Department of Computer Science and Engineering

Lab Manual

for

Operating Systems Laboratory

Regulation 2021

CS3461 – IV Semester

ADHIPARASAKTHI ENGINEERING COLLEGE

Melmaruvathur, Chengalpet District- 603319.

CONTENTS

Expt. No.	Name of Experiment	Page No.
1	INSTALLATION OF WINDOWS OPERATING SYSTEM	3
2A.	ILLUSTRATE UNIX COMMANDS	13
2B.	ILLUSTRATE SHELL PROGRAMMING	15
3A.	PROCESS MANAGEMENT USING SYSTEMS CALLS: FORK,WAIT AND EXIT	17
3B.	PROCESS MANAGEMENT USING SYSTEMS CALLS: GETPID	19
3C.	PROCESS MANAGEMENT USING SYSTEMS CALLS: CLOSE	20
4A.	IMPLEMENTATION OF CPU SCHEDULING ALGORITHM – ROUND ROBIN	23
4B.	IMPLEMENTATION OF CPU SCHEDULING ALGORITHM – SHORTEST JOB FIRST	28
4C.	IMPLEMENTATION OF CPU SCHEDULING ALGORITHM – FIRST COME FIRST SERVE	30
4D.	IMPLEMENTATION OF CPU SCHEDULING ALGORITHM – PRIORITY SCHEDULING	33
5.	IMPLEMENTATION OF THE INTER PROCESS COMMUNICATION STRATEGY	37
6.	IMPLEMENTATION OF MUTUAL EXCLUSION BY SEMAPHORE – DINING PHILOSOPHER	41
7.	BANKER'S ALGORITHM FOR DEALOCK AVOIDANCE	46
8.	IMPLEMENTATION OF DEADLOCK DETECTION ALGORITHM	50
9.	IMPLEMENTATION OF THREADING & SYNCHRONIZATION APPLICATIONS	53
10.	IMPLEMENTATION OF PAGING TECHNIQUE FOR MEMORY MANAGEMENT	57
11A.	IMPLEMENTATION OF MEMORY ALLOCATION METHOD – FIRST FIT ALGORITHM	60
11B.	IMPLEMENTATION OF MEMORY ALLOCATION METHOD – WORST FIT ALGORITHM	63
11C.	IMPLEMENTATION OF MEMORY ALLOCATION METHOD – BEST FIT ALGORITHM	66

12A.	**IMPLEMENTATION OF PAGE REPLACEMENT ALGORITHM - FIFO**	**70**
12B.	**IMPLEMENTATION OF PAGE REPLACEMENT ALGORITHM - LRU**	**73**
12C.	**IMPLEMENTATION OF PAGE REPLACEMENT ALGORITHM - LFU**	**77**
13.	**IMPLEMENTATION OF FILE ORGANIZATION TECHNIQUES**	**81**
14A.	**IMPLEMENTATION OF SEQUENTIAL FILE ALLOCATION STRATEGIES**	**84**
14B.	**IMPLEMENTATION OF INDEXED FILE ALLOCATION STRATEGIES**	**87**
14C.	**IMPLEMENTATION OF LINKED FILE ALLOCATION STRATEGIES**	**90**
15A.	**IMPLEMENTATION OF DISK SCHEDULING ALGORITHM - FCFS**	**93**
15B.	**IMPLEMENTATION OF DISK SCHEDULING ALGORITHM - SSTF**	**95**
15C.	**IMPLEMENTATION OF DISK SCHEDULING ALGORITHM - SCAN**	**98**
15D.	**IMPLEMENTATION OF DISK SCHEDULING ALGORITHM - CSCAN**	**101**
15E.	**IMPLEMENTATION OF DISK SCHEDULING ALGORITHM - LOOK**	**105**
15F.	**IMPLEMENTATION OF DISK SCHEDULING ALGORITHM - CLOOK**	**108**
16.	**INSTALL ANY GUEST OPERATING SYSTEM LIKE LINUX USING VMWARE.**	**111**

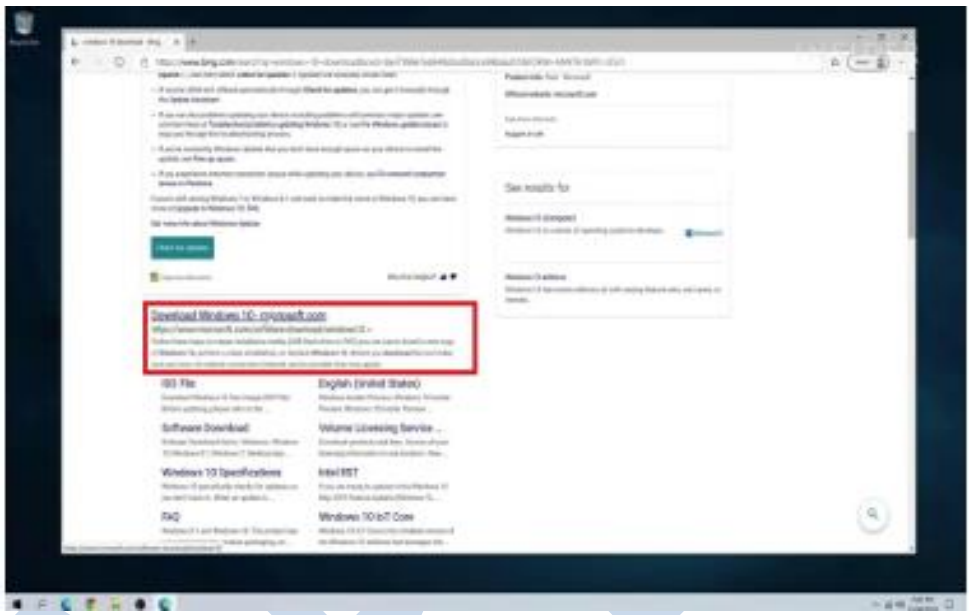
EX. NO: 1	INSTALLATION OF WINDOWS OPERATING SYSTEM

Aim:

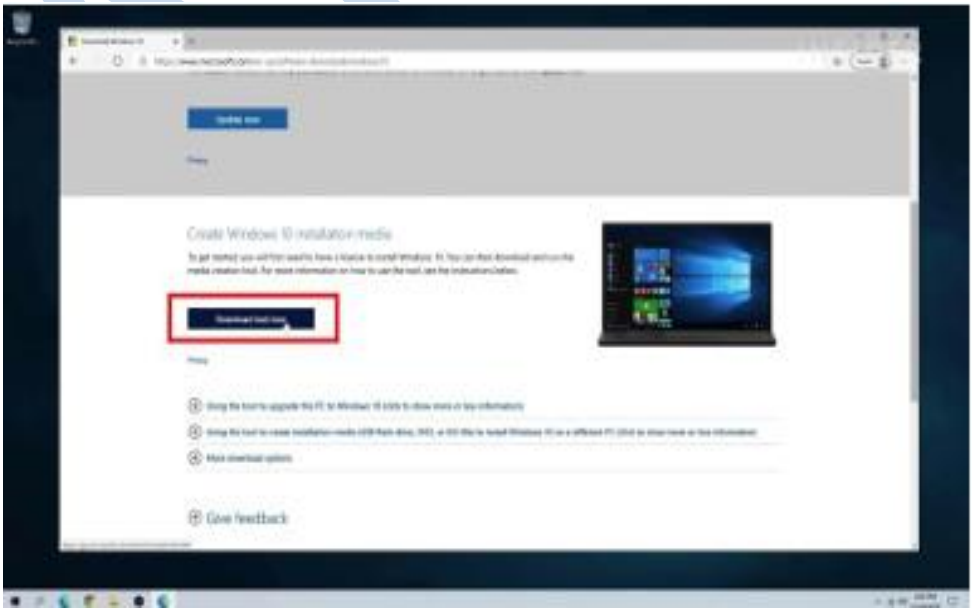
To install windows operating system.

Procedure:

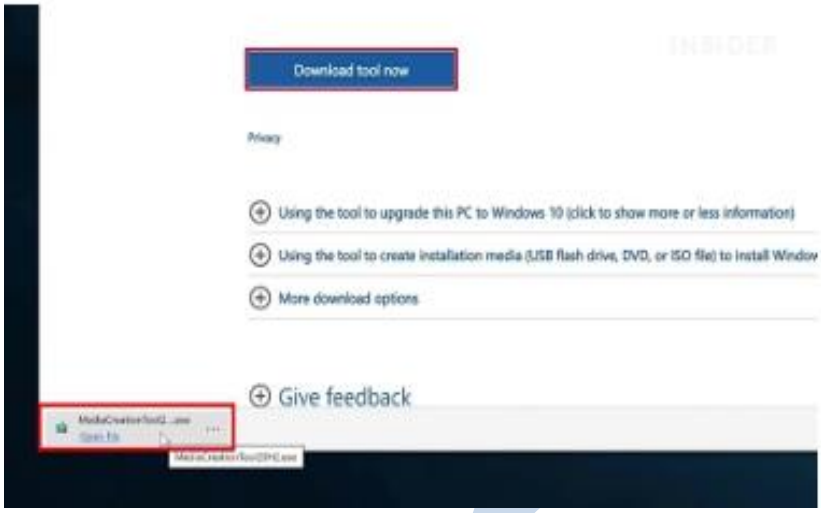
Step1:



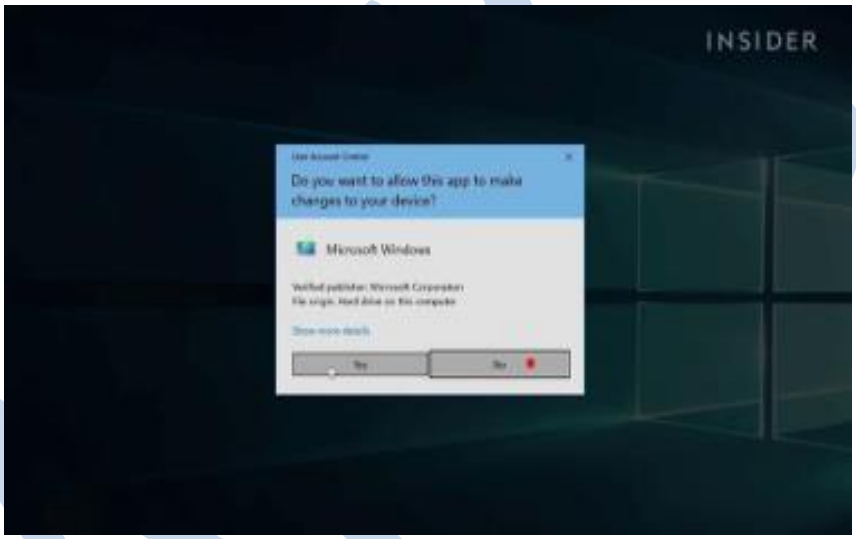
Step2:



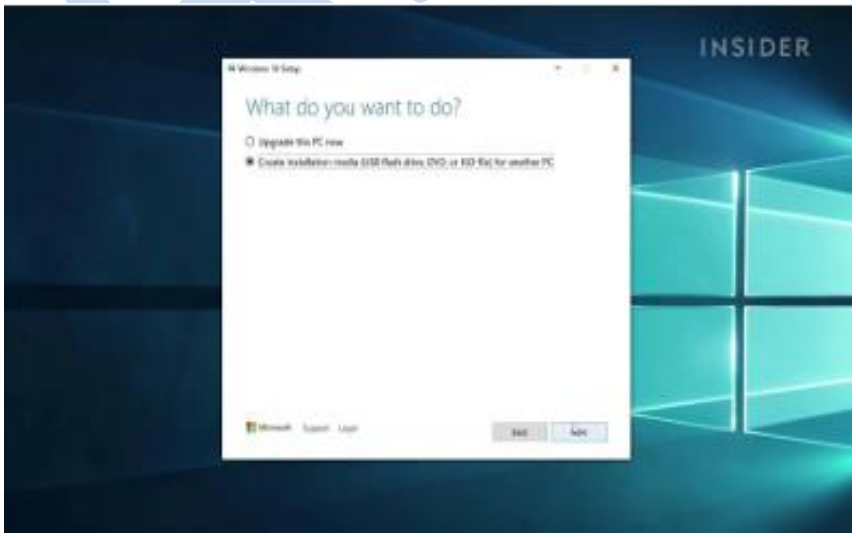
Step3:



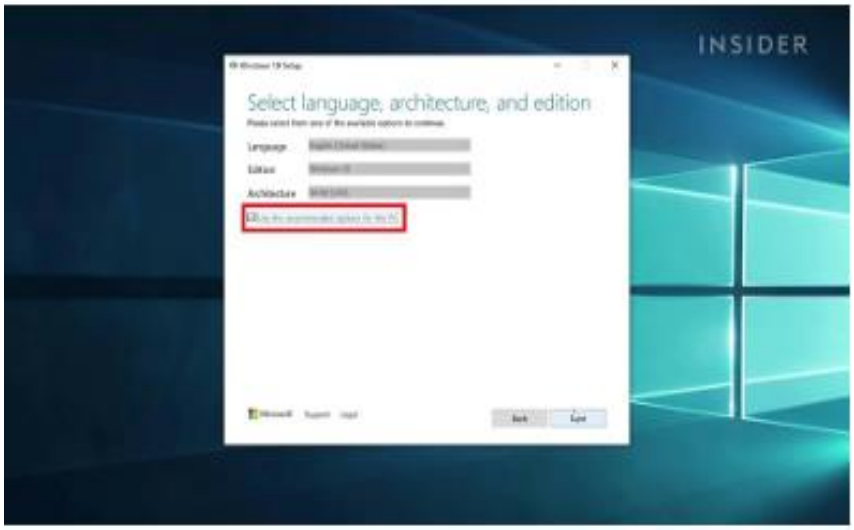
Step4:



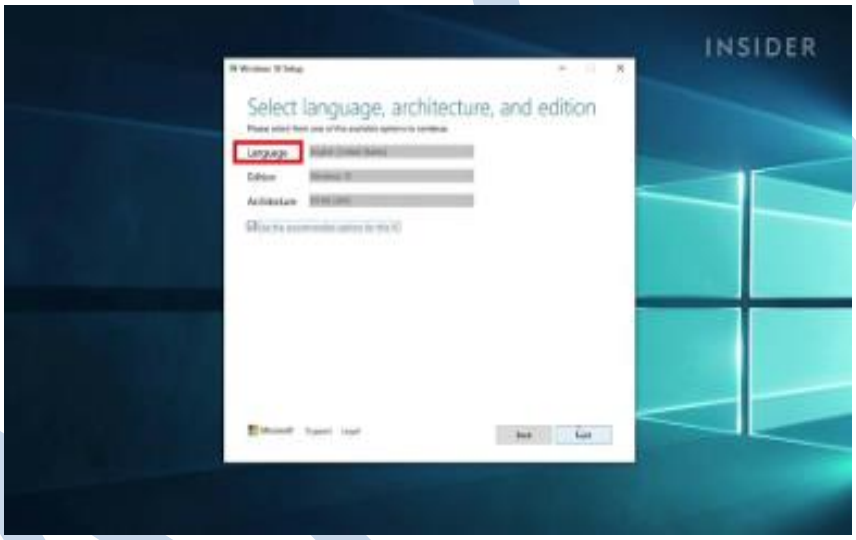
Step5:



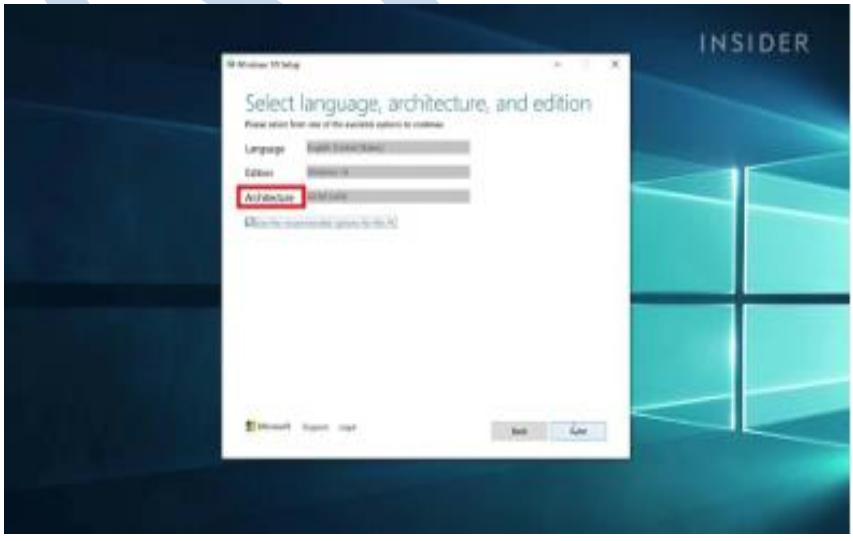
Step6:



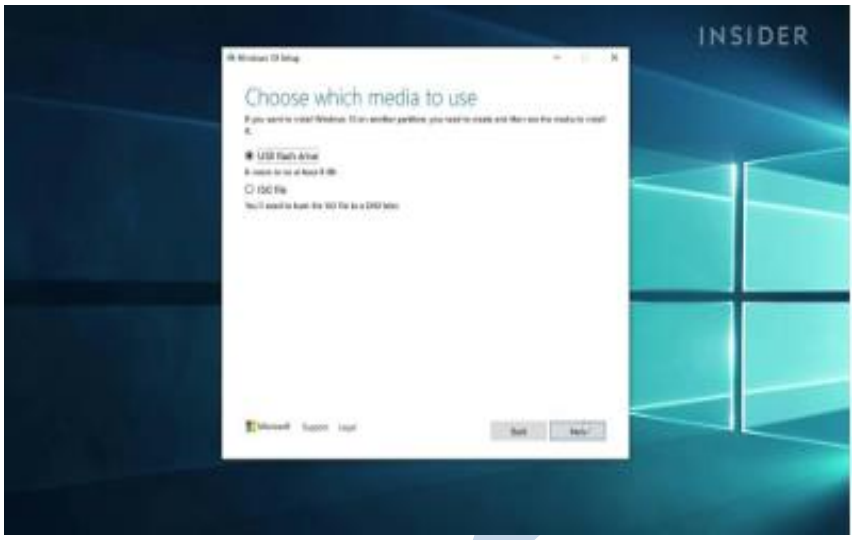
Step7:



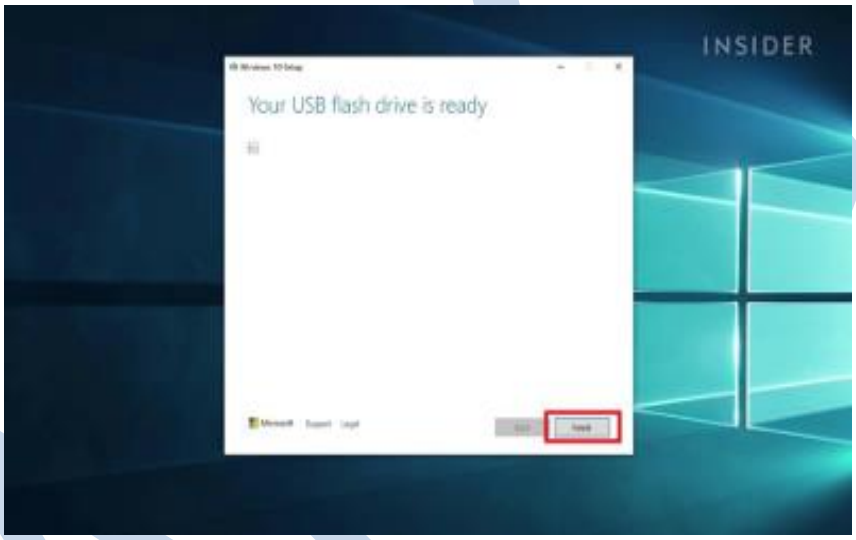
Step8:



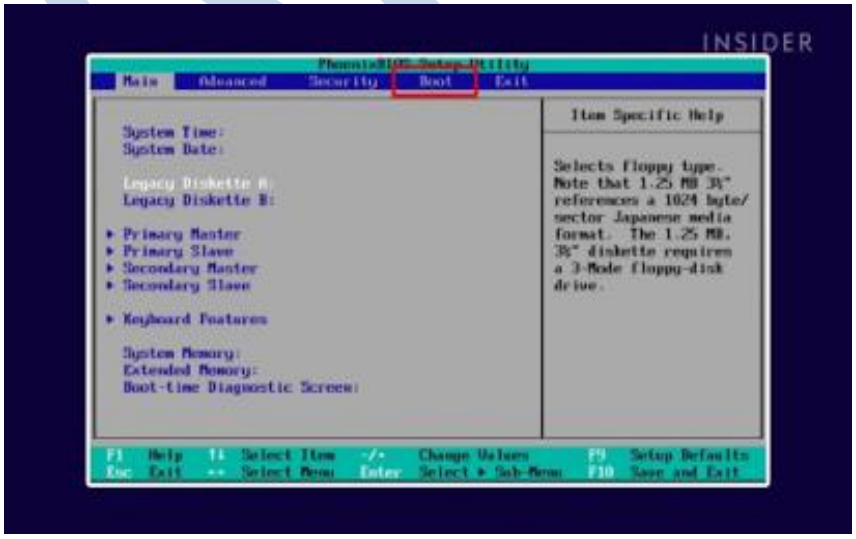
Step9:



Step10:

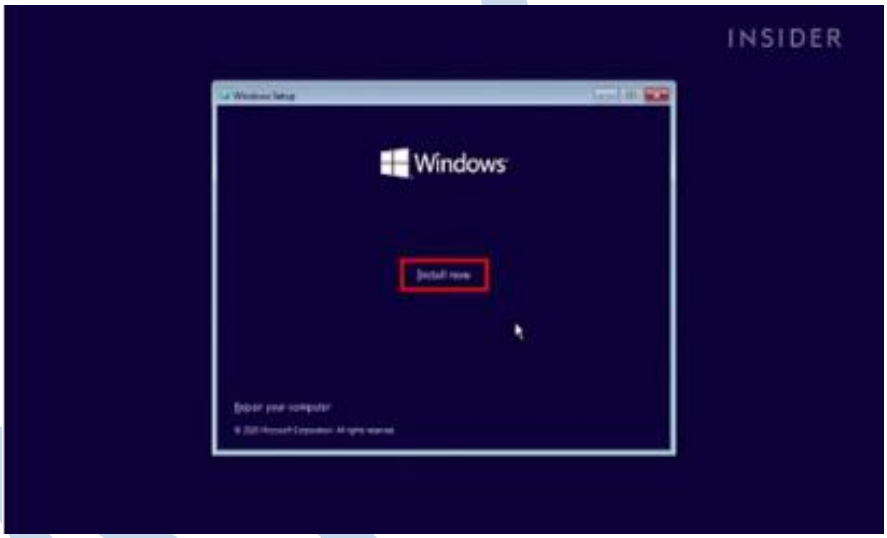


Step11:



Step12:

Step13:



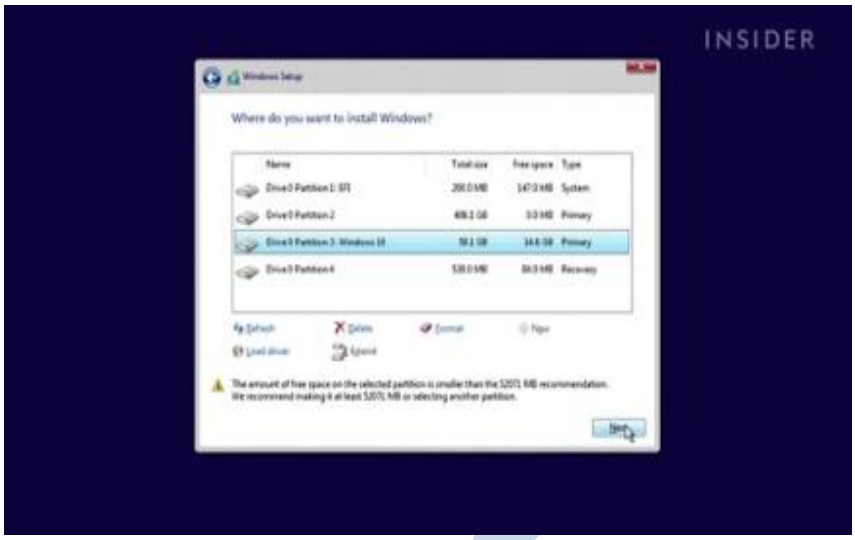
Step14:

Step15:

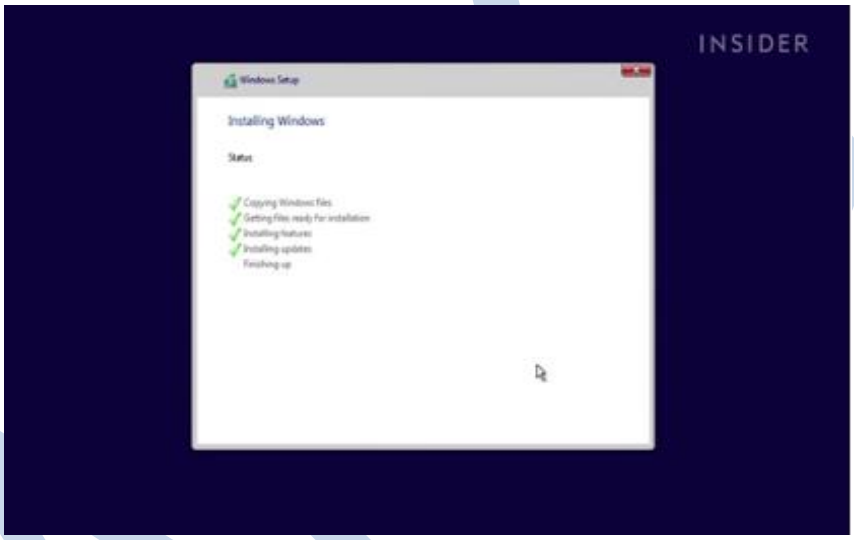
Step16:

Step17:

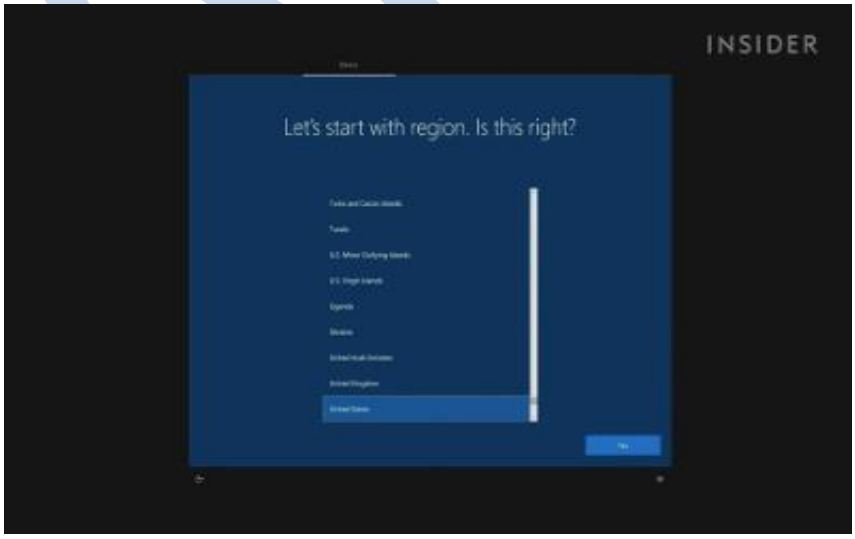
Step18:



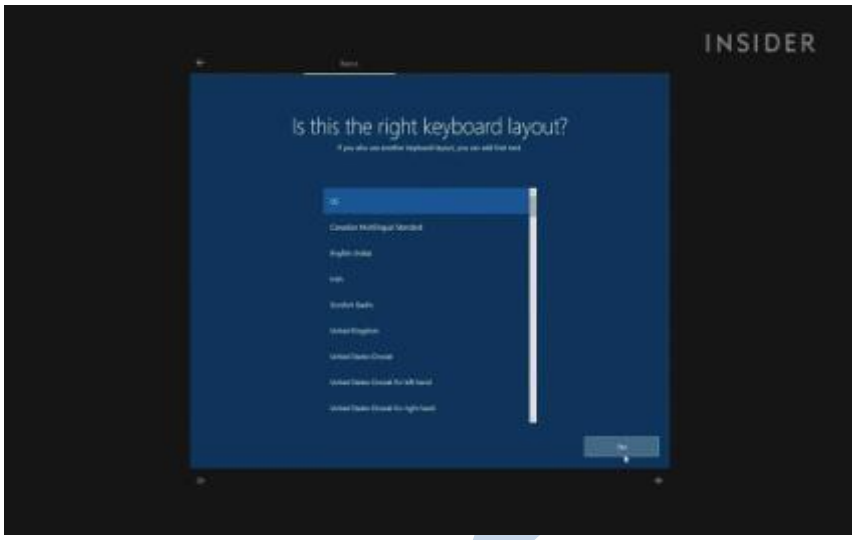
Step19:



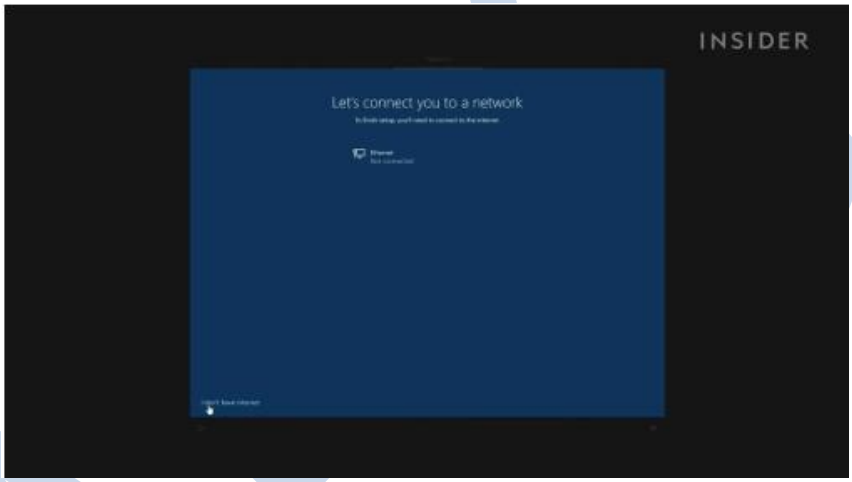
Step20:



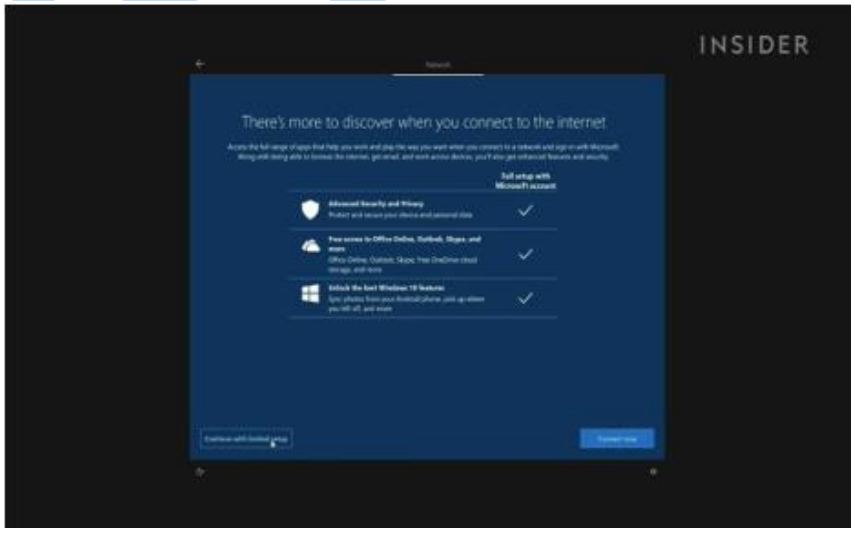
Step21:



Step22:

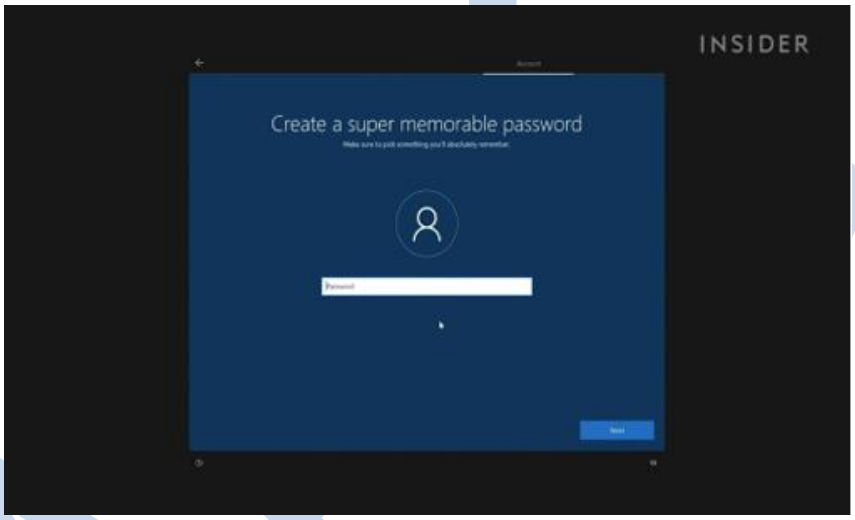


Step23:



Step24:

Step25:



Step26:

Step27:

Step28:

Result:

Thus the Installation of windows operating system is completed successfully.

EX. NO: 2A	ILLUSTRATE UNIX COMMANDS

Aim:

To study and execute the basic UNIX commands.

Description:

Command	Example	Description
ls	ls ls -alF	Lists files in current directory List in long format
cd	cd tempdir cd .. cd ~dhyatt/web-docs	Change directory to tempdir Move back one directory Move into dhyatt's web-docs directory
mkdir	mkdir graphics	Make a directory called graphics
rmdir	rmdir emptydir	Remove directory (must be empty)
cp	cp file1 web-docs cp file1 file1.bak	Copy file into directory Make backup of file1
rm	rm file1.bak rm *.tmp	Remove or delete file Remove all file
mv	mv old.html new.html	Move or rename files
more	more index.html	Look at file, one page at a time
lpr	lpr index.html	Send file to printer
man	man ls	Online manual (help) about command

Files and Directories:

Command	Description
Cat	Display File Contents
Cd	Changes Directory to dirname
Chgrp	Change file group
Chmod	Changing Permissions
Cp	Copy source file into destination
File	Determine file type
Find	Find files
Grep	Search files for regular expressions.
Head	Display first few lines of a file
Ln	Create softlink on oldname
Mkdir	Create a new directory dirname
More	Display data in paginated form.
Mv	Move (Rename) a oldname to newname.
Pwd	Print current working directory.
Rm	Remove (Delete) filename
Rmdir	Delete an existing directory provided it is empty.

Result:

Thus the basic UNIX commands were studied.

EX. NO: 2B	ILLUSTRATE SHELL PROGRAMMING

Aim:

To implement the shell programming.

Description:

The shell provides an interface to the UNIX system. It gathers input from user and executes programs based on that input. A shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavours of shells, just as there are different flavours of operating systems. Shell Types are The Bourne shell. If you are using a Bourne-type shell, the default prompt is the $ character. The C shell. If you are using a C-type shell, the default prompt is the % character.

Program:

Factorial of N Number:

```
echo "Total no of factorial wants"
read fact
ans=1 counter=0
while [ $fact -ne $counter ] do
done
counter=`expr $counter + 1`
ans=`expr $ans \* $counter
echo "Total of factorial is $ans"
```

To check the number Prime or not:

```
echo "Enter a number: " read num
i=2
while [ $i -lt
$num ] do
if [ `expr $num % $i` -eq 0 ] then
echo "$num is not a prime
number" echo "Since it is
divisible by $i" exit
fi
i=`expr
$i + 1`
done
echo "$num is a prime number "
```

Output:

Factorial of N Number:

Enter the number: 5

The Factorial is: 120

To check the number prime or not:

Enter the number: 7

7 is prime number

Result:

Thus the basic shell programming was implemented.

EX. NO: 3A	PROCESS MANAGEMENT USING SYSTEM CALLS : FORK, WAIT AND EXIT

Aim:

To write a program in C to implement the system calls fork, wait and exit.

Algorithm:

1. Start the program.
2. Declare the variables to store the status value and fork return value.
3. Check the fork return value.
4. If fork_return==0, then the child process is created.
5. If fork_return== -1, then the creation of child process is failed.
6. Display the status values.
7. Stop the program.

Program:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        int status;
        pid_t fork_return;
        fork_return = fork();
        if (fork_return == 0)
        {
                printf("\n I'm the child!!");
                exit(0);
        }
        else
        {
                wait(&status);
                printf("\n I'm the parent!!");
                printf("\n Child returned:%d\n", status);
        }
        return 0;
}
```

Output:

```
[exam127@redhat exam127]$ cc anufork.c
[exam127@redhat exam127]$ ./a.out

I'm the child!!
I'm the parent!!
Child returned: 0
```


Result:

Thus the implementation of system calls fork exit and wait has been executed successfully.

EX. NO: 3B	PROCESS MANAGEMENT USING SYSTEM CALLS : GETPID

Aim:

To write a program in C to implement the system calls getpid.

Algorithm:

1. Start the program.
2. Invoke the built in function getpid() which will return current process identification number.
3. Display the process ID.
4. Stop the program.

Program:

```
#include<stdio.h>
#include<unistd.h>
int main()
{
fork();
printf("\n Hello,I am the process with ID=%d\n",getpid());
return 0;
}
```

Output:

```
[exam127@redhat exam127]$ cc anugetpaid.c
[exam127@redhat exam127]$ ./a.out

Hello, I am the process with ID=27995
Hello, I am the process with ID=27994
```

Result:

Thus the implementation of system calls Getpid has been executed successfully.

EX. NO: 3C	PROCESS MANAGEMENT USING SYSTEM CALLS : CLOSE

Aim:

To write a program in C to implement the system call opendir (), readdir () and closedir().

Algorithm:

1. Start the program.
2. The function opendir() is invoked to open the directory which contains name of the directory a parameter.
3. The content of the directory can be read using the function readdir().
4. It contains the name of the directory as a parameter.
5. Display the content of the directory.
6. The directory is closed using the function closedir().
7. Stop the program.

Program:

```
#include<sys/types.h>
#include<dirent.h>
#include<sys/stat.h>
#include<stdio.h>
void traverse(char *fn)
{
        DIR *dir;
        struct dirent *entry;
        char path[1025];
        struct stat info;
        printf("%s\n",fn);
        if((dir=opendir(fn))==NULL)
        printf("error");
        else
        {
                while((entry=readdir(dir))!=NULL)
                {
                        if((entry->d_name[0])!='.')
```

```
                        {
                                strcpy(path,fn);
                                strcat(path,"/");
                                strcat(path,(entry->d_name));
                                if(stat(path,&info)!=0)
                                printf("error");
                                else if(S_ISDIR(info.st_mode))
                                traverse(path);
                        }
                }
        closedir(dir);
        }
}
main()
{
        printf("directory structure");
        traverse("/home/exam127/flower");
}
```

Output:

```
[exam127@redhat exam127]$ cc anuopen.c
```

```
[exam127@redhat exam127]$ ./a.out

directory structure/home/exam127/flower
/home/exam127/flower/lotus
/home/exam127/flower/rose
/home/exam127/flower/lilly
```


Result:

Thus the implementation of system calls Opendir(), Readdir() and Closedir() has been executed successfully.

EX. NO: 4A	IMPLEMENTATION OF CPU SCHEDULING ALGORITHM – ROUND ROBIN

Aim:

To simulate the CPU scheduling algorithm round-robin.

Algorithm:

1. Start the process
2. Accept the number of processes in the ready Queue and time quantum (or) time slice
3. For each process in the ready Q, assign the process id and accept the CPU burst time
4. Calculate the no. of time slices for each process where No. of time slice for process (n) = burst time process (n)/time slice
5. If the burst time is less than the time slice then the no. of time slices =1.
6. Consider the ready queue is a circular Q, calculate
   a) Waiting time for process (n) = waiting time of process(n-1)+ burst time of process(n-1) + the time difference in getting the CPU from process(n-1)
   b) Turnaround time for process(n) = waiting time of process(n) + burst time of process(n) + the time difference in getting CPU from process(n).
7. Calculate
   a) Average waiting time = Total waiting Time / Number of process
   b) Average Turnaround time = Total Turnaround Time / Number of process
8. Stop the process

Program:

```
#include <stdio.h>
main()
{
        int s[10], p[10], n, i, j, w1 = 0, w[10], t[10], st[10], tq, tst = 0;
        int tt = 0, tw = 0;
        float aw, at;
        printf("Enter no.of processes \n");
        scanf("%d", &n);
        printf("\n Enter the time quantum \n");
        scanf("%d", &tq);
        printf("\n Enter the process &service time of each process separated by a space \n");
        for (i = 0; i < n; i++)
        scanf("%d%d", &p[i], &s[i]);

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

```
        {
                st[i] = s[i];
                tst = tst + s[i];
        }

        for (j = 0; j < tst; j++)
                for (i = 0; i < n; i++)
                {
                        if (s[i] > tq)
                        {
                                s[i] = s[i] - tq;
                                w1 = w1 + tq;
                                t[i] = w1;
                                w[i] = t[i] - st[i];
                        }
                        else if (s[i] != 0)
                        {
                                w1 = w1 + tq;
                                18

                                t[i] = w1;
                                w[i] = t[i] - st[i];
                                s[i] = s[i] - tq;
                        }
                }

        for (i = 0; i < n; i++)
        {
                tw = tw + w[i];
                tt = tt + t[i];
        }
        aw = tw / n;
        at = tt / n;
        printf("process\tst\twt\ttt \n");
        for (i = 0; i < n; i++)
        printf("%d\t%d\t%d\t%d \n", p[i], st[i], w[i], t[i]);
        printf("awt=%f\n", aw);
        printf("att=%f\n", at);

}
```

Output:

Enter Number of process: 3

Enter the Time Quantum: 3

Enter the process and service time of each process separated by a space

1 3 2 6 3 9

Process	st	wt	tt
1	3	0	3
2	6	6	12
3	9	9	18

Awt=5.000000

Att=11.000000

Result:

Thus the Round Robin CPU scheduling algorithm was implemented and average waiting time, average turnaround time was computed.

EX. NO: 4B	IMPLEMENTATION OF CPU SCHEDULING ALGORITHM – SHORTEST JOB FIRST

Aim:

To write a C program to simulate the shortest job first CPU scheduling algorithm.

Algorithm:

1. Start the program.
2. Accept the number of processes in the ready Queue
3. For each process in the ready Q, assign the process id and accept the CPU burst time and process arrival time.
4. Start the Ready Q according the shortest Burst time by sorting according to lowest to highest burst time.
5. Set the waiting time of the first process as _0'and its turnaround time as its burst time.
6. Sort the processes names based on their Burt time and Arrival time
7. For each process in the ready queue, calculate
   a). Waiting times(n) = waiting time (n-1) + Burst time (n-1)-arrival time(n)
   b). Turnaround time (n) = waiting time (n) +Burst time (n)
8. Step 8: Calculate and print the results
9. (a) Average waiting time = Total waiting Time / Number of process
10. (b) Average Turnaround time = Total Turnaround Time / Number of process.
11. Stop the program.

Program:

```
#include <stdio.h>
main()
{
    int s[10], p[10], n, i, j, w1 = 0, w[10], t[10], st[10], tq, tst = 0;
    int tt = 0, tw = 0;
    float aw, at;
    clrscr();
    printf("Enter no.of processes \n");
    scanf("%d", &n);
    printf("\n Enter the time quantum \n");
    scanf("%d", &tq);
```

```
printf("\n Enter the process &service time of each process separated by a space \n");
for (i = 0; i < n; i++)
        scanf("%d%d", &p[i], &s[i]);
for (i = 0; i < n; i++)
{
        st[i] = s[i];
        tst = tst + s[i];
}

for (j = 0; j < tst; j++)
        for (i = 0; i < n; i++)
        {
                if (s[i] > tq)
                {
                        s[i] = s[i] - tq;
                        w1 = w1 + tq;
                        t[i] = w1;
                        w[i] = t[i] - st[i];
                }
                else if (s[i] != 0)
                {
                        20

                        w1 = w1 + tq;
                        t[i] = w1;
                        w[i] = t[i] - st[i];
                        s[i] = s[i] - tq;
                }
        }



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

```
            tw = tw + w[i];
            tt = tt + t[i];
        }

        aw = tw / n;
        at = tt / n;
        printf("process\tst\twt\ttt \n");
        for (i = 0; i < n; i++)
            printf("%d\t%d\t%d\t%d \n", p[i], st[i], w[i], t[i]);
        printf("awt=%f\n", aw);
        printf("att=%f\n", at);
        getch();
}
```

Output:

Enter number of process

3

Enter the Burst Time of Process 04

Enter the Burst Time of Process 13

Enter the Burst Time of Process 25

SHORTEST JOB FIRST SCHEDULING ALGORITHM

PROCESS ID	BURST TIME	WAITING TIME	TURNAROUND TIME
1	3	0	3
0	4	3	7
2	5	7	12

AVERAGE WAITING TIME 3.33

AVERAGE TURN AROUND TIME 7.33

Result:

Thus the Shortest Job First (SJF) CPU scheduling algorithm was implemented and average waiting time, average turnaround time was computed.

EX. NO: 4C	IMPLEMENTATION OF CPU SCHEDULING ALGORITHM – FIRST COME FIRST SERVE

Aim:

To write a c program to simulate the CPU scheduling algorithm First Come First Serve (FCFS)

Algorithm:

1. Start the process
2. Accept the number of processes in the ready Queue
3. For each process in the ready Q, assign the process name and the burst time
4. Set the waiting of the first process as _0'and its burst time as its turnaround time
5. For each process in the Ready Q calculate
   a) Waiting timess(n)= waiting time (n-1) + Burst time (n-1)
   b) Turnaround time (n)= waiting time(n)+Burst time(n)
6. Calculate
   a) Average waiting time = Total waiting Time / Number of process
   b) Average Turnaround time = Total Turnaround Time / Number of process
7. Stop the process

Program:

```
#include <stdio.h>
void main()
        {
                int i, n, sum, wt, tat, twt, ttat;
                int t[10];
                float awt, atat;
                clrscr();
                printf("Enter number of processors:\n");
                scanf("%d", &n);


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

```
                printf("\n Enter the Burst Time of the process %d", i + 1);
                scanf("\n %d", &t[i]);
        }
        printf("\n\n FIRST COME FIRST SERVE SCHEDULING ALGORITHM \n");
        printf("\n Process ID \t Waiting Time \t Turn Around Time \n");
        printf("1 \t\t 0 \t\t %d \n", t[0]);
        sum = 0;
        twt = 0;
        ttat = t[0];
        for (i = 1; i < n; i++)
        {
                sum += t[i - 1];
                wt = sum;
                tat = sum + t[i];
                twt = twt + wt;
                ttat = ttat + tat;
                printf("\n %d \t\t %d \t\t %d", i + 1, wt, tat);
                printf("\n\n");
        }
        awt = (float) twt / n;
        atat = (float) ttat / n;
        printf("\n Average Waiting Time %4.2f", awt);
        printf("\n Average Turnaround Time %4.2f", atat);
        getch();
}
```

Output:

Enter number of processors:

3

Enter the Burst Time of the process 1: 2

Enter the Burst Time of the process 2: 5

Enter the Burst Time of the process 3: 4

FIRST COME FIRST SERVE SCHEDULING ALGORITHM

Process ID	Waiting Time	Turn Around Time
1	0	2
2	2	7
3	7	11

Average Waiting Time 3.00

Average Turnaround Time 6.67

Result:

Thus the FCFS CPU scheduling algorithm was implemented and average waiting time, average turnaround time was computed.

EX. NO: 4D	IMPLEMENTATION OF CPU SCHEDULING ALGORITHM – PRIORITY SCHEDULING

Aim:

To write a c program to simulate the priority CPU scheduling algorithm.

Algorithm:

1. Start the process
2. Accept the number of processes in the ready Queue
3. For each process in the ready Q, assign the process id and accept the CPU burst time
4. Sort the ready queue according to the priority number.
5. Set the waiting of the first process as _0‘ and its burst time as its turnaround time
6. Arrange the processes based on process priority
7. For each process in the Ready Q calculate
8. For each process in the Ready Q calculate
   a) Waiting timess(n)= waiting time (n-1) + Burst time (n-1)
   b) Turnaround time (n)= waiting time(n)+Burst time(n)
9. Calculate
   a) Average waiting time = Total waiting Time / Number of process
   b) Average Turnaround time = Total Turnaround Time / Number of process
10. Print the results in an order.
11. Stop the process

Program:

```
#include <stdio.h>
main()
{
        int i, j, bt[10], n, pt[10], wt[10], tt[10], t, k, l, w1 = 0, t1 = 0, b = 0, p = 0;
        float at, aw;
        clrscr();
        printf("enter no of jobs");
        scanf("%d", &n);
        printf("enter burst time");
        for (i = 0; i < n; i++)
        {
                scanf("%d", &b);
                bt[i] = b;
        }
        printf("enter priority values");
        for (i = 0; i < n; i++)
        {
                scanf("%d", &p);
                pt[i] = p;
        }
        for (i = 1; i < n; i++)
                for (j = 0; j < n - i; j++)
                        if (pt[j] < pt[j + 1])
                        {
                                t = pt[j];
                                pt[j] = pt[j + 1];
                                pt[j + 1] = t;
                                k = bt[j];
                                bt[j] = bt[j + 1];
                                bt[j + 1] = k;
                        }

        wt[0] = 0;
        for (i = 0; i < n; i++)
```

```
        {
                wt[i + 1] = bt[i] + wt[i];
                tt[i] = bt[i] + wt[i];
                w1 = w1 + wt[i];
                t1 = t1 + tt[i];
        }
        aw = w1 / n;
        at = t1 / n;
        printf("\nbt\tprority\twt\ttt\n");
        for (i = 0; i < n; i++)
        printf("%d\t%d\t%d\t%d\n", bt[i], pt[i], wt[i], tt[i]);
        printf("aw=%f\nat=%f", aw, at);
        getch();
}
```


Output:

Enter no of jobs: 3

Enter burst time: 10 11 12

Enter priority values: 3 2 1

Bt	priority	wt	tt
10	3	0	10
11	2	10	21
12	1	21	33

Aw=10.000000

At=21.000000

Result:

Thus the Priority CPU scheduling algorithm was implemented and average waiting time, average turnaround time was computed.

EX. NO: 5	IMPLEMENTATION OF THE INTER PROCESS COMMUNICATION STRATEGY

Aim:

To develop a client-server application program, this uses shared memory using Inter Process Communication (IPC).

Algorithm:

Client:

1. Define the key to be 5600
2. Attach the client to the shared memory created by the server.
3. Read the content from the shared memory.
4. Display the content on the screen.

Server:

1. Define shared memory size of 30 bytes
2. Define the key to be 5600
3. Create a shared memory using shmget () system calls and gets the shared memory id in variable shmid.
4. Attach the shared memory to server data space
5. Get the content to be placed in the shared memory from the user of the server.
6. Write the content in the shared memory, which will read out by the client.
7. Stop

Program:

Shared memory and IPC

Server:

```
#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdio.h>

#include <stdlib.h>

#define MAXSIZE 27

void die(char *s)

        {
```

```
                perror(s);
                exit(1);
        }
int main()
        {
                char c;
                int shmid;
                key_t key;
                char *shm, *s;
                key = 5678;
                if ((shmid = shmget(key, MAXSIZE, IPC_CREAT | 0666)) < 0)
                die("shmget");
                if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
                die("shmat");
                s = shm;
                for (c = 'a'; c <= 'z'; c++)
                *s++ = c;
                while (*shm != '*')
                sleep(1);
                exit(0);
        }
```

Client:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#define MAXSIZE 27
void die(char *s)
        {
                perror(s);
                exit(1);
        }
int main()
        {
                int shmid;
                key_t key;
                char *shm, *s;
                key = 5678;
                if ((shmid = shmget(key, MAXSIZE, 0666)) < 0)
                die("shmget");
                if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
                die("shmat");
                for (s = shm; *s != '\0'; s++)
                putchar(*s);
                putchar('\n');
                *shm = '*';
                exit(0);
        }
```

Output:

```
[gokul@localhost ~]$ ipcs -m
 ------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x0000162e 98307      gokul      666        27         1

[gokul@localhost ~]$ cc shmclient1.c
[gokul@localhost ~]$ ./a.out
```

abcdefghijklmnopqrstuvwxyz

Result:

Thus the above program executed and verified successfully.

EX. NO: 6	IMPLEMENTATION OF MUTUAL EXCLUSION BY SEMAPHORE – DINING PHILOSOPHER

Aim:

To implement dining philosopher problem using semaphores.

Algorithm:

1. There are N philosophers meeting around a table, eating spaghetti and talking about philosophy.
2. There are only N forks available such that only one fork between each philosopher.
3. There are only 5 philosophers and each one requires 2 forks to eat.
4. A solution to the problem is to ensure that at most number of philosophers can eat Spaghetti at once.

Program:

```
#include <stdio.h
#define n 4
int compltedPhilo = 0, i;
struct fork
{
        int taken;
}
ForkAvil[n];
struct philosp
{
        int left;
        int right;
}
Philostatus[n];
void goForDinner(int philID)
{
        int otherFork = philID - 1;      //same like threads concept here cases implemented
        if (Philostatus[philID].left == 10 && Philostatus[philID].right == 10)
                printf("Philosopher %d completed his dinner\n", philID + 1);
        //if already completed dinner
```

```
        else if (Philostatus[philID].left == 1 && Philostatus[philID].right == 1)
        {
                //if just taken two forks
                printf("Philosopher %d completed his dinner\n", philID + 1);
                Philostatus[philID].left = Philostatus[philID].right = 10;      //remembering that
he completed dinner by assigning value 10
                if (otherFork == -1)
                        otherFork = (n - 1);
                ForkAvil[philID].taken = ForkAvil[otherFork].taken = 0;    //releasing forks
                printf("Philosopher %d released fork %d and fork %d\n", philID + 1, philID + 1,
otherFork + 1);
                compltedPhilo++;
        }
        else if (Philostatus[philID].left == 1 && Philostatus[philID].right == 0)
        {
                //left already taken, trying for
                right fork
                if (philID == (n - 1))
                {
                        if (ForkAvil[philID].taken == 0)
                        {
                                //KEY POINT OF THIS PROBLEM, THAT LAST
                                PHILOSOPHER TRYING IN reverse DIRECTION
                                ForkAvil[philID].taken = Philostatus[philID].right = 1;
                                printf("Fork %d taken by philosopher %d\n", philID + 1, philID + 1);
                        }
                        else
                        {
                        printf("Philosopher %d is waiting for fork %d\n", philID + 1, philID + 1);
                        }
                }


                else
                {
```

```
                26
                int dupphilID = philID;
                philID -= 1;
                if (philID == -1)
                        philID = (n - 1);
                if (ForkAvil[philID].taken == 0)
                {
                ForkAvil[philID].taken = Philostatus[dupphilID].right = 1;
                printf("Fork %d taken by Philosopher %d\n", philID + 1, dupphilID + 1);
                }
                else
                {
                printf("Philosopher %d is waiting for Fork %d\n", dupphilID + 1, philID+ 1);
                }
        }
}
else if (Philostatus[philID].left == 0)
{
        //nothing taken yet
        if (philID == (n - 1))
        {
                if (ForkAvil[philID - 1].taken == 0)
                {
                        PHILOSOPHER TRYING IN reverse DIRECTION
                        ForkAvil[philID - 1].taken = Philostatus[philID].left = 1;
                        printf("Fork %d taken by philosopher %d\n", philID, philID + 1);
                }
                else
                {
                printf("Philosopher %d is waiting for fork %d\n", philID + 1, philID);
                }
        }
        else
        {
                //except last philosopher case
```

```
                        if (ForkAvil[philID].taken == 0)
                        {
                        ForkAvil[philID].taken = Philostatus[philID].left = 1;
                        printf("Fork %d taken by Philosopher %d\n", philID + 1, philID + 1);
                        }
                        else
                        {
                        printf("Philosopher %d is waiting for Fork %d\n", philID +1, philID+ 1);
                        }
                }
        }
        else {}
}
int main()
{
        for (i = 0; i < n; i++)
                ForkAvil[i].taken = Philostatus[i].left = Philostatus[i].right = 0;
        while (compltedPhilo < n)
        {
                for (i = 0; i < n; i++)
                goForDinner(i);
                printf("\nTill now num of philosophers completed dinner are %d\n\n",
        compltedPhilo);
        }
        return 0;
}
```

Output:

```
Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Philosopher 3 released fork 3 and fork 2
Fork 3 taken by philosopher 4

Till now num of philosophers completed dinner are 3

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Fork 4 taken by philosopher 4

Till now num of philosophers completed dinner are 3

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Philosopher 4 completed his dinner
Philosopher 4 released fork 4 and fork 3

Till now num of philosophers completed dinner are 4
```

Result:

Thus the program for demonstrating Dining-Philosopher problem was implemented.

EX. NO: 7	BANKER'S ALGORITHM FOR DEALOCK AVOIDANCE

Aim:

To simulate bankers algorithm for Dead Lock Avoidance.

Algorithm:

1. Start the program.
2. Get the values of resources and processes.
3. Get the avail value.
4. After allocation find the need value.
5. Check whether it's possible to allocate.
6. If it is possible then the system is in safe state.
7. Else system is not in safety state.
8. If the new request comes then check that the system is in safety or not if we allow the request.
9. Stop the program.

Program:

```
#include <stdio.h>
main()
{
        int r[1][10], av[1][10];
        int all[10][10], max[10][10], ne[10][10], w[10], safe[10];
        int i = 0, j = 0, k = 0, l = 0, np = 0, nr = 0, count = 0, cnt = 0;
        printf("enter the number of processes in a system");
        scanf("%d", &np);
        printf("enter the number of resources in a system");
        scanf("%d", &nr);
        for (i = 1; i <= nr; i++)
        {
                printf("\n enter the number of instances of resource R%d ", i);
                scanf("%d", &r[0][i]);
                av[0][i] = r[0][i];
        }
        for (i = 1; i <= np; i++)
                for (j = 1; j <= nr; j++)
                        all[i][j] = ne[i][j] = max[i][j] = w[i] = 0;
        printf(" \nEnter the allocation matrix");
        for (i = 1; i <= np; i++)
```

```
        {
                printf("\n");
                for (j = 1; j <= nr; j++)
                {
                        scanf("%d", &all[i][j]);
                        av[0][j] = av[0][j] - all[i][j];
                }
        }
        printf("\nEnter the maximum matrix");
        for (i = 1; i <= np; i++)
        {
                printf("\n");
                for (j = 1; j <= nr; j++)
                {
                        scanf("%d", &max[i][j]);
                }
        }
        for (i = 1; i <= np; i++)
        {
                for (j = 1; j <= nr; j++)
                        31
                {

                        ne[i][j] = max[i][j] - all[i][j];
                }
        }
        for (i = 1; i <= np; i++)
        {
                printf("pocess P%d", i);
                for (j = 1; j <= nr; j++)
                {
                        printf("\n allocated %d\t", all[i][j]);
                        printf("maximum %d\t", max[i][j]);
                        printf("need %d\t", ne[i][j]);
                }
                printf("\n \n");
        }
        printf("\nAvailability");
        for (i = 1; i <= nr; i++)
                printf("R%d %d\t", i, av[0][i]);

        printf("\n ");
        printf("\n safe sequence");
        for (count = 1; count <= np; count++)
        {
```

```
                for (i = 1; i <= np; i++)
                {
                        cnt = 0;
                        for (j = 1; j <= nr; j++)
                        {
                                if (ne[i][j] <= av[0][j] && w[i] == 0)
                                        cnt++;
                        }
                        if (cnt == nr)
                        {
                                k++;
                                safe[k] = i;
                                for (l = 1; l <= nr; l++)
                                        av[0][l] = av[0][l] + all[i][l];
                                printf("\n P%d ", safe[k]);
                                printf("\t Availability");
                                for (l = 1; l <= nr; l++)
                                        printf("R%d %d\t", l, av[0][l]);
                                w[i] = 1;
                        }
                }
        }
}
```

Output:

Enter the number of resources in a system 4
Enter the number of instances of resource R1 2
Enter the number of instances of resource R2 2
Enter the number of instances of resource R3 2

Enter the number of instances of resource R4 2
Enter the allocation matrix
1 0 1 1 0
1 1 0 0 0
0 0 0 1 0
0 0 0 0 0
Enter the maximum matrix
1 0 1 0 0
1 0 0 0 0
0 0 0 0 1

pocess P1
allocated 1 maximum 0 need -1
allocated 0 maximum 0 need 0
allocated 1 maximum 0 need -1
allocated 1 maximum 0 need -1

pocess P2
allocated 0 maximum 1 need 1
allocated 1 maximum 0 need -1
allocated 1 maximum 1 need 0
allocated 0 maximum 0 need 0

pocess P3
allocated 0 maximum 0 need 0
allocated 0 maximum 1 need 1
allocated 0 maximum 0 need 0
allocated 0 maximum 0 need 0

pocess P4
allocated 0 maximum 0 need 0
allocated 1 maximum 0 need -1
allocated 0 maximum 0 need 0
allocated 0 maximum 0 need 0
AvailabilityR1 1 R2 0 R3 0 R4 1

safe sequence
P1 AvailabilityR1 2 R2 0 R3 1 R4 2
P2 AvailabilityR1 2 R2 1 R3 2 R4 2
P3 AvailabilityR1 2 R2 1 R3 2 R4 2
P4 AvailabilityR1 2 R2 2 R3 2 R4 2

Result:

Thus the banker's algorithm for deadlock avoidance was simulated.

EX. NO: 8	IMPLEMENTATION OF DEADLOCK DETECTION ALGORITHM

Aim:

To implement an algorithm for deadlock detection.

Algorithm:

Simply detects the existence of a cycle:

1. Start at any vertex finds all its immediate neighbours.
2. From each of these find all immediate neighbours, etc.
3. Until a vertex repeats (there is a cycle) or one cannot continue (there is no cycle).
4. Stop.

On a copy of the graph:

1. See if any Processes NEEDs can all be satisfied.
2. If so satisfy the needs with holds and remove that Process and all the Resources it holds from the graph.
3. If any Process are left Repeat step a
4. If all Processes are finally removed by this procedure there is no Deadlock in the original graph, if not there is.
5. Stop.

Program:

```
#include <stdio.h>
static int mark[20];
int i, j, np, nr;
int main()
{
        int alloc[10][10], request[10][10], avail[10], r[10], w[10];
        printf("\nEnter the no of process: ");
        scanf("%d", &np);
        printf("\nEnter the no of resources: ");
        scanf("%d", &nr);
        for (i = 0; i < nr; i++)
        {
```

```
            printf("\nTotal Amount of the Resource R%d: ", i + 1);
            scanf("%d", &r[i]);
    }
    printf("\nEnter the request matrix:");
    for (i = 0; i < np; i++)
            for (j = 0; j < nr; j++)
                    scanf("%d", &request[i][j]);
    printf("\nEnter the allocation matrix:");
    for (i = 0; i < np; i++)
            for (j = 0; j < nr; j++)
                    scanf("%d", &alloc[i][j]);
    /*Available Resource calculation*/
    for (j = 0; j < nr; j++)
    {
            avail[j] = r[j];
            for (i = 0; i < np; i++)
            {
                    avail[j] -= alloc[i][j];
            }
    }
    //marking processes with zero allocation
    for (i = 0; i < np; i++)
    {
            int count = 0;
            for (j = 0; j < nr; j++)
            {
                    if (alloc[i][j] == 0)
                            count++;
                    else
                            break;
            }

            if (count == nr)
                    mark[i] = 1;
    }
    // initialize W with avail
    for (j = 0; j < nr; j++)
            w[j] = avail[j];
    //mark processes with request less than or equal to W
    for (i = 0; i < np; i++)
    {
```

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

                                for (j = 0; j < nr; j++)
                                        w[j] += alloc[i][j];
                        }
                }
        }
        //checking for unmarked processes
        int deadlock = 0;
        for (i = 0; i < np; i++)
                if (mark[i] != 1)
                        deadlock = 1;
        if (deadlock)
                printf("\n Deadlock detected");
        else
                printf("\n No Deadlock possible");
}
```

Output:

Enter the no of process: 4
Enter the no of resources: 5

Total Amount of the Resource R1: 2
Total Amount of the Resource R2: 1
Total Amount of the Resource R3: 1
Total Amount of the Resource R4: 2
Total Amount of the Resource R5: 1

Enter the request matrix:0 1 0 0 1
0 0 1 0 1
0 0 0 0 1
1 0 1 0 1

Enter the allocation matrix:1 0 1 1 0
1 1 0 0 0
0 0 0 1 0
0 0 0 0 0

Deadlock detected

Result:

Thus the deadlock detection algorithm was implemented.

EX. NO: 9	IMPLEMENTATION OF THREADING & SYNCHRONIZATION APPLICATIONS

Aim:

To implement threading & synchronization applications using c.

Algorithm:

1. Start the program
2. Read the Input
3. Allocate the memory
4. Process the input
5. Checking error
6. Print result

Program:

```
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
pthread_t tid[2];
int counter;
void* trythis(void *arg)
{
unsigned long i = 0;
counter += 1;
printf("\n Job %d has started\n", counter);
for(i=0; i<(0xFFFFFFFF);i++);
printf("\n Job %d has finished\n", counter);
return NULL;
}
int main(void)
{
int i = 0;
int error;
```

```
while(i < 2)
{
error = pthread_create(&(tid[i]), NULL, &trythis, NULL);
if (error != 0)
printf("\nThread can't be created : [%s]", strerror(error));
i++;
}
pthread_join(tid[0], NULL);
pthread_join(tid[1], NULL);
return 0;
}
```


Output:

```
gokul@localhost ~]$ cc filename.c -lpthread
```

Job 1 has started

Job 2 has started

Job 2 has finished

Job 2 has finished

Result:

Thus the threading and synchronization concept was implemented.

EX. NO: 10	IMPLEMENTATION OF PAGING TECHNIQUE FOR MEMORY MANAGEMENT

Aim:

To implement simple paging technique.

Algorithm:

1. Start the process
2. Declare the size with respect to page length
3. Check the need of replacement from the page to memory
4. Check the need of replacement from old page to new page in memory
5. Forma queue to hold all pages
6. Insert the page require memory into the queue
7. Check for bad replacement and page fault
8. Get the number of processes to be inserted
9. Display the values
10. Stop the process

Program:

```
#include <stdio.h>
#define max 25
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("%d", &nb);
        printf("Enter the number of files:");
        scanf("%d", &nf);
        printf("\nEnter the size of the blocks:-\n");
        for (i = 1; i <= nb; i++)
        {
```

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

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

Output:

Memory Management Scheme - First Fit

Enter the number of blocks:3

Enter the number of files:2

Enter the size of the blocks:-

Block 1:5

Block 2:2

Block 3:7

Enter the size of the files :-

File 1:1

File 2:4

File_no:	File_size :	Block_no:	Block_size:	Fragement
1	1	1	5	4
2	4	3	7	3

Result:

Thus the simple paging technique was implemented.

EX. NO: 11A	IMPLEMENTATION OF MEMORY ALLOCATION METHOD – FIRST FIT ALGORITHM

Aim:

To write a program to implement first fit algorithm for memory management.

Algorithm:

1. Start the process.
2. Declare the size.
3. Get the number of processes to be inserted.
4. Allocate the first hole that is big enough for searching.
5. Start the beginning of the set of holes.
6. If not start at the hole, which is sharing the previous first fit search end.
7. Compare the hole.
8. If large enough, then stop searching in the procedure.
9. Display the values.
10. Terminate the process.

Program:

```
#include<stdio.h>
#include<process.h>
void main()
{
int a[20],p[20],i,j,n,m;
clrscr();
printf("Enter no of blocks:\n");
scanf("%d",&n);
for(i=0;i<n;i++)
        {
                printf("\nEnter the %dst block size:",i);
                scanf("%d",&a[i]);
        }
printf("\nEnter the no of process:");
scanf("%d",&m);
for(i=0;i<m;i++)
```

```
        {
                printf("\nEnter the size of %dst process:",i);
                scanf("%d",&p[i]);
        }
for(i=0;i<n;i++)
{
        for(j=0;j<m;j++)
        {
                if(p[j]<=a[i])
                {
                        printf("The process %d allocated %d\n",j,i);
                        p[j]=10000;
                        break;
                }
        }
}
for(j=0;j<m;j++)
{
        if(p[j]!=10000)
        {
                printf("\n The process is not allocated \n",j);
        }
}
getch();
}
```

Output:

Enter no of blocks:

3

Enter the 0st block size:70

Enter the 1st block size:50

Enter the 2st block size:90

Enter the no of process:2

Enter the size of 0st process:50

Enter the size of 1st process:80

The process 0 allocated 0

The process 1 allocated 2

Result:

Thus the first fit algorithm was implemented successfully.

EX. NO: 11B	IMPLEMENTATION OF MEMORY ALLOCATION METHOD – WORST FIT ALGORITHM

Aim:

To write a program to implement worst fit algorithm for memory management.

Algorithm:

1. Read the number of processes and number of available memory blocks.
2. Next read the processes' memory requirements.
3. Initialize the sizes of each memory block.
4. For first fit algorithm, select the blocks in given order that are greater than or equal to that of the process' requirements.
5. For best fit, sort the memory blocks in ascending order. Then choose the suitable memory block for each process.
6. For worst fit, sort the memory blocks in descending order. Then choose the suitable memory block for each process.
7. The internal fragmentation is computed by adding the remaining memory available in the allocated memory blocks.
8. The external fragmentation is computed by adding the unallocated memory blocks.

Program:

```
#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];
        clrscr();
        printf("\n\tMemory Management Scheme - Worst Fit");
        printf("\nEnter the number of blocks:");
        scanf("%d", &nb);
        printf("Enter the number of files:");
        scanf("%d", &nf);
        printf("\nEnter the size of the blocks:-\n");
```

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

Enter the no of blocks: 3

Enter the no of files: 2

Enter the size of blocks

Block 1:5

Block 2: 2

Block 3:7

Enter the size of the files:

File 1:1

File 2:4

File no	File size	Block no	Block size	Fragment
1	1	3	7	6
2	4	1	5	1

Result:

Thus the Worst fit algorithm was implemented successfully.

EX. NO: 11C	IMPLEMENTATION OF MEMORY ALLOCATION METHOD – BEST FIT ALGORITHM

Aim:

To write a program to implement best fit algorithm for memory management.

Algorithm:

1. Start the process.
2. Declare the size.
3. Get the number of processes to be inserted.
4. Allocate the best hole that is small enough for searching.
5. Start at the best of the set of holes.
6. If not start at the hole, which is sharing the previous best fit search end.
7. Compare the hole.
8. If small enough, then stop searching in the procedure.
9. Display the values.
10. Terminate the process.

Program:

```
#include <stdio.h>
#include <process.h>
void main()
{
        int a[20], p[20], i, j, n, m, temp, b[20], temp1, temp2, c[20];
        clrscr();
        printf("Enter the no of blocks:\n");
        scanf("%d", &n);
        for (i = 0; i < n; i++)
        {
                printf("Enter the %dst block size:", i);
                scanf("%d", &a[i]);
                b[i] = i;
        }
        printf("Enter the no of process:");
        scanf("%d", &m);
```

```
for (i = 0; i < m; i++)
{
        printf("Enter the size of %dst process:", i);
        scanf("%d", &p[i]);
        c[i] = i;
}
for (i = 0; i < n; i++)
{
        for (j = 0; j < m; j++)
        {
                if (a[i] < a[j])
                {
                        temp = a[i];
                        temp1 = b[i];
                        a[i] = a[j];
                        b[i] = b[j];
                        a[j] = temp;
                        b[j] = temp1;
                }

                if (p[i] < p[j])
                {
                        temp = p[i];
                        temp2 = c[i];
                        p[i] = p[j];
                        c[i] = c[j];
                        p[j] = temp;
                        c[j] = temp2;
                }
        }
}
for (i = 0; i < n; i++)
{
        for (j = 0; j < m; j++)
        {
```

```
                        if (p[j] <= a[i])
                        {
                                printf("\n The process %d allocated to block %d\n", c[j], b[i]);
                                p[j] = 10000;
                                break;
                        }
                }
        }
        for (j = 0; j < m; j++)
        {
                if (p[j] != 10000)
                {
                        printf("The process %d is not allocated:", j);
                }
        }
        getch();
}
```

Output:

Enter the no of blocks:

3

Enter the 0st block size:50

Enter the 1st block size:70

Enter the 2st block size:90

Enter the no of process:2

Enter the size of 0st process:60

Enter the size of 1st process:80

The process 0 allocated to block 1

The process 1 allocated to block 2

Result:

Thus the Best fit algorithm was implemented successfully.

EX. NO: 12A	IMPLEMENTATION OF PAGE REPLACEMENT ALGORITHM - FIFO

Aim:

To implement FIFO page replacement technique.

Algorithm:

1. Start the process
2. Declare the size with respect to page length
3. Check the need of replacement from the page to memory
4. Check the need of replacement from old page to new page in memory
5. Forma queue to hold all pages
6. Insert the page require memory into the queue
7. Check for bad replacement and page fault
8. Get the number of processes to be inserted
9. Display the values
10. Stop the process

Program:

```
#include<stdio.h>
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("\tref string\t page frames\n");
        for(i=1;i<=n;i++)
        {
                if(frame[k]==a[i])
                printf("%d\t\t",a[i]);
                avail=0;
                for(k=0;k<no;k++)
                avail=1;
```

```
                if (avail==0)
                {
                 frame[j]=a[i];
                j=(j+1)%no;
                count++;
                for(k=0;k<no;k++)
                printf("%d\t",frame[k]);
                }
        }
        printf("\n");
        printf("Page Fault Is %d",count);
        return 0;
}
```


Output:

ENTER THE NUMBER OF PAGES: 20

ENTER THE PAGE NUMBER: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

ENTER THE NUMBER OF FRAMES: 3

ref string	page frames		
7	7	-1	-1
0	7	0	-1
1	7	0	1
2	2	0	1

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

Page Fault Is 15

Result:

Thus the implementation of FIFO page replacement algorithm was executed and output verified.

EX. NO: 12B	IMPLEMENTATION OF PAGE REPLACEMENT ALGORITHM - LRU

Aim:

To implement LRU page replacement technique.

Algorithm:

1. Start the process
2. Declare the size
3. Get the number of pages to be inserted
4. Get the value
5. Declare counter and stack
6. Select the least recently used page by counter value
7. Stack them according the selection.
8. Display the values
9. Stop the process

Program:

```
#include <stdio.h>
main()
{
        int q[20], p[50], c = 0, c1, d, f, i, j, k = 0, n, r, t, b[20], c2[20];
        printf("Enter no of pages:");
        scanf("%d", &n);
        printf("Enter the reference string:");
        for (i = 0; i < n; i++)
                scanf("%d", &p[i]);
        printf("Enter no of frames:");
        scanf("%d", &f);
        q[k] = p[k];
        printf("\n\t%d\n", q[k]);
        c++;
        k++;
        for (i = 1; i < n; i++)
```

```
        {
                c1 = 0;
                for (j = 0; j < f; j++)
                {
                        if (p[i] != q[j])
                                c1++;
                }
                if (c1 == f)
                {
                        c++;
                        if (k < f)
                        {
                                q[k] = p[i];
                                k++;
                                for (j = 0; j < k; j++)
                                        printf("\t%d", q[j]);
                                printf("\n");
                        }
                        else
                        {
                                for (r = 0; r < f; r++)
                                {

                                        c2[r] = 0;
                                        for (j = i - 1; j < n; j--)
                                        {
                                                if (q[r] != p[j])
                                                        c2[r]++;
                                                else
                                                        break;
                                        }
                                }
                                for (r = 0; r < f; r++)
                                        b[r] = c2[r];
                                for (r = 0; r < f; r++)
                                {
                                        for (j = r; j < f; j++)
                                        {
                                                if (b[r] < b[j])
                                                {
                                                        t = b[r];
                                                        b[r] = b[j];
```

```
                                                b[j] = t;
                                        }
                                }
                        }
                        for (r = 0; r < f; r++)
                        {
                                if (c2[r] == b[0])
                                        q[r] = p[i];
                                printf("\t%d", q[r]);
                        }
                        printf("\n");
                }
        }
        printf("\nThe no of page faults is %d", c);
}
```

Output:

Enter no of pages: 10

Enter the reference string: 7 5 9 4 3 7 9 6 2 1

Enter no of frames: 3

7

7 5

7 5 9

4 5 9

4 3 9

4 3 7

9 3 7

9 6 7

9 6 2

1 6 2

The no of page faults is 10

Result:

Thus the LRU page replacement algorithm was implemented and output verified.

EX. NO: 12C	IMPLEMENTATION OF PAGE REPLACEMENT ALGORITHM - LFU

Aim:

To implement LFU page replacement technique.

Algorithm:

1. Start program
2. Read number of pages and frames
3. Read each page value
4. Search for page in the frames
5. If not available allocate free frame
6. If no frames is free replace the page with the page that is lastly used
7. Print page number of page faults
8. Stop process.

Program:

```
#include <stdio.h>
int main()
{
	int f, p;
	int pages[50], frame[10], hit = 0, count[50], time[50];
	int i, j, page, flag, least, minTime, temp;
	printf("Enter no of frames : ");
	scanf("%d", &f);
	printf("Enter no of pages : ");
	scanf("%d", &p);
	for (i = 0; i < f; i++)
	{
		frame[i] = -1;
	}
	for (i = 0; i < 50; i++)
	{
		count[i] = 0;
```

```
        }

        printf("Enter page no : \n");
        for (i = 0; i < p; i++)
        {
                scanf("%d", &pages[i]);
        }
        printf("\n");
        for (i = 0; i < p; i++)
        {
                count[pages[i]]++;
                time[pages[i]] = i;
                flag = 1;
                least = frame[0];
                for (j = 0; j < f; j++)
                {
                        if (frame[j] == -1 || frame[j] == pages[i])
                        {
                                if (frame[j] != -1)
                                {
                                        hit++;
                                        50
                                }
                                flag = 0;
                                frame[j] = pages[i];
                                break;
                        }
                        if (count[least] > count[frame[j]])
                        {
                                least = frame[j];
                        }
                }

                if (flag)
                {
```

```
                        minTime = 50;
                        for (j = 0; j < f; j++)
                        {
                                if (count[frame[j]] == count[least] && time[frame[j]] < minTime)
                                {
                                        temp = j;
                                        minTime = time[frame[j]];
                                }
                        }
                        count[frame[temp]] = 0;
                        frame[temp] = pages[i];
                }
                for (j = 0; j < f; j++)
                {
                        printf("%d ", frame[j]);
                }
                printf("\n");
        }
        printf("Page hit = %d", hit);
        return 0;
}
```

Output:

Enter no of frames: 3

Enter no of pages: 1 4 7 8 5 2 3 6 0 9

Enter page no:

4 -1 -1

Page hit = 0

Result:

Thus the LFU page replacement algorithm was implemented.

EX. NO: 13	IMPLEMENTATION OF FILE ORGANIZATION TECHNIQUES

Aim:

To implement Single level directory structure in C.

Algorithm:

1. Start
2. Declare the number, names and size of the directories and file names.
3. Get the values for the declared variables.
4. Display the files that are available in the directories.
5. Stop.

Program:

```
#include <stdio.h>
int main()
{
        int master, s[20];
        char f[20][20][20];
        char d[20][20];
        int i, j;
        printf("enter number of directorios:");
        scanf("%d", &master);
        printf("enter names of directories:");
        for (i = 0; i < master; i++)
                scanf("%s", &d[i]);
        printf("enter size of directories:");
        for (i = 0; i < master; i++)
                scanf("%d", &s[i]);
        printf("enter the file names :");
        for (i = 0; i < master; i++)
                for (j = 0; j < s[i]; j++)
                        scanf("%s", &f[i][j]);
        printf("\n");
```

```
        printf(" directory\tsize\tfilenames\n");
        for (i = 0; i < master; i++)
        {
                printf("%s\t\t%2d\t", d[i], s[i]);
                for (j = 0; j < s[i]; j++)
                        printf("%s\n\t\t\t", f[i][j]);
                printf("\n");
        }
        printf("\t\n");
}
```


Output:

enter number of directorios:3

enter names of directories: at er org

enter size of directories:1 1 1

enter the file names :a s d

directory size filenames

at 1 a

er 1 s

org 1 d

Result:

Thus the above program executed successfully.

EX. NO: 14A	IMPLEMENTATION OF SEQUENTIAL FILE ALLOCATION STRATEGIES

Aim:

To implement the sequential file allocation strategies

Algorithm:

1. Start the program.
2. Get the number of files.
3. Get the memory requirement of each file.
4. Allocate the required locations to each in sequential order
    a. Randomly select a location from available location s1= random(100);
    b. Check whether the required locations are free from the selected location.

```
if(b[s1].flag==0)
{
        for(j=s1;j<s1+p[i];j++) {
        if((b[j].flag)==0)
        count++;
}
if(count==p[i])
break;
}
```

    c. Allocate and set flag=1 to the allocated locations.

```
for(s=s1;s<(s1+p[i]);s++)
{
        k[i][j]=s;
        j=j+1;
        b[s].bno=s;
        b[s].flag=1;
}
```

5. Print the results fileno, lenth ,Blocks allocated.
6. Stop the program.

Program:

```
#include <stdio.h>
main()
{
	int f[50], i, st, j, len, c, k;

	for (i = 0; i < 50; i++)
		f[i] = 0;
	X:
	printf("\n Enter the starting block &length of file");
	scanf("%d%d", &st, &len);
	for (j = st; j < (st + len); j++)
		if (f[j] == 0)
		{
			f[j] = 1;
			printf("\n%d->%d", j, f[j]);
		}
	else
	{
		printf("Block already allocated");
		break;
	}
	if (j == (st + len))
	printf("\n the file is allocated to disk");
	printf("\n if u want to enter more files?(y-1/n-0)");
	scanf("%d", &c);
	if (c == 1)
		goto X;
	else
		exit();
}
```

Output:

Result:

Thus the file organization scheme was implemented.

EX. NO: 14B	IMPLEMENTATION OF INDEXED FILE ALLOCATION STRATEGIES

Aim:

To implement the linked file allocation technique.

Algorithm:

1) Start the program.
2) Get the number of files.
3) Get the memory requirement of each file.
4) Allocate the required locations by selecting a location randomly q= random(100);
   a. Check whether the selected location is free.
   b. If the location is free allocate and set flag=1 to the allocated locations. While allocating next location address to attach it to previous location

```
for(i=0;i<n;i++) {
for(j=0;j<s[i];j++) {
q=random(100);
if(b[q].flag==0)
b[q].flag=1;
b[q].fno=j;
r[i][j]=q;
if(j>0) {
p=r[i][j-1];
b[p].next=q; }
}
}
```

5) Print the results fileno, lenth ,Blocks allocated.
6) Stop the program

Program:

```
#include <stdio.h>
int f[50], i, k, j, inde[50], n, c, count = 0, p;
main()
{
        for (i = 0; i < 50; i++)
                f[i] = 0;
        x:
                printf("enter index block\t");
        scanf("%d", &p);
        if (f[p] == 0)
        {
                f[p] = 1;
                printf("enter no of files on index\t");
                scanf("%d", &n);
        }
        else
        {
                printf("Block already allocated\n");
                goto x;
        }

        for (i = 0; i < n; i++)
                scanf("%d", &inde[i]);
        for (i = 0; i < n; i++)
                if (f[inde[i]] == 1)
                {
                        printf("Block already allocated");
                        goto x;
                }
        for (j = 0; j < n; j++)
                f[inde[j]] = 1;
        printf("\n allocated");
        printf("\n file indexed");
        for (k = 0; k < n; k++)
                printf("\n %d->%d:%d", p, inde[k], f[inde[k]]);
        printf(" Enter 1 to enter more files and 0 to exit\t");
        scanf("%d", &c);
        if (c == 1)
                goto x;
        else
                exit();
}
```

Output:

Enter how many blocks that are already allocated 3

Enter the blocks no that are already al

Result:

Thus the file organization scheme was implemented.

EX. NO: 14C	IMPLEMENTATION OF LINKED FILE ALLOCATION STRATEGIES

Aim:

To implement the file allocation method using Linked method

Algorithm:

1. Start the program.
2. Get the number of files.
3. Get the memory requirement of each file.
4. Allocate the required locations by selecting a location randomly q= random(100);
   a) Check whether the selected location is free .
   b) If the location is free allocate and set flag=1 to the allocated locations

```
q=random(100);
{
    if(b[q].flag==0)
    b[q].flag=1;
    b[q].fno=j;
    r[i][j]=q;
}
```

5. Print the results fileno, lenth, Blocks allocated.
6. Stop the program.

Program:

```
#include <stdio.h>
#include <conio.h>
main()
{
    int f[50], p, i, j, k, a, st, len, n, c;
    clrscr();
    for (i = 0; i < 50; i++)
        f[i] = 0;
    printf("Enter how many blocks that are already allocated");
    scanf("%d", &p);
```

```
        printf("\nEnter the blocks no.s that are already allocated");
        for (i = 0; i < p; i++)
        {
                scanf("%d", &a);
                f[a] = 1;
        }
        X:
                printf("Enter the starting index block &length");
        scanf("%d%d", &st, &len);
        k = len;
        for (j = st; j < (k + st); j++)
        {
                if (f[j] == 0)
                {
                        f[j] = 1;
                        printf("\n%d->%d", j, f[j]);
                }
                else
                {
                        printf("\n %d->file is already allocated", j);
                        k++;
                }
        }
        printf("\n If u want to enter one more file? (yes-1/no-0)");
        scanf("%d", &c);
        if (c == 1)
                goto X;
        else
                exit();
        getch();
}
```

Output:

Enter index block 9

Enter no of files on index 3

1 2 3

Allocated file indexed

9->1:1

9->2:1

9->3:1

Enter 1 to enter more files and 0 to exit

Result:

Thus the file organization scheme was implemented.

EX. NO: 15A	IMPLEMENTATION OF DISK SCHEDULING ALGORITHM - FCFS

Aim:

To implement the disk scheduling using First Come First Serve Algorithm.

Algorithm:

1. Start the program.
2. Get the number of files.
3. Get the memory requirement of each file.
4. Allocate the required locations by selecting a location randomly
5. Print the results.
6. Stop the program.

Program:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
	int RQ[100], i, n, TotalHeadMoment = 0, initial;
	printf("Enter the number of Requests\n");
	scanf("%d", &n);
	printf("Enter the Requests sequence\n");
	for (i = 0; i < n; i++)
		scanf("%d", &RQ[i]);
	printf("Enter initial head position\n");
	scanf("%d", &initial);
// logic for FCFS disk scheduling
	for (i = 0; i < n; i++)
	{
		TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
		initial = RQ[i];
	}
	printf("Total head moment is %d", TotalHeadMoment);
	return 0;

}
```

Output:

Enter the number of Request

8

Enter the Requests Sequence

95 180 34 119 11 123 62 64

Enter initial head position

50

Total head movement is 644

Result:

Thus the disk scheduling using First Come First Serve was implemented successfully.

EX. NO: 15B	IMPLEMENTATION OF DISK SCHEDULING ALGORITHM - SSTF

Aim:

To implement the disk scheduling using Short Seek Time First.

Algorithm:

1. Start the program.
2. Get the number of files.
3. Get the memory requirement of each file.
4. Allocate the required locations by selecting a location randomly
5. Print the results.
6. Stop the program.

Program:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        int RQ[100], i, n, TotalHeadMoment = 0, initial, count = 0;
        printf("Enter the number of Requests\n");
        scanf("%d", &n);
        printf("Enter the Requests sequence\n");
        for (i = 0; i < n; i++)
                scanf("%d", &RQ[i]);
        printf("Enter initial head position\n");
        scanf("%d", &initial);

        // logic for sstf disk scheduling

        /*loop will execute until all process is completed*/
        while (count != n)
        {
                int min = 1000, d, index;
```

```
            for (i = 0; i < n; i++)
            {
                d = abs(RQ[i] - initial);
                if (min > d)
                {
                    min = d;
                    index = i;
                }
            }
            TotalHeadMoment = TotalHeadMoment + min;
            initial = RQ[index];
            // 1000 is for max
            // you can use any number
            RQ[index] = 1000;
            count++;
        }
        printf("Total head movement is %d", TotalHeadMoment);
        return 0;
}
```

Output:

Enter the number of Request

8

Enter the Requests Sequence

95 180 34 119 11 123 62 64

Enter initial head position

50

Total head movement is 236

Result:

Thus the disk scheduling using Short Seek Time First was implemented successfully.

EX. NO: 15C	IMPLEMENTATION OF DISK SCHEDULING ALGORITHM - SCAN

Aim:

To implement the disk scheduling using SCAN

Algorithm:

1. Start the program.
2. Get the number of files.
3. Get the memory requirement of each file.
4. Allocate the required locations by selecting a location randomly
5. Print the results.
6. Stop the program.

Program:

```
#include <stdio.h>
#include <math.h>
int main()
{
	int queue[20], n, head, i, j, k, seek = 0, max, diff, temp, queue1[20],
		queue2[20], temp1 = 0, temp2 = 0;
	float avg;
	printf("Enter the max range of disk\n");
	scanf("%d", &max);
	printf("Enter the initial head position\n");
	scanf("%d", &head);
	printf("Enter the size of queue request\n");
	scanf("%d", &n);
	printf("Enter the queue of disk positions to be read\n");
	for (i = 1; i <= n; i++)
	{
		scanf("%d", &temp);
		if (temp >= head)
		{
			queue1[temp1] = temp;
			temp1++;
		}
		else
		{
			queue2[temp2] = temp;
```

```
                        temp2++;
                }
        }
        for (i = 0; i < temp1 - 1; i++)
        {
                for (j = i + 1; j < temp1; j++)
                {
                        if (queue1[i] > queue1[j])
                        {
                                temp = queue1[i];
                                queue1[i] = queue1[j];
                                queue1[j] = temp;
                        }
                }
        }
        for (i = 0; i < temp2 - 1; i++)
        {
                for (j = i + 1; j < temp2; j++)
                {
                        if (queue2[i] < queue2[j])
                        {
                                temp = queue2[i];
                                queue2[i] = queue2[j];
                                queue2[j] = temp;
                        }
                }
        }
        for (i = 1, j = 0; j < temp1; i++, j++)
                queue[i] = queue1[j];
        queue[i] = max;
        for (i = temp1 + 2, j = 0; j < temp2; i++, j++)
                queue[i] = queue2[j];
        queue[i] = 0;
        queue[0] = head;
        for (j = 0; j <= n + 1; j++)
        {
                diff = abs(queue[j + 1] - queue[j]);
                seek += diff;
                printf("Disk head moves from %d to %d with seek %d\n", queue[j],
                        queue[j + 1], diff);
        }
        printf("Total seek time is %d\n", seek);
        avg = seek / (float) n;
        printf("Average seek time is %f\n", avg);
        return 0;
}
```

Output:

Enter the max range of disk

500

Enter the initial head position

100

Enter the size of queue request

6

Enter the queue of disk positions to be read

150

210

86

405

325

65

Disk head moves from 100 to 150 with seek 50

Disk head moves from 150 to 210 with seek 60

Disk head moves from 210 to 325 with seek 115

Disk head moves from 325 to 405 with seek 80

Disk head moves from 405 to 500 with seek 95

Disk head moves from 500 to 86 with seek 414

Disk head moves from 86 to 65 with seek 21

Disk head moves from 65 to 0 with seek 65

Total seek time is 900

Average seek time is 150.000000

Result:

Thus the disk scheduling using SCAN was implemented successfully.

EX. NO: 15D	IMPLEMENTATION OF DISK SCHEDULING ALGORITHM - CSCAN

Aim:

To implement the disk scheduling using Circular SCAN.

Algorithm:

1. Start the program.
2. Get the number of files.
3. Get the memory requirement of each file.
4. Allocate the required locations by selecting a location randomly
5. Print the results.
6. Stop the program.

Program:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        int RQ[100], i, j, n, TotalHeadMoment = 0, initial, size, move;
        printf("Enter the number of Requests\n");
        scanf("%d", &n);
        printf("Enter the Requests sequence\n");
        for (i = 0; i < n; i++)
                scanf("%d", &RQ[i]);
        printf("Enter initial head position\n");
        scanf("%d", &initial);
        printf("Enter total disk size\n");
        scanf("%d", &size);
        printf("Enter the head movement direction for high 1 and for low 0\n");
        scanf("%d", &move);
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n - i - 1; j++)
```

```
            {
                if (RQ[j] > RQ[j + 1])
                {
                    int temp;
                    temp = RQ[j];
                    RQ[j] = RQ[j + 1];
                    RQ[j + 1] = temp;
                }
            }
        }
        int index;
        for (i = 0; i < n; i++)
        {
            if (initial < RQ[i])
            {
                index = i;
                break;
            }
        }

        if (move == 1)
        {
            for (i = index; i < n; i++)
                66
            {

                TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
                initial = RQ[i];
            }
            TotalHeadMoment = TotalHeadMoment + abs(size - RQ[i - 1] - 1);
            TotalHeadMoment = TotalHeadMoment + abs(size - 1 - 0);
            initial = 0;
            for (i = 0; i < index; i++)
            {
                TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
```

```
                initial = RQ[i];
            }
        }
        // if movement is towards low value
        else
        {
            for (i = index - 1; i >= 0; i--)
            {
                TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
                initial = RQ[i];
            }

            TotalHeadMoment = TotalHeadMoment + abs(RQ[i + 1] - 0);
            TotalHeadMoment = TotalHeadMoment + abs(size - 1 - 0);
            initial = size - 1;
            for (i = n - 1; i >= index; i--)
            {
                TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
                initial = RQ[i];
            }
        }
        printf("Total head movement is %d", TotalHeadMoment);
        return 0;
}
```

Output:

Enter the number of Request

8

Enter the Requests Sequence

95 180 34 119 11 123 62 64

Enter initial head position

50

Enter total disk size

200

Enter the head movement direction for high 1 and for low 0

1

Total head movement is 382

Result:

Thus the disk scheduling using C - SCAN was implemented successfully.

EX. NO: 15E	IMPLEMENTATION OF DISK SCHEDULING ALGORITHM - LOOK

Aim:

To implement the disk scheduling using LOOK.

Algorithm:

1. Start the program.
2. Get the number of files.
3. Get the memory requirement of each file.
4. Allocate the required locations by selecting a location randomly
5. Print the results.
6. Stop the program.

Program:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        int RQ[100], i, j, n, TotalHeadMoment = 0, initial, size, move;
        printf("Enter the number of Requests\n");
        scanf("%d", &n);
        printf("Enter the Requests sequence\n");
        for (i = 0; i < n; i++)
                scanf("%d", &RQ[i]);
        printf("Enter initial head position\n");
        scanf("%d", &initial);
        printf("Enter total disk size\n");
        scanf("%d", &size);
        printf("Enter the head movement direction for high 1 and for low 0\n");
        scanf("%d", &move);
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n - i - 1; j++)
                {
                        if (RQ[j] > RQ[j + 1])
                        {
                                int temp;
                                temp = RQ[j];
                                RQ[j] = RQ[j + 1];
```

```
                                RQ[j + 1] = temp;
                        }
                }
        }
        int index;
        for (i = 0; i < n; i++)
        {
                if (initial < RQ[i])
                {
                        index = i;
                        break;
                }
        }
        if (move == 1)
        {
                68

                for (i = index; i < n; i++)
                {
                        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
                        initial = RQ[i];
                }
                for (i = index - 1; i >= 0; i--)
                {
                        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
                        initial = RQ[i];
                }
        }
        else
        {
                for (i = index - 1; i >= 0; i--)
                {
                        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
                        initial = RQ[i];
                }

                for (i = index; i < n; i++)
                {
                        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
                        initial = RQ[i];
                }
        }
        printf("Total head movement is %d", TotalHeadMoment);
        return 0;
}
```

Output:

Enter the number of Request

8

Enter the Requests Sequence

95 180 34 119 11 123 62 64

Enter initial head position

50

Enter the head movement direction for high 1 and for low 0

1

Total head movement is 299

Result:

Thus the disk scheduling using LOOK was implemented successfully.

EX. NO: 15F	IMPLEMENTATION OF DISK SCHEDULING ALGORITHM - CLOOK

Aim:

To implement the disk scheduling using CLOOK.

Algorithm:

1. Start the program.
2. Get the number of files.
3. Get the memory requirement of each file.
4. Allocate the required locations by selecting a location randomly
5. Print the results.
6. Stop the program.

Program:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
        int RQ[100], i, j, n, TotalHeadMoment = 0, initial, size, move;
        printf("Enter the number of Requests\n");
        scanf("%d", &n);
        printf("Enter the Requests sequence\n");
        for (i = 0; i < n; i++)
                scanf("%d", &RQ[i]);
        printf("Enter initial head position\n");
        scanf("%d", &initial);
        printf("Enter total disk size\n");
        scanf("%d", &size);
        printf("Enter the head movement direction for high 1 and for low 0\n");
        scanf("%d", &move);
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n - i - 1; j++)
                {
                        if (RQ[j] > RQ[j + 1])
                        {
                                int temp;
                                temp = RQ[j];
                                RQ[j] = RQ[j + 1];
```

```
                                RQ[j + 1] = temp;
                        }
                }
        }
        int index;
        for (i = 0; i < n; i++)
        {
                if (initial < RQ[i])
                {
                        index = i;
                        break;
                }
        }

        if (move == 1)
        {
                for (i = index; i < n; i++)
                {
                        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
                        initial = RQ[i];
                }

                for (i = index - 1; i >= 0; i--)
                {
                        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
                        initial = RQ[i];
                }
        }
        else
        {
                for (i = index - 1; i >= 0; i--)
                {
                        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
                        initial = RQ[i];
                }

                for (i = index; i < n; i++)
                {
                        TotalHeadMoment = TotalHeadMoment + abs(RQ[i] - initial);
                        initial = RQ[i];
                }
        }
        printf("Total head movement is %d \n", TotalHeadMoment);
        return 0;
}
```

Output:

Enter the number of Requests

3

Enter the Requests sequence

1

2

3

4

Enter initial head position

2

Enter total disk size

6

Enter the head movement direction for high 1 and for low 0

2

Total head movement is 4

Result:

Thus the disk scheduling using CLOOK was implemented successfully.

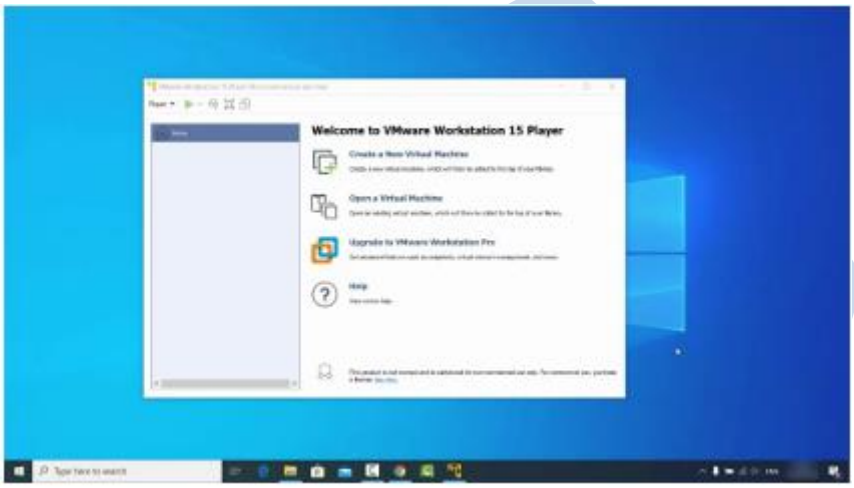
EX. NO: 16	INSTALL ANY GUEST OPERATING SYSTEM LIKE LINUX USING VMWARE.

Aim:

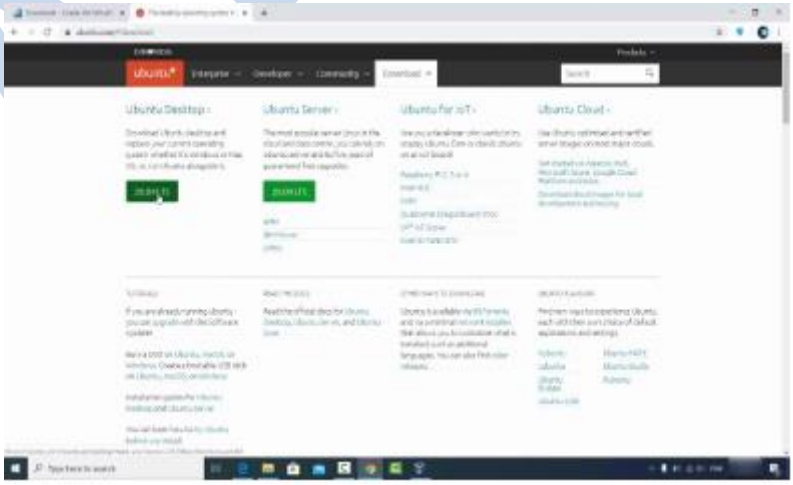
To install LINUX operating system using VMWARE.

Procedure:

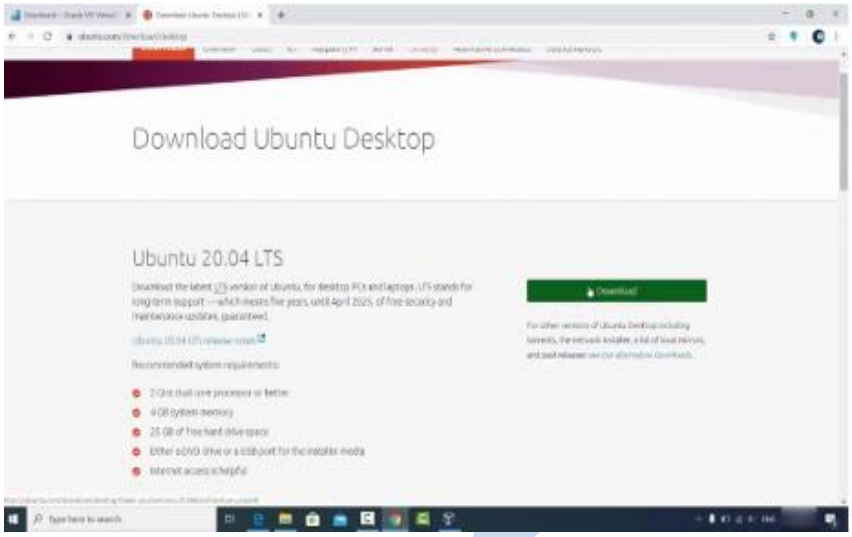
Step 1:



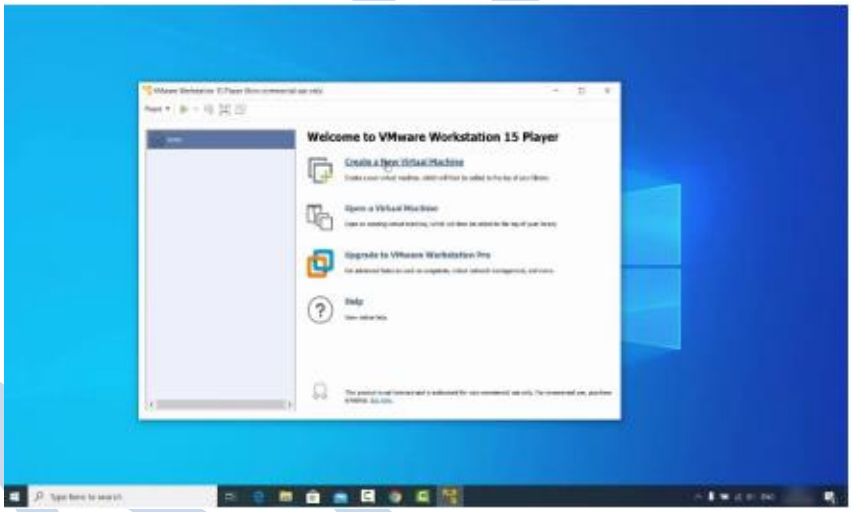
Step 2:



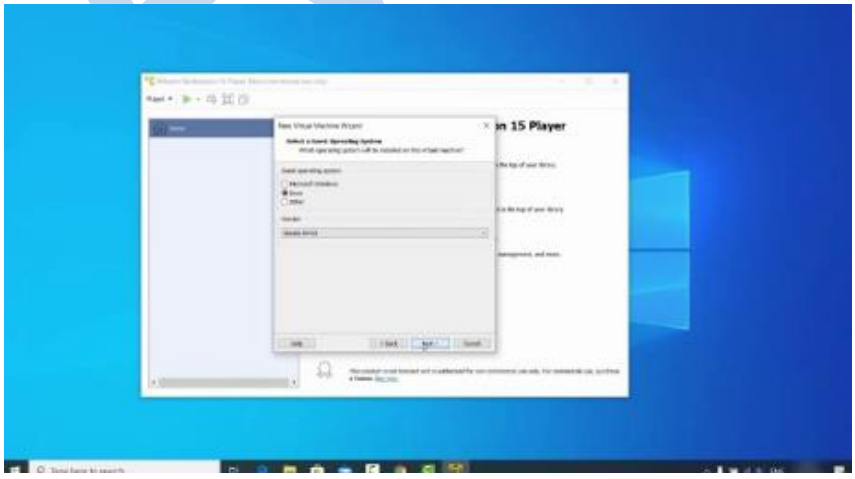
Step 3:



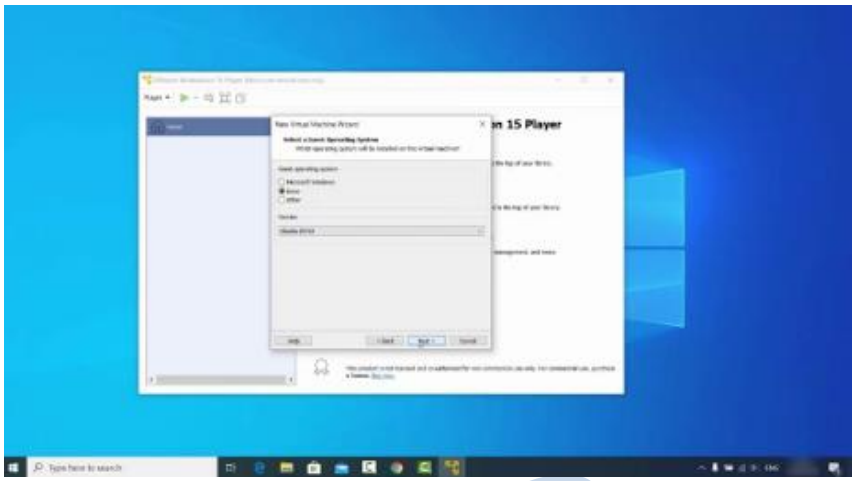
Step 4:



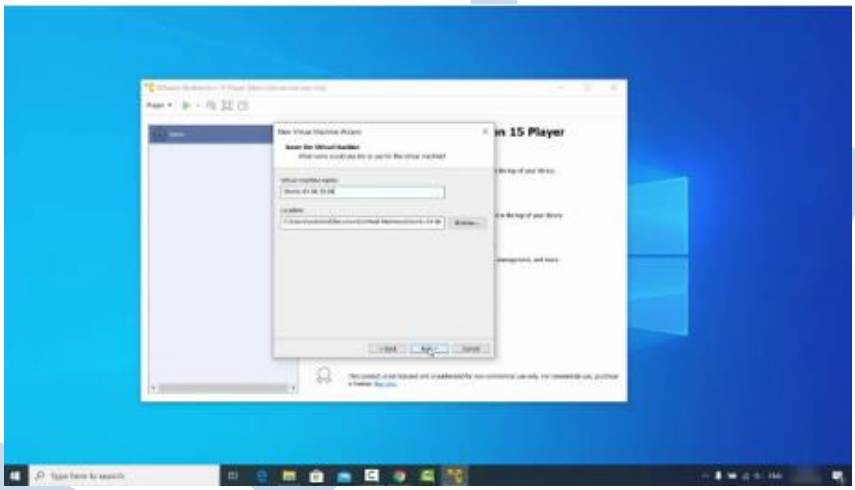
Step 5:



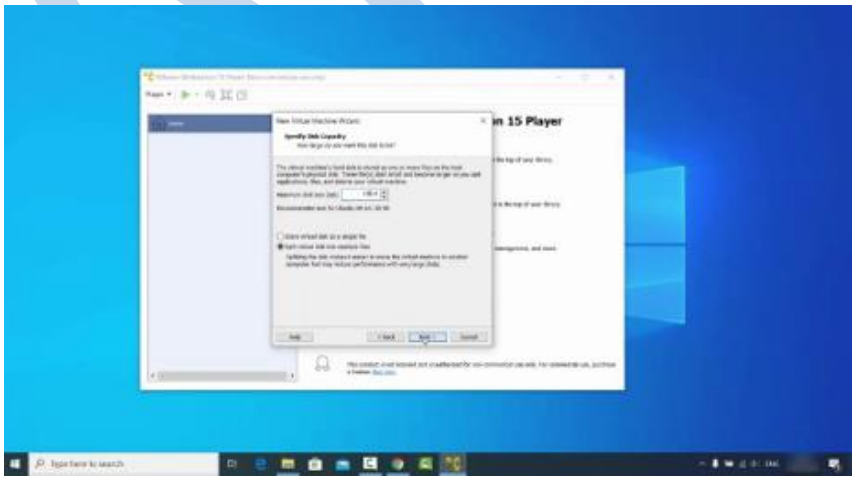
Step 6:



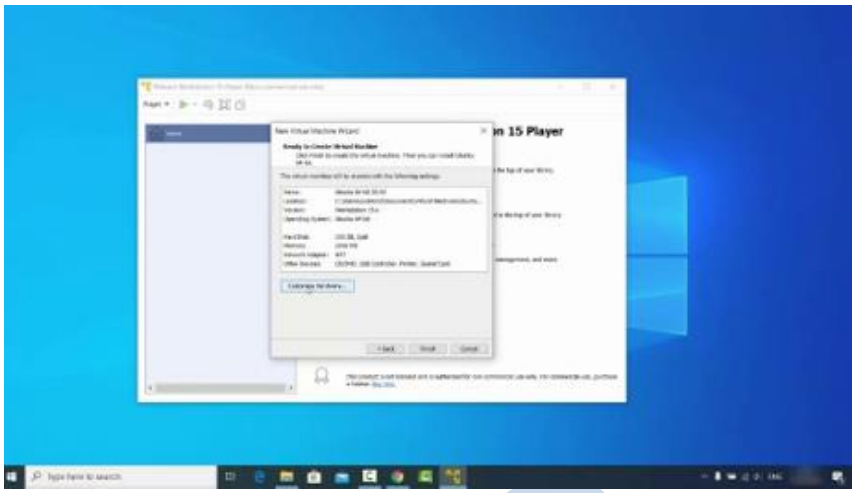
Step 7:



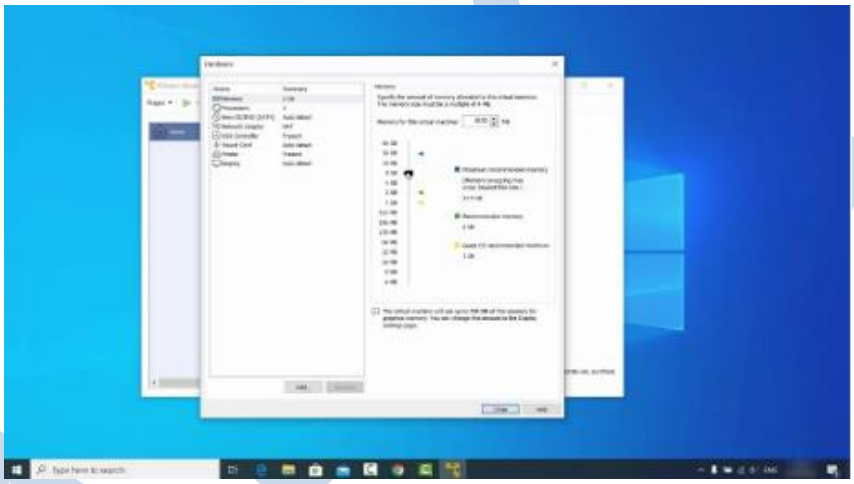
Step 8:



