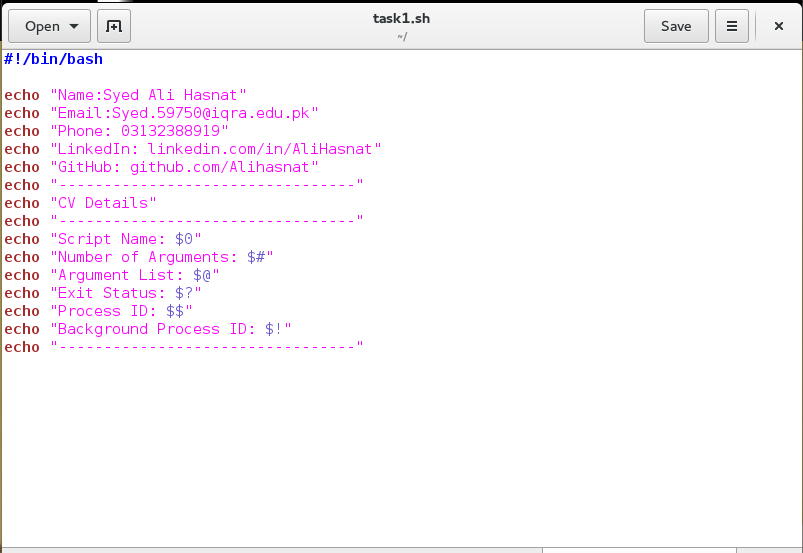
**Course: Operating System (Lab)**

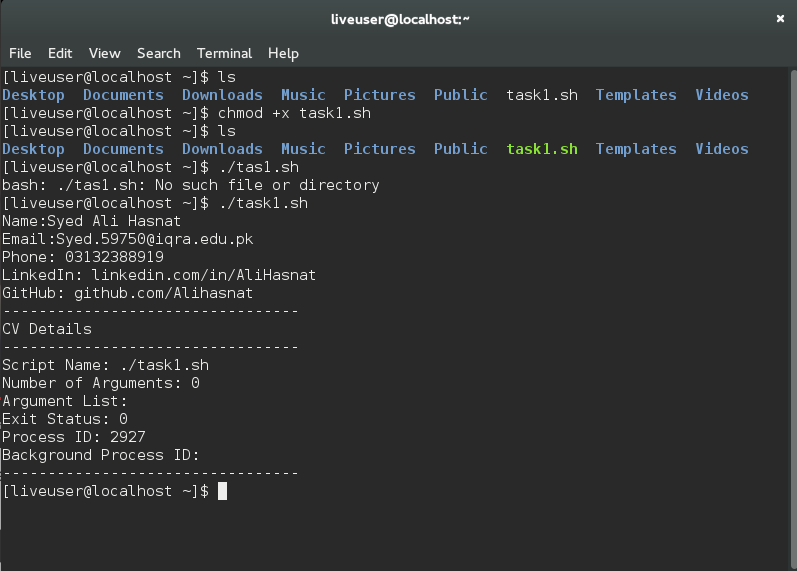
Task 1



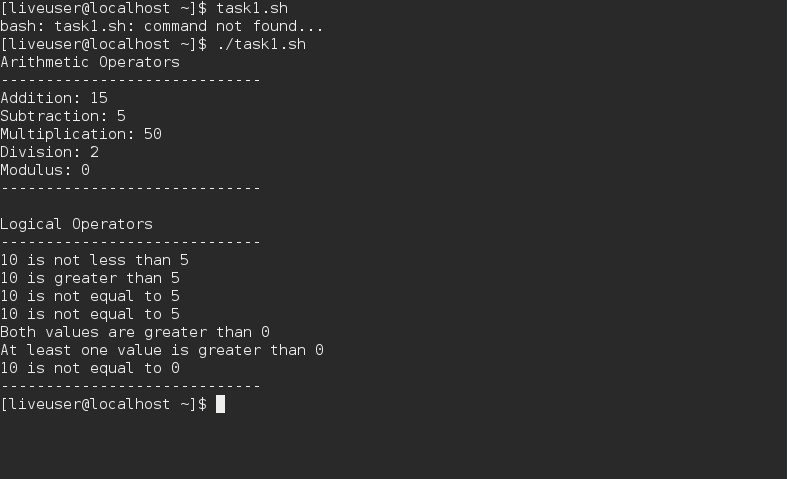


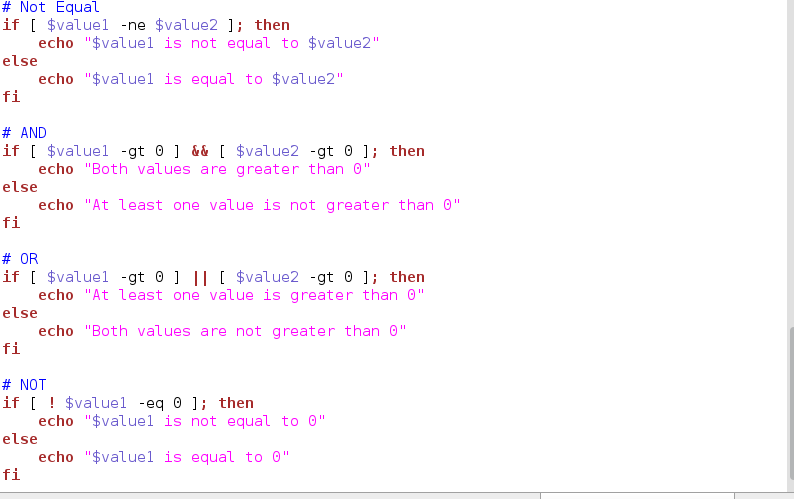
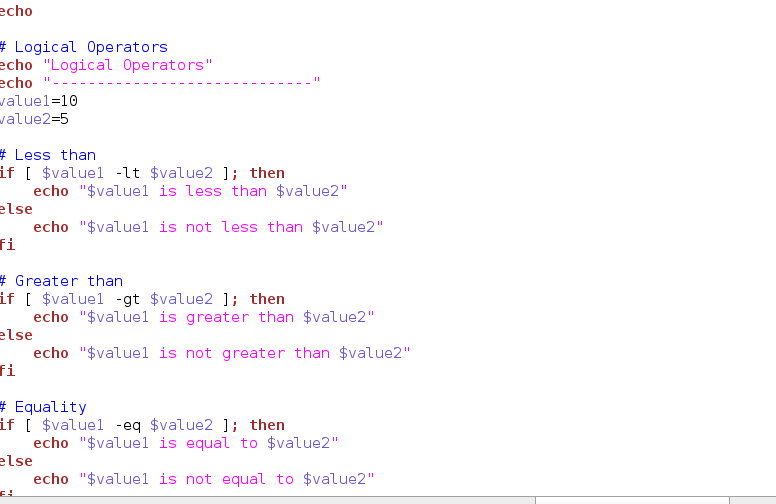
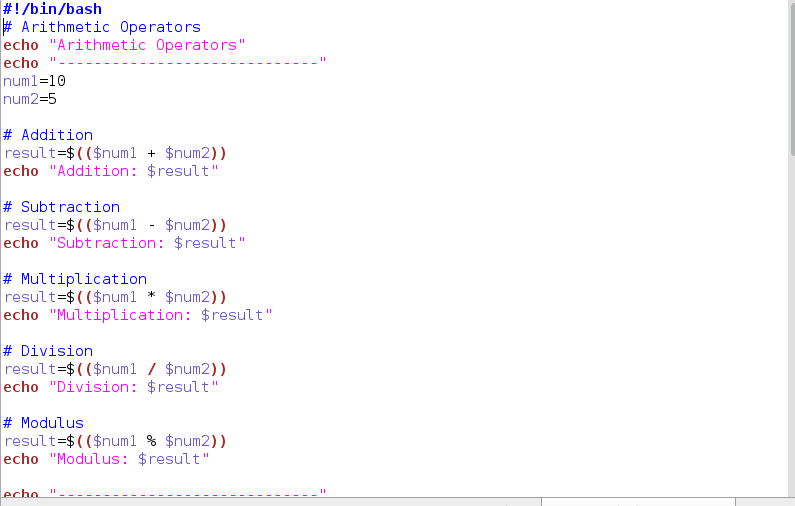
Task 2



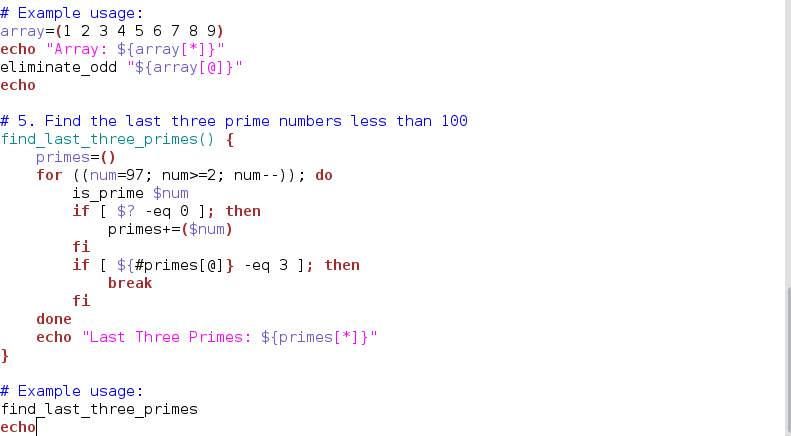
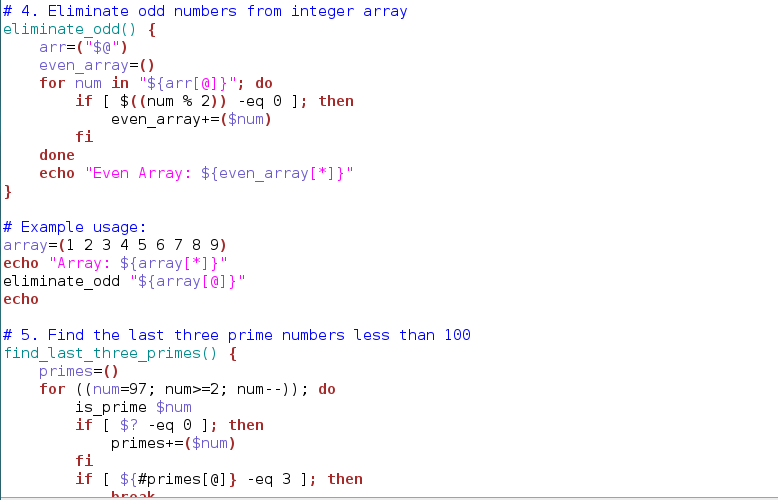
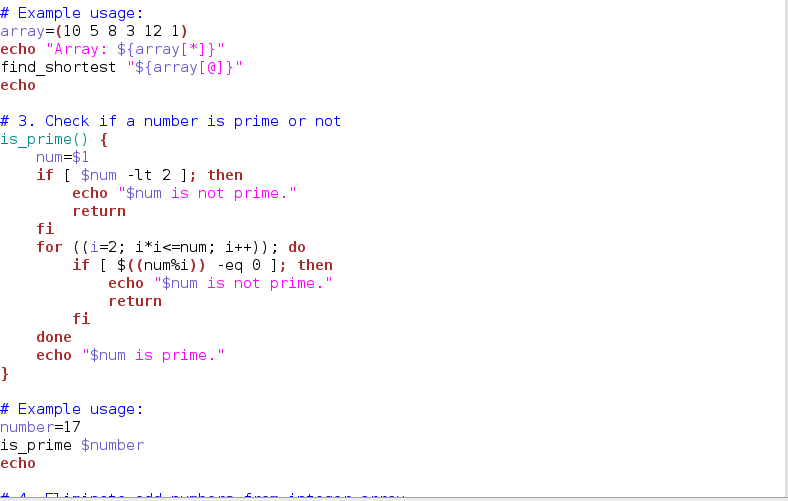
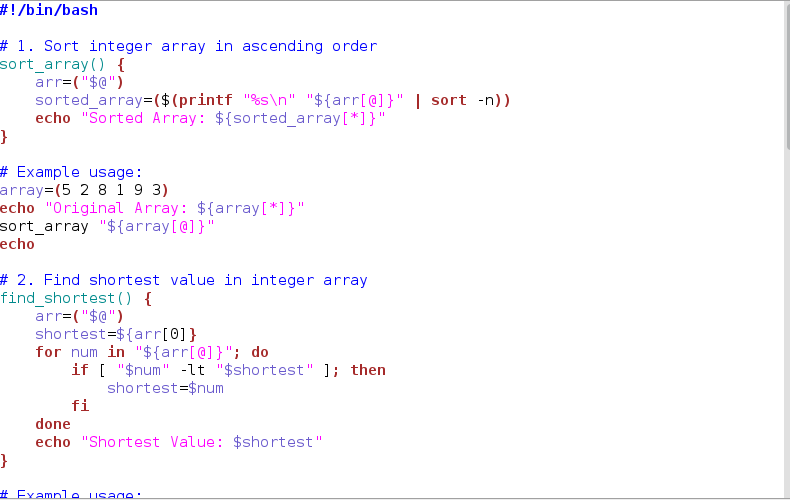
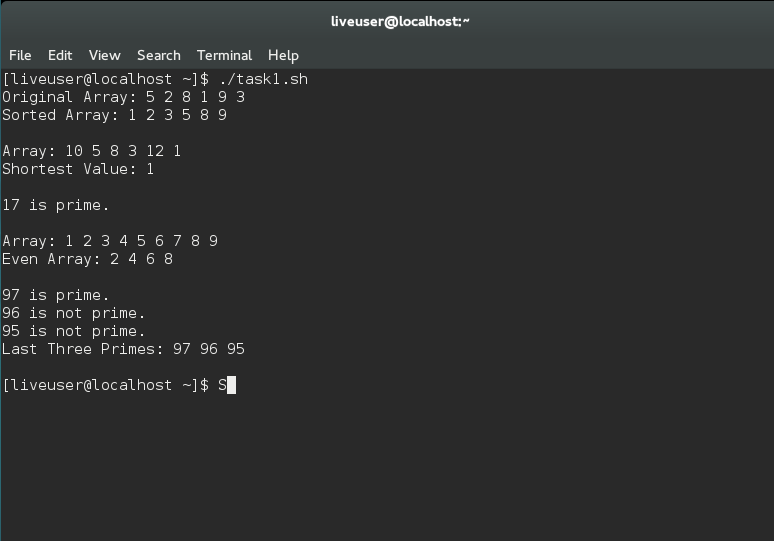


Task3

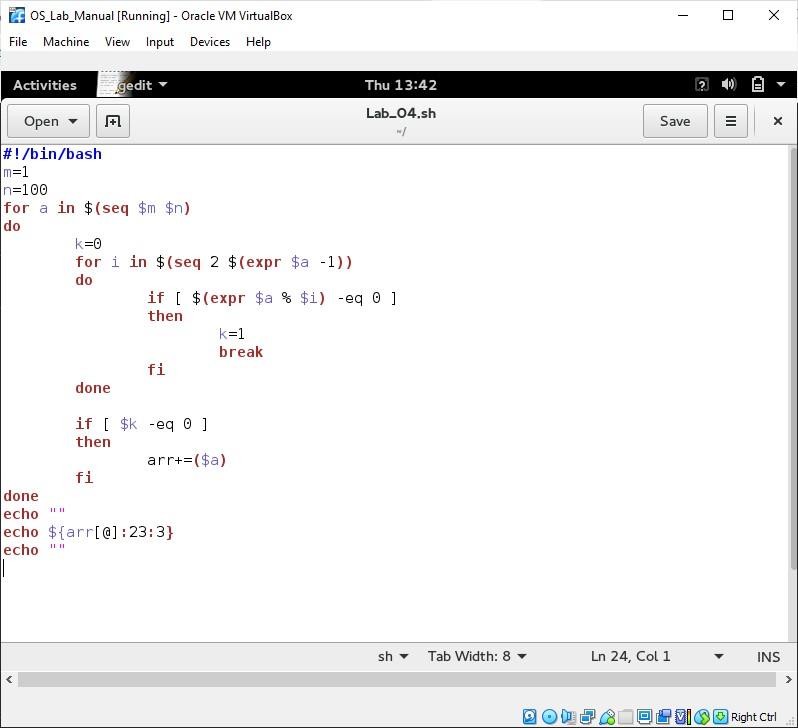




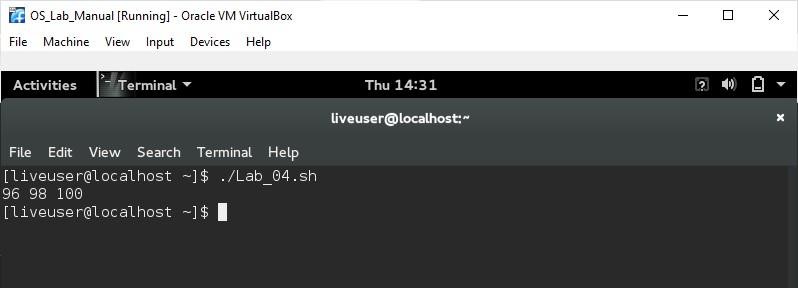
Task4



**5.Make shell function which can find the last three prime number lesser than 100. Code Screen Shot:**



**Code Output Screen Shot:**



# Task #5

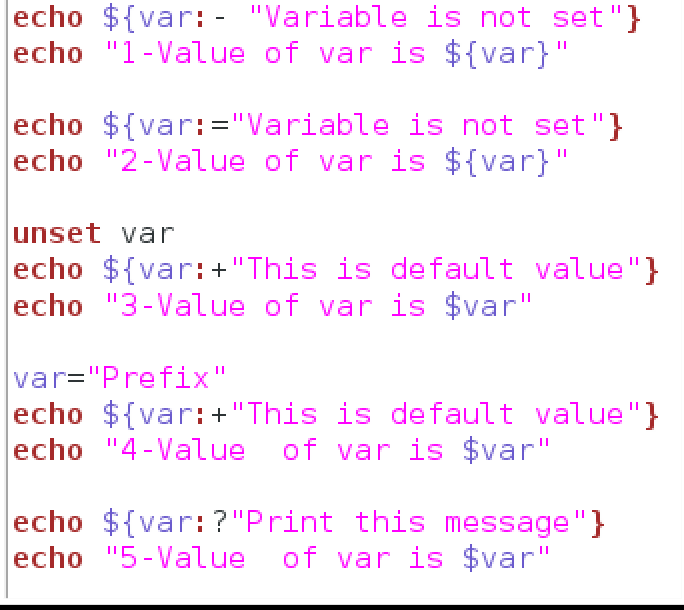
Shell Substitution

|  |  |
| --- | --- |
| **INPUT** | **OUTPUT** |
|  |  |

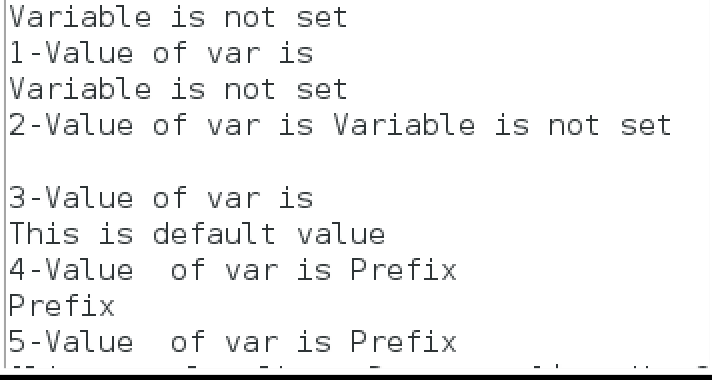
# Variable Substitution

|  |  |
| --- | --- |
| **Form** | **Description** |
| **${var}** | Substitue the value of *var*. |
| **${var:-word}** | If *var* is null or unset, *word* is substituted for **var**. The value of *var*  does not change. |
| **${var:=word}** | If *var* is null or unset, *var* is set to the value of **word**. |
| **${var:?message}** | If *var* is null or unset, *message* is printed to standard error. This checks that variables are set correctly. |
| **${var:+word}** | If *var* is set, *word* is substituted for var. The value of *var*does not change. |

**INPUT**



**OUTPUT**

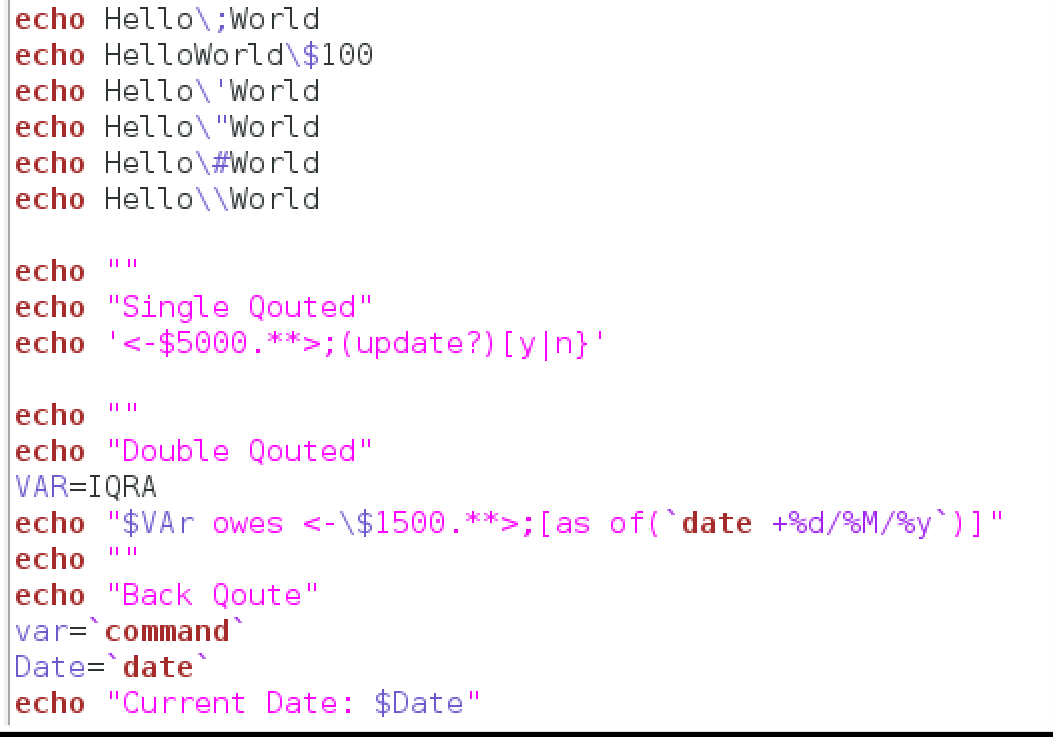


**Quoting mechanisms**

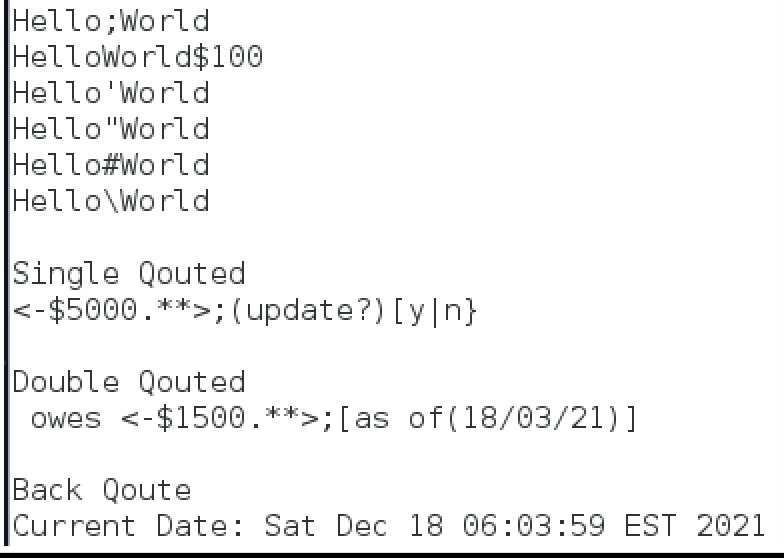
|  |  |
| --- | --- |
| **Quoting** | **Description** |
| **Single quote** | All special characters between these quotes lose their special meaning. |

|  |  |
| --- | --- |
| **Double quote** | Most special characters between these quotes lose their special meaning with these exceptions:   * $ * ` * \$ * \' * \" * \\ |
| **Backslash** | Any character immediately following the backslash loses its special meaning. |
| **Back Quote** | Anything in between back quotes would be treated as a command and would be executed. |

**INPUT**

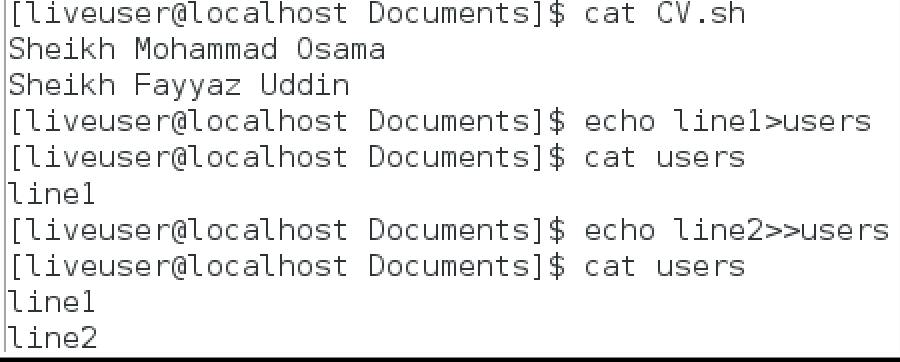


**OUTPUT**

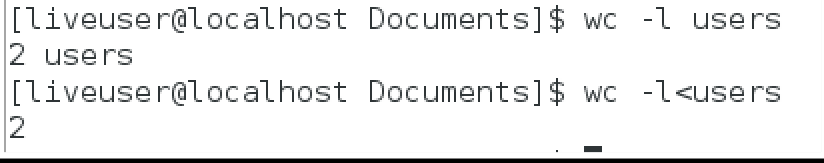


# I/O Redirection

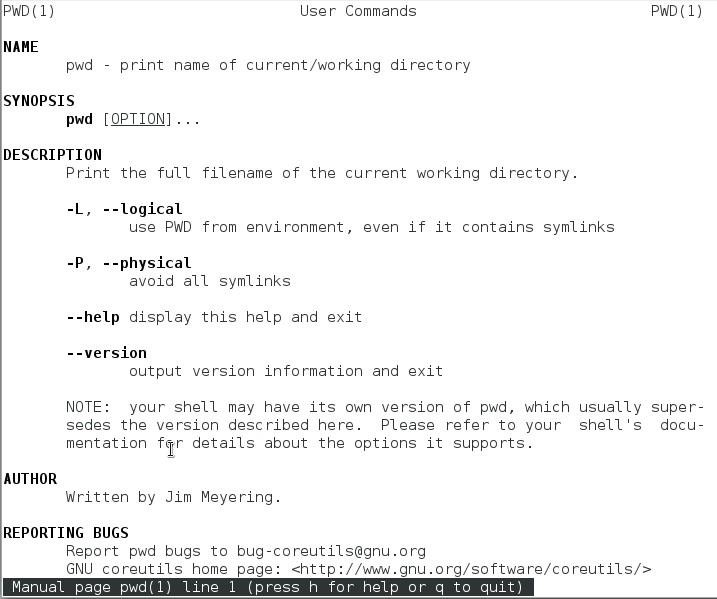
Output Redirection

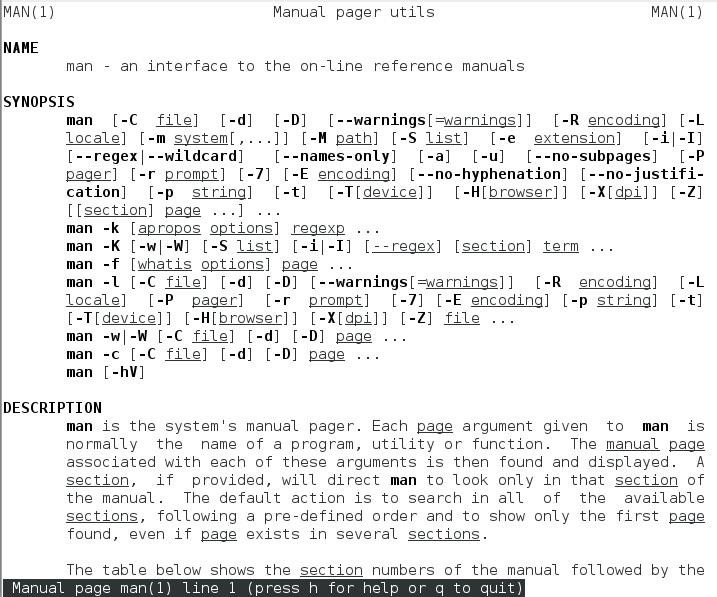


Input Redirection

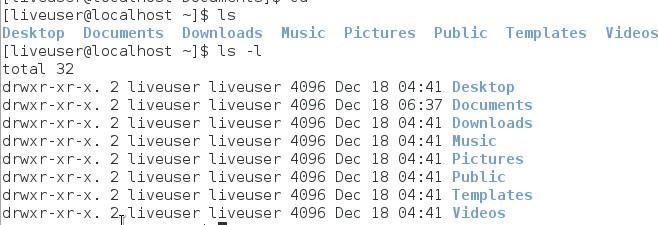


# Man Page Section

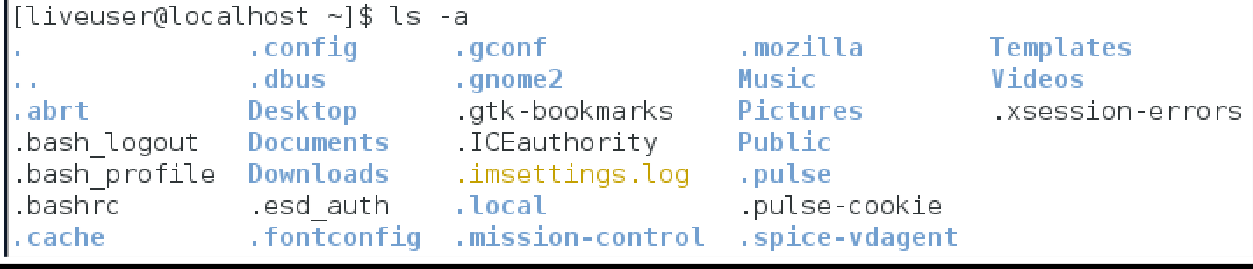




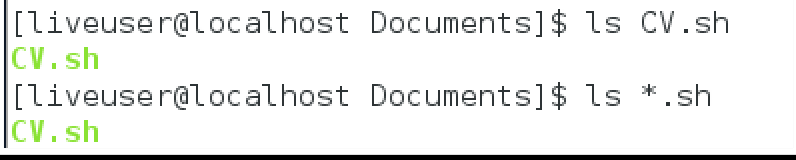
**Task #6**

Listing Files

# Hidden Files



**Meta Characters**

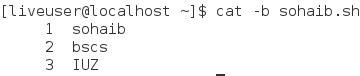
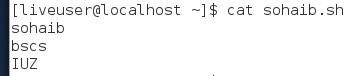


**Creating Files**





# Display Content of a File



**Counting Words in a File**



# Copying Files



**Renaming Files**

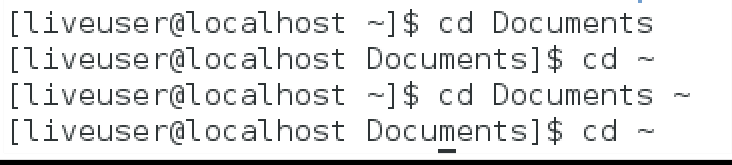


# Deleting Files



**Task #7**

Home Directory



# Absolute/Relative Pathnames

**Listing Directories**



# Creating Directories



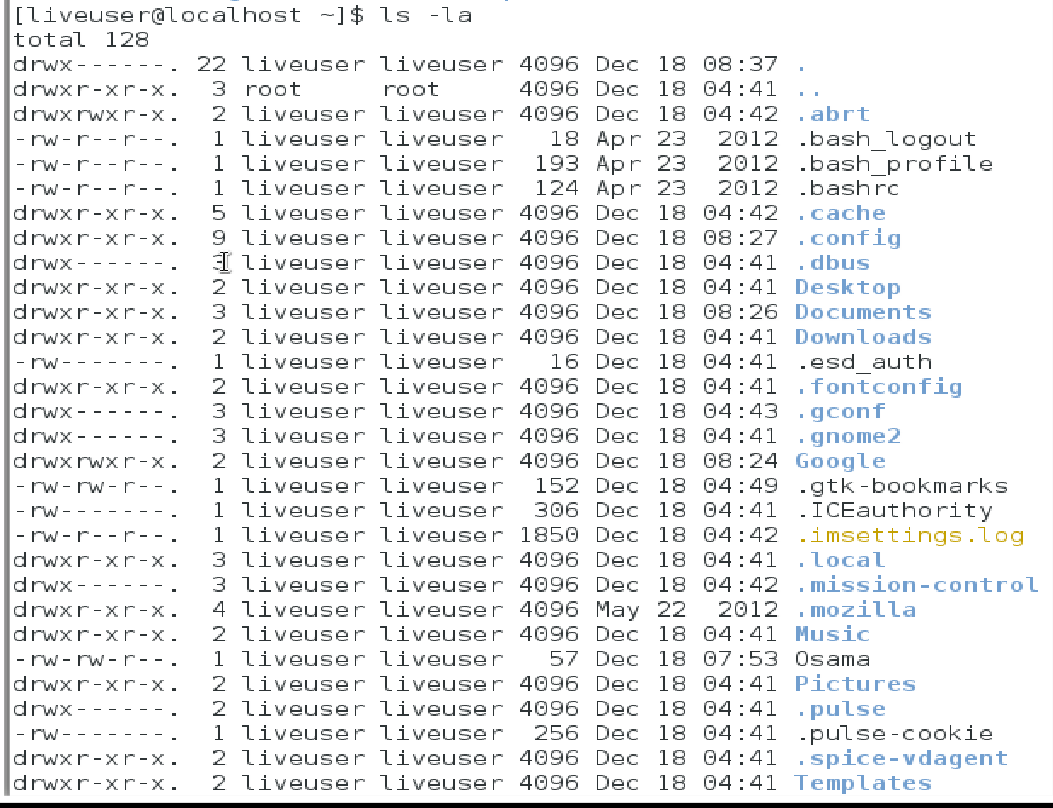
**Creating Parent Directories**



# Changing Directories

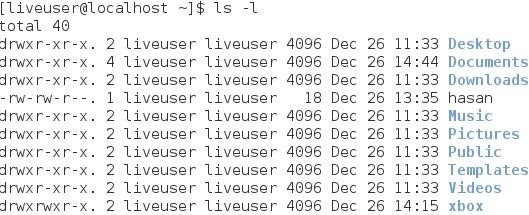


**The directories . (dot) and .. (dot dot)**



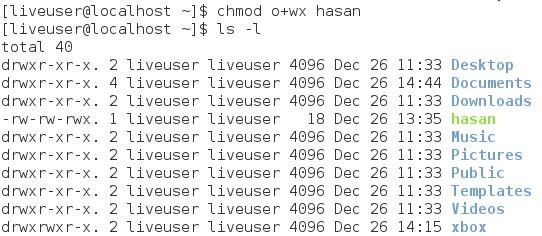
# Task #8

**The Permission Indicators**

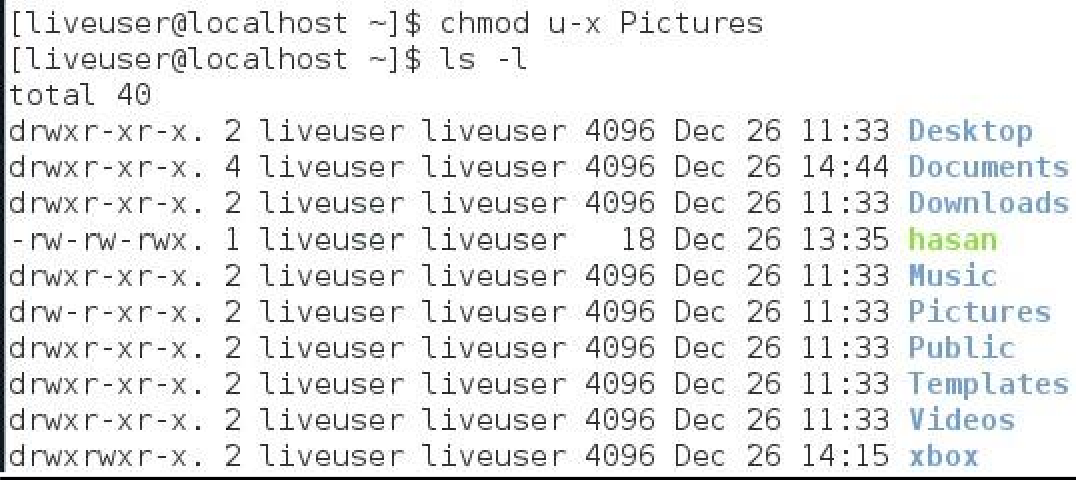


# Changing Permissions

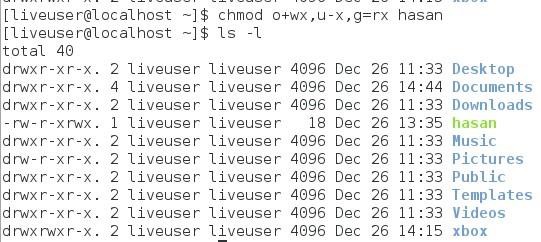
**Chmod o+wx filename**



**Chmod u-x**



**Combine Chmod o+wx u-x**



# Example

**Changing Ownership**



# Task #9 a

**DESCRIPTION**

Assume all the processes arrive at the same time.

**FCFS CPU SCHEDULING ALGORITHM**

For FCFS scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. The scheduling is performed on the basis of arrival time of the processes irrespective of their other parameters. Each process will be executed according to its arrival time. Calculate the waiting time and turnaround time of each of the processes accordingly.

**SJF CPU SCHEDULING ALGORITHM**

For SJF scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. Arrange all the jobs in order with respect to their burst times. There may be two jobs in queue with the same execution time, and then FCFS approach is to be performed. Each process will be executed according to the length of its burst time. Then calculate the waiting time and turnaround time of each of the processes accordingly.

**ROUND ROBIN CPU SCHEDULING ALGORITHM**

For round robin scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the size of the time slice. Time slices are assigned to each process in equal portions and in circular order, handling all processes execution. This allows every process to get an equal chance. Calculate the waiting time and turnaround time of each of the processes accordingly.

**PRIORITY CPU SCHEDULING ALGORITHM**

For priority scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the priorities. Arrange all the jobs in order with respect to their priorities. There may be two jobs in queue with the same priority, and then FCFS approach is to be performed. Each process will be executed according to its priority. Calculate the waiting time and turnaround time of each of the processes accordingly.

# FCFS CPU SCHEDULING ALGORITHM

**Code:**

#include<stdio.h> #include<conio.h> int main() {

int bt[20], wt[20], tat[20], i, n; float wtavg, tatavg;

printf("\nEnter the number of processes -- "); scanf\_s("%d", &n); for (i = 0;i < n;i++)

{

printf("\nEnter Burst Time for Process %d -- ", i); scanf\_s("%d", &bt[i]);

}

wt[0] = wtavg = 0; tat[0] = tatavg = bt[0]; for (i = 1;i < n;i++)

{

wt[i] = wt[i - 1] + bt[i - 1];tat[i] = tat[i - 1] + bt[i]; wtavg = wtavg + wt[i]; tatavg = tatavg + tat[i];

}

printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");

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

{

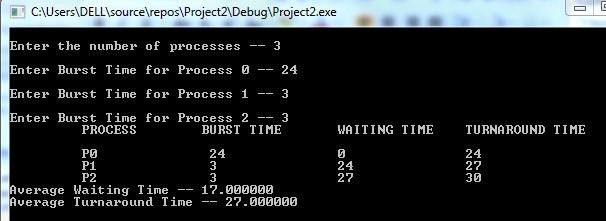
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);

}

printf("\nAverage Waiting Time -- %f", wtavg / n); printf("\nAverage Turnaround Time -- %f", tatavg / n); \_getch();

}

OUTPUT



# SJF CPU SCHEDULING ALGORITHM

**Code:**

#include<stdio.h> #include<conio.h> int main()

{int p[20], bt[20], wt[20], tat[20], i, k, n, temp; float wtavg, tatavg; printf("\nEnter the number of processes -- "); scanf\_s("%d", &n); for (i = 0;i

< n;i++)

{p[i] = i;

printf("Enter Burst Time for Process %d -- ", i); scanf\_s("%d", &bt[i]);

}

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

for (k = i + 1;k < n;k++) if (bt[i] > bt[k])

{

temp = bt[i]; bt[i] = bt[k]; bt[k] = temp; temp = p[i]; p[i] = p[k];

p[k] = temp; }

wt[0] = wtavg = 0; tat[0] = tatavg = bt[0];

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

wtavg = wtavg + wt[i]; tatavg = tatavg + tat[i]; }

printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n"); for (i = 0;i

< n;i++)

printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]); printf("\nAverage Waiting Time -- %f", wtavg / n); printf("\nAverage Turnaround Time -- %f", tatavg / n);

\_getch(); }

# Output:

**Round Robin CPU SCHEDULING ALGORITHM**

# Code:

#include<stdio.h> #include<conio.h> int main()

{int i, j, n, bu[10], wa[10], tat[10], t, ct[10], max; float awt = 0, att = 0, temp = 0; printf("Enter the no of processes -- "); scanf\_s("%d", &n); for (i = 0;i < n;i++)

{printf("\nEnter Burst Time for process %d -- ", i + 1); scanf\_s("%d", &bu[i]);ct[i] = bu[i];} printf("\nEnter the size of time slice -- "); scanf\_s("%d", &t);

max = bu[0];

for (i = 1;i < n;i++) if (max < bu[i]) max = bu[i];

for (j = 0;j < (max / t) + 1;j++) for (i = 0;i < n;i++)

if (bu[i] != 0)

if (bu[i] <= t){ tat[i] = temp + bu[i]; temp = temp + bu[i];

bu[i] = 0;

}else

{bu[i] = bu[i] - t;temp = temp + t;} for (i = 0;i < n;i++){

wa[i] = tat[i] - ct[i];att += tat[i]; awt += wa[i];}

printf("\nThe Average Turnaround time is -- %f", att / n); printf("\nThe Average Waiting time is -- %f ", awt / n);

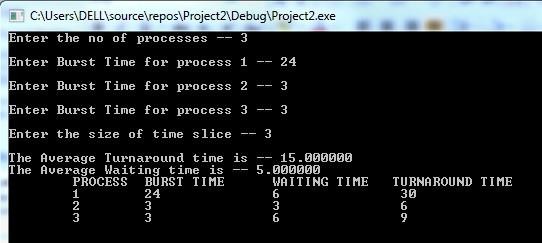
printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n"); for (i = 0;i

< n;i++)

printf("\t%d \t %d \t\t %d \t\t %d \n", i + 1, ct[i], wa[i], tat[i]);

\_getch();}

# Output:



**Priority CPU SCHEDULING ALGORITHM Code:**

#include<stdio.h> #include<conio.h> int main()

{int p[20], bt[20], pri[20], wt[20], tat[20], i, k, n, temp; float wtavg, tatavg; printf("Enter the number of processes --- "); scanf\_s("%d", &n);

for (i = 0;i < n;i++) { p[i] = i;

printf("Enter the Burst Time & Priority of Process %d --- ", i); scanf\_s("%d %d", &bt[i], &pri[i]); }

for (i = 0;i < n;i++) for (k = i + 1;k < n;k++) if (pri[i] > pri[k]){ temp = p[i];

p[i] = p[k]; p[k] = temp; temp = bt[i]; bt[i] = bt[k]; bt[k] = temp; temp = pri[i]; pri[i] = pri[k]; pri[k] = temp;

}wtavg = wt[0] = 0; tatavg = tat[0] = bt[0]; for (i = 1;i < n;i++){

wt[i] = wt[i - 1] + bt[i - 1];tat[i] = tat[i - 1] + bt[i]; wtavg = wtavg + wt[i];

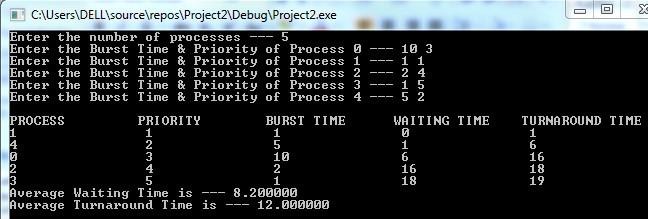
tatavg = tatavg + tat[i];}

printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND TIME"); for (i = 0;i < n;i++)

printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ", p[i], pri[i], bt[i], wt[i], tat[i]); printf("\nAverage Waiting Time is --- %f", wtavg / n); printf("\nAverage Turnaround Time is --- %f", tatavg / n);

\_getch();}

# Output:



**Task #9 b**

### DESCRIPTION

Multi-level queue scheduling algorithm is used in scenarios where the processes can be classified into groups based on property like process type, CPU time, IO access, memory size, etc. In a multi-level queue scheduling algorithm, there will be 'n' number of queues, where 'n' is the number of groups the processes are classified into. Each queue will be assigned a priority and will have its own scheduling algorithm like round-robin scheduling or FCFS. For the process in a queue to execute, all the queues of priority higher than it should be empty, meaning the process in those high priority queues should have completed its execution. In this scheduling algorithm, once assigned to a queue, the process will not move to any other queues.

# Code:

#include<stdio.h> #include<conio.h> int main() {

int p[20], bt[20], su[20], wt[20], tat[20], i, k, n, temp; float wtavg, tatavg; printf("Enter the number of processes --- "); scanf\_s("%d", &n);

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

{ p[i] = i;

printf("Enter the Burst Time of Process %d --- ", i); scanf\_s("%d", &bt[i]); printf("System/User Process (0/1) ? --- "); scanf\_s("%d", &su[i]);

}

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

for (k = i + 1;k < n;k++) if (su[i] > su[k]) {

temp = p[i]; p[i] = p[k];

p[k] = temp; temp = bt[i]; bt[i] = bt[k]; bt[k] = temp; temp = su[i]; su[i] = su[k];

su[k] = temp; }

wtavg = wt[0] = 0; tatavg = tat[0] = bt[0]; for (i = 1;i < n;i++)

{

wt[i] = wt[i - 1] + bt[i - 1];tat[i] = tat[i - 1] + bt[i]; wtavg = wtavg + wt[i];

tatavg = tatavg + tat[i];

}

printf("\nPROCESS\t\t SYSTEM/USER PROCESS \tBURST TIME\tWAITING TIME\tTURNAROUND

TIME"); for (i = 0;i < n;i++)

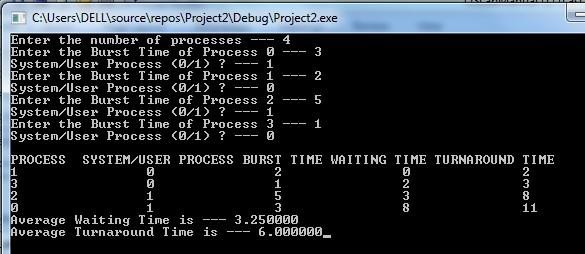
printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ", p[i], su[i], bt[i], wt[i], tat[i]); printf("\nAverage Waiting Time is --- %f", wtavg / n);

printf("\nAverage Turnaround Time is --- %f", tatavg / n);

\_getch();

}

# Output:



**Task #10**

#### DESCRIPTION

A file is a collection of data, usually stored on disk. As a logical entity, a file enables to divide data into meaningful groups. As a physical entity, a file should be considered in terms of its organization. The term "file organization" refers to the way in which data is stored in a file and, consequently, the method(s) by which it can be accessed.

#### SEQUENTIAL FILE ALLOCATION

In this file organization, the records of the file are stored one after another both physically and logically. That is, record with sequence number 16 is located just after the 15th record. A record of a sequential file can only be accessed by reading all the previous records.

#### LINKED FILE ALLOCATION

With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. Each block contains a pointer to the next block.

#### INDEXED FILE ALLOCATION

Indexed file allocation strategy brings all the pointers together into one location: an index block. Each file has its own index block, which is an array of disk-block addresses. The ith entry in the index block points to the ith block of the file. The directory contains the address of the index block. To find and read the ith block, the pointer in the ith index-block entry is used.

# SEQUENTIAL FILE ALLOCATION

**Code:**

#include<stdio.h> #include<conio.h> struct fileTable

{

char name[20]; int sb, nob;

}ft[30]; void main()

{ int i, j, n; char s[20]; printf("Enter no of files :"); scanf\_s("%d", &n);

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

{ printf("\nEnter file name %d ", i + 1); scanf\_s("%s", ft[i].name);

printf("Enter starting block of file %d :", i + 1); scanf\_s("%d", &ft[i].sb);

printf("Enter no of blocks in file %d :", i + 1); scanf\_s("%d", &ft[i].nob);

}

printf("\nEnter the file name to be searched-- "); scanf("%s", s);

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

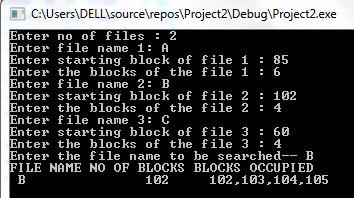
if (strcmp(s, ft[i].name) == 0) break; if (i == n) printf("\nFile Not Found"); else {

printf("\nFILE NAME START BLOCK NO OF BLOCKS BLOCKS OCCUPIED\n");

printf("\n%s\t\t%d\t\t%d\t", ft[i].name, ft[i].sb, ft[i].nob); for (j = 0;j < ft[i].nob;j++) printf("%d, ", ft[i].sb + j);}

\_getch();}

# Output:



**LINKED LIST FILE ALLOCATION**

# Code:

#include<stdio.h> #include<conio.h>

struct fileTable

{

char name[20]; int nob;

struct block \*sb;

}ft[30]; struct block

{

int bno;

struct block \*next;

}; void

main()

{

int i, j, n; char s[20]; struct block \*temp; printf("Enter no of files

:"); scanf\_s("%d", &n); for (i = 0;i < n;i++) {

printf("\nEnter file name %d :", i

+ 1);

scanf\_s("%s",

ft[i].name);

printf("Enter no of blocks in file %d :", i + 1); scanf\_s("%d", &ft[i].nob);

ft[i].sb = (struct block\*)malloc(sizeof(struct block)); temp = ft[i].sb;

printf("Enter the blocks of the file :"); scanf\_s("%d", &temp->bno);temp->next = NULL;

for (j = 1;j < ft[i].nob;j++) {

temp->next = (struct block\*)malloc(sizeof(struct block)); temp = temp-

>next;

scanf\_s("%d", &temp->bno);

}

temp->next = NULL;

}

printf("\nEnter the file name to be searched -- "); scanf\_s("%s", s);

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

if (strcmp(s, ft[i].name) == 0) break; if (i == n)

printf("\nFile Not Found"); else

{

printf("\nFILE NAME NO OF BLOCKS BLOCKS OCCUPIED"); printf("\n %s\t\t%d\t", ft[i].name, ft[i].nob); temp

= ft[i].sb;

for (j = 0;j < ft[i].nob;j++)

{

printf("%d ", temp->bno);temp = temp->next;



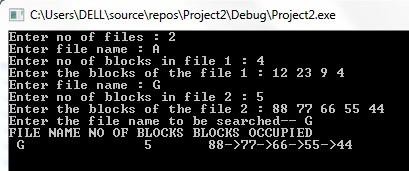
}

}

\_getch();

}

# Output:



**INDEXED FILE ALLOCATION**

# Code:

#include<stdio.h> #include<conio.h> struct fileTable

{

char name[20]; int nob, blocks[30]; }ft[30]; void main() {

int i, j, n; char s[20];

printf("Enter no of files :"); scanf\_s("%d", &n); for (i = 0;i < n;i++) { printf("\nEnter file name %d :", i + 1); scanf\_s("%s", ft[i].name);

printf("\nEnter no of blocks in file %d :", i + 1); scanf\_s("%d", &ft[i].nob); printf("\nEnter the blocks of the file :"); for (j = 0;j < ft[i].nob;j++) scanf\_s("%d", &ft[i].blocks[j]); }

printf("\nEnter the file name to be searched-- "); scanf\_s("%s", s);

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

if (strcmp(s, ft[i].name) == 0) if (i == n)

printf("\nFile Not Found");

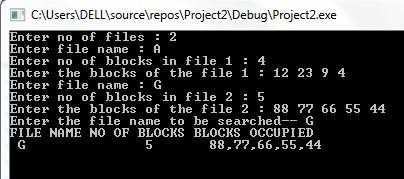
else { printf("\nFILE NAME NO OF BLOCKS BLOCKS OCCUPIED");

printf("\n %s\t\t%d\t", ft[i].name, ft[i].nob); for (j = 0;j < ft[i].nob;j++)

printf("%d, ", ft[i].blocks[j]); }

\_getch(); }

# Output:



**Task #11**

### DESCRIPTION

MFT (Multiprogramming with a Fixed number of Tasks) is one of the old memory management techniques in which the memory is partitioned into fixed size partitions and each job is assigned to a partition. The memory assigned to a partition does not change. MVT (Multiprogramming with a Variable number of Tasks) is the memory management technique in which each job gets just the amount of memory it needs. That is, the partitioning of memory is dynamic and changes as jobs enter and leave the system. MVT is a more ``efficient'' user of resources. MFT suffers with the problem of internal fragmentation and MVT suffers with external fragmentation.

**PROGRAM**

# MFT MEMORY MANAGEMENT TECHNIQUE

**Code:**

#include<stdio.h> #include<conio.h> int main() {

int ms, bs, nob, ef, n, mp[10], tif = 0; int i, p = 0;

printf("Enter the total memory available (in Bytes) -- "); scanf\_s("%d", &ms); printf("Enter the block size (in Bytes) -- "); scanf\_s("%d", &bs); nob = ms / bs; ef = ms

- nob \* bs;

printf("\nEnter the number of processes -- "); scanf\_s("%d", &n); for (i = 0;i < n;i++)

{

printf("Enter memory required for process %d (in Bytes)-- ", i + 1); scanf\_s("%d", &mp[i]);

}

printf("\nNo. of Blocks available in memory -- %d", nob);

printf("\n\nPROCESS\tMEMORY REQUIRED\t ALLOCATED\tINTERNAL FRAGMENTATION");

for (i = 0;i < n && p < nob;i++) {

printf("\n %d\t\t%d", i + 1, mp[i]); if (mp[i] > bs) printf("\t\tNO\t\t---");

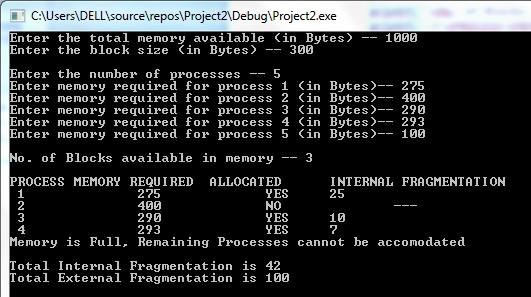
else { printf("\t\tYES\t%d", bs - mp[i]);tif = tif + bs - mp[i]; p++; } }

if (i < n) printf("\nMemory is Full, Remaining Processes cannot be accomodated");

printf("\n\nTotal Internal Fragmentation is %d", tif); printf("\nTotal External Fragmentation is %d", ef); \_getch();

}

# Output:



**MVT MEMORY MANAGEMENT TECHNIQUE**

**Code:**

#include<stdio.h> #include<conio.h> int main() {

int ms, mp[10], i, temp, n = 0; char ch = 'y'; printf("\nEnter the total memory available (in Bytes)-- "); scanf\_s("%d", &ms); temp = ms;

for (i = 0;ch == 'y';i++, n++) {

printf("\nEnter memory required for process %d (in Bytes) -- ", i + 1); scanf\_s("%d", &mp[i]); if (mp[i] <= temp) {

printf("\nMemory is allocated for Process %d ", i + 1); temp = temp - mp[i]; } else { printf("\nMemory is Full");

break; }

printf("\nDo you want to continue(y/n) -- "); scanf\_s(" %c", &ch); }

printf("\n\nTotal Memory Available -- %d", ms);

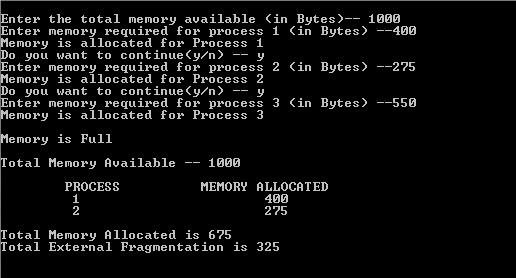
printf("\n\n\tPROCESS\t\t MEMORY ALLOCATED "); for (i = 0;i < n;i++) printf("\n \t%d\t\t%d", i + 1, mp[i]);

printf("\n\nTotal Memory Allocated is %d", ms - temp); printf("\nTotal External Fragmentation is %d", temp);

\_getch();

}

# Output:



**Task #12**

### DESCRIPTION

One of the simplest methods for memory allocation is to divide memory into several fixedsized partitions. Each partition may contain exactly one process. In this multiplepartition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate. Best-fit strategy chooses the block that is closest in size to the request. First-fit chooses the first available block that is large enough. Worst-fit chooses the largest available block.

**PROGRAM**

# FIRST-FIT Code:

#include<stdio.h> #include<conio.h>

#define max 25 void main()

{ int frag[max], b[max], f[max], i, j, nb, nf, temp; static int bf[max], ff[max];

printf("\n\tMemory Management Scheme - First Fit"); printf("\nEnter the number of blocks:"); scanf\_s("%d", &nb); printf("Enter the number of files:"); scanf\_s("%d", &nf);

printf("\nEnter the size of the blocks:-\n");for (i = 1;i <= nb;i++)

{ printf("Block %d:", i); scanf\_s("%d", &b[i]);

}

printf("Enter the size of the files :-\n");for (i = 1;i <= nf;i++)

{ printf("File %d:", i); scanf\_s("%d", &f[i]); } for (i = 1;i <= nf;i++)

{ for (j = 1;j <= nb;j++)

{ if (bf[j] != 1)

{ temp = b[j] - f[i];if (temp >= 0)

{ ff[i] = j;

break; } } } frag[i] = temp;

bf[ff[i]] = 1;

}

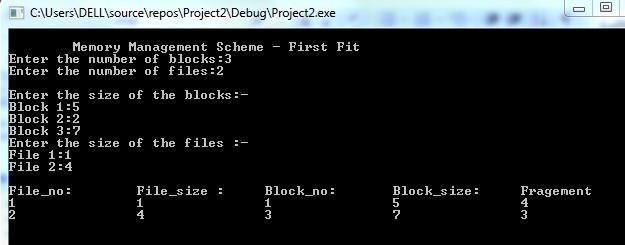
printf("\nFile\_no:\tFile\_size :\tBlock\_no:\tBlock\_size:\tFragement"); for (i = 1;i <= nf;i++)

printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);

\_getch();

}

# Output:



**BEST-FIT Code:**

#include<stdio.h> #include<conio.h> #define max 25 void main()

{ int frag[max], b[max], f[max], i, j, nb, nf, temp, lowest = 10000; static int bf[max], ff[max];

printf("\n\tMemory Management Scheme - Best Fit"); printf("\nEnter the number of blocks:");

scanf\_s("%d", &nb);

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

printf("\nEnter the size of the blocks:-\n")

;for (i = 1;i <= nb;i++)

{

printf("Block %d:", i); scanf\_s("%d", &b[i]); }

printf("Enter the size of the files :-\n");for (i = 1;i <= nf;i++) { printf("File %d:", i); scanf\_s("%d", &f[i]); }

for (i = 1;i <= nf;i++)

{ for (j = 1;j <= nb;j++)

{ if (bf[j] != 1) {

temp = b[j] - f[i];if (temp >= 0) if (lowest > temp) { ff[i] = j; lowest = temp; } } }

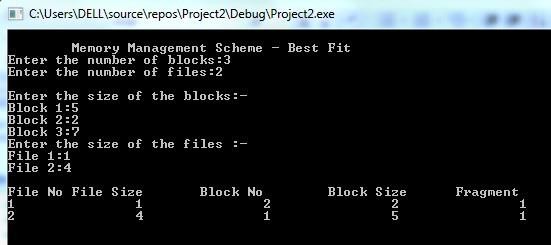
frag[i] = lowest; bf[ff[i]] = 1; lowest = 10000; }

printf("\nFile No\tFile Size \tBlock No\tBlock Size\tFragment"); for (i = 1;i <= nf && ff[i] != 0;i++) printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);

\_getch();

}

# Output:



**WORST FIT**

# Code:

#include<stdio.h> #include<conio.h> #define max 25 void main()

{ int frag[max], b[max], f[max], i, j, nb, nf, temp, highest = 0; static int bf[max], ff[max];

printf("\n\tMemory Management Scheme - Worst Fit"); printf("\nEnter the number of blocks:"); scanf\_s("%d", &nb); printf("Enter the number of files:"); scanf\_s("%d", &nf); printf("\nEnter the size of the blocks:-\n");for (i = 1;i <= nb;i++) { printf("Block %d:", i); scanf\_s("%d", &b[i]); }

printf("Enter the size of the files :-\n");for (i = 1;i <= nf;i++)

{ printf("File %d:", i); scanf\_s("%d", &f[i]);

}

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

for (j = 1;j <= nb;j++) { if (bf[j] != 1) //if bf[j] is not allocated

{ temp = b[j] - f[i];if (temp >= 0) if (highest < temp)

{ ff[i] = j;

highest = temp} } }

frag[i] = highest; bf[ff[i]] = 1; highest = 0; }

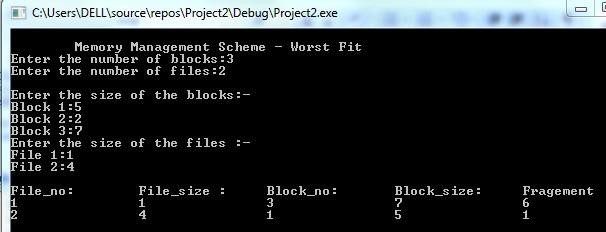
printf("\nFile\_no:\tFile\_size :\tBlock\_no:\tBlock\_size:\tFragement"); for (i = 1;i <= nf;i++)

printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, f[i], ff[i], b[ff[i]], frag[i]);

\_getch();

}

# Output:



**Task #13**

**Objective:**

Write a C program to simulate the pagging memory management techniques.

### 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-managementscheme, 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.

**PROGRAM**

#include<stdio.h> #include<conio.h> int 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\_s("%d", &ms);

printf("\nEnter the page size -- "); scanf\_s("%d", &ps);

nop = ms / ps;

printf("\nThe no. of pages available in memory are -- %d ", nop); printf("\nEnter number of processes -- "); scanf\_s("%d", &np); rempages = nop;

for (i = 1;i <= np;i++) { printf("\nEnter no. of pages required for p[%d]-- ", i); scanf\_s("%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\_s("%d", &fno[i][j]);

}

printf("\nEnter Logical Address to find Physical Address "); printf("\nEnter process no. and pagenumber and offset -- ");

scanf\_s("%d %d %d", &x, &y, &offset); f (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:

**Task #14**

Deadlock Management Techniques

**Objective:**

## Write a C program to simulate Bankers algorithm for the purpose of deadlock avoidance.

Write a C program to simulate disk scheduling algorithms

## FCFS

* SCAN

## C-SCAN

### 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.

**Code:**

#include<stdio.h> #include<conio.h> struct file {

int all[10]; int max[10]; int need[10]; int flag;

}; void

main()

{

struct file f[10]; int fl;

int i, j, k, p, b, n, r, g, cnt = 0, id, newr; I nt avail[10], seq[10];

printf("Enter number of processes -- "); scanf\_s("%d", &n);

printf("Enter number of resources -- "); scanf\_s("%d", &r); for (i = 0;i < n;i++)

{

printf("Enter details for P%d", i); printf("\nEnter allocation\t -- \t"); for (j = 0;j < r;j++)

scanf\_s("%d", &f[i].all[j]); printf("Enter Max\t\t -- \t"); for (j = 0;j < r;j++) scanf\_s("%d", &f[i].max[j]); f[i].flag = 0;

}

printf("\nEnter Available Resources\t -- \t"); for (i = 0;i < r;i++)

scanf\_s("%d", &avail[i]);

printf("\nEnter New Request Details -- "); printf("\nEnter pid \t -- \t"); scanf\_s("%d", &id); printf("Enter Request for Resources \t -- \t");

for (i = 0;i < r;i++) {

scanf\_s("%d", &newr); f[id].all[i] += newr; avail[i] = avail[i] - newr;

}

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

{

for (j = 0;j < r;j++)

{ f[i].need[j] = f[i].max[j] - f[i].all[j];if (f[i].need[j] < 0) f[i].need[j] = 0; } } cnt = 0;

fl = 0;

while (cnt != n) { g = 0; for (j = 0;j < n;j++) { if (f[j].flag == 0)

{

b = 0;

for (p = 0;p < r;p++) {

if (avail[p] >= f[j].need[p]) b = b + 1;

else

b = b - 1;}

if (b == r)

{

printf("\nP%d is visited", j); seq[fl++] = j; f[j].flag = 1;

for (k = 0;k < r;k+

avail[k] = avail[k] + f[j].all[k]; cnt = cnt + 1;

printf("(");

for (k = 0;k < r;k++) printf("%3d", avail[k]);

printf(")"); g = 1; } }}

if (g == 0)

{printf("\n REQUEST NOT GRANTED -- DEADLOCK OCCURRED"); printf("\n SYSTEM IS IN UNSAFE STATE");

goto y;}}

printf("\nSYSTEM IS IN SAFE STATE");

printf("\nThe Safe Sequence is -- ("); for (i = 0;i < fl;i++) printf("P%d ", seq[i]);

printf(")"); y:printf("\nProcess\t\tAllocation\t\tMax\t\t\tNeed\n"); for (i = 0;i < n;i++)

{printf("P%d\t", i); for (j = 0;j < r;j++)

printf("%6d", f[i].all[j]); for (j = 0;j < r;j++) printf("%6d", f[i].max[j]); for (j = 0;j < r;j++) printf("%6d", f[i].need[j]); printf("\n");

}

\_getch();

}

# Output:

**Task #15**

#### DESCRIPTION

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth. Both the access time and the bandwidth can be improved by managing the order in which disk I/O requests are serviced which is called as disk scheduling. The simplest form of disk scheduling is, of course, the firstcome, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. In the SCAN algorithm, the disk arm starts at one end, and moves towards the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. C-SCAN is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip

**PROGRAM**

**FCFS DISK SCHEDULING ALGORITHM Code:**

#include<stdio.h> #include<conio.h> int main() {

int t[20], n, i, j, tohm[20], tot = 0; float avhm;

printf("enter the no.of tracks"); scanf\_s("%d", &n);

printf("enter the tracks to be traversed"); for (i = 2;i < n + 2;i++) scanf\_s("%d"," &t\*i+" );

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

{

tohm[i] = t[i + 1] - t[i];if (tohm[i] < 0) tohm[i] = tohm[i] \* (-1);

}

for (i = 1;i < n + 1;i++) tot += tohm[i];

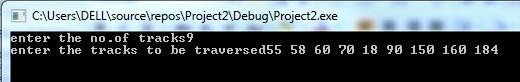
avhm = (float)tot / n;

printf("Tracks traversed\tDifference between tracks\n"); for (i = 1;i < n + 1;i++) printf("%d\t\t\t%d\n" ,"t\*i+" , "tohm\*i+" );

printf("\nAverage header movements:%f", avhm);

\_getch();

}

**Output:**

# SCAN DISK SCHEDULING ALGORITHM

**Code:**

#include<stdio.h> #include<conio.h> int main() {

int t[20], d[20], h, i, j, n, temp, k, atr[20], tot, p, sum = 0; printf("enter the no of tracks to be traveresed");

scanf\_s("%d", &n); printf("enter the position of head"); scanf\_s("%d", &h);

t[0] = 0;t[1] = h; printf("enter the tracks"); f or (i = 2;i < n + 2;i++)

scanf\_s("%d", &t[i]); for (i = 0;i < n + 2;i++)

{

for (j = 0;j < (n + 2) - i - 1;j++)

{ if (t[j] > t[j + 1]) { temp = t[j];

t[j] = t[j + 1];

t[j + 1] = temp; }}}

for (i = 0;i < n + 2;i++)

if (t[i] == h) j = i;k = i; p = 0;

while (t[j] != 0)

{ atr[p] = t[j]; j--; p++;

}

atr[p] = t[j];

for (p = k + 1;p < n + 2;p++, k++) atr[p] = t[k + 1];

for (j = 0;j < n + 1;j++)

{ if (atr[j] > atr[j + 1]) d[j] = atr[j] - atr[j + 1]; else

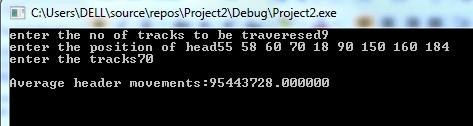
d[j] = atr[j + 1] - atr[j];sum += d[j];

}

printf("\nAverage header movements:%f", (float)sum / n);

\_getch();

}

**Output:**

# C-SCAN DISK SCHEDULING ALGORITHM

**Code:**

#include<stdio.h> #include<conio.h> I nt main() {

int t[20], d[20], h, i, j, n,

temp, k, atr[20], tot, p, sum = 0; printf("enter the track position"); scanf\_s("%d'", &n); printf("enter the starting point"); scanf\_s("%d", &h);

t[0] = 0;t[1] = h;

scanf\_s("%d", &tot); t[2] = tot - 1; for (i = 3;i <= n + 2;i++) scanf\_s("%d", &t[i]);

for (i = 0;i <= n + 2;i++)

for (j = 0;j <= (n + 2) - i - 1;j++)if (t[j] > t[j + 1])

{

temp = t[j]; t[j] = t[j + 1];

t[j + 1] = temp;

}

for (i = 0;i <= n + 2;i++) if (t[i] == h) (j = i);

p = 0; while (t[j] != tot - 1)

{

atr[p] = t[j]; j++; p++;

}

atr[p] = t[j]; p++; i = 0;

while (p != (n + 3) && t[i] != t[h])

{

atr[p] = t[i]; i++;

p++;

}

for (j = 0;j < n + 2;j++)

{

if (atr[j] > atr[j + 1]) d[j] = atr[j] - atr[j + 1]; else

d[j] = atr[j + 1] - atr[j];sum += d[j];

}

printf("total header movements%d", sum); printf("avg is %f", (float)sum / n);

\_getch();

}

# Output:

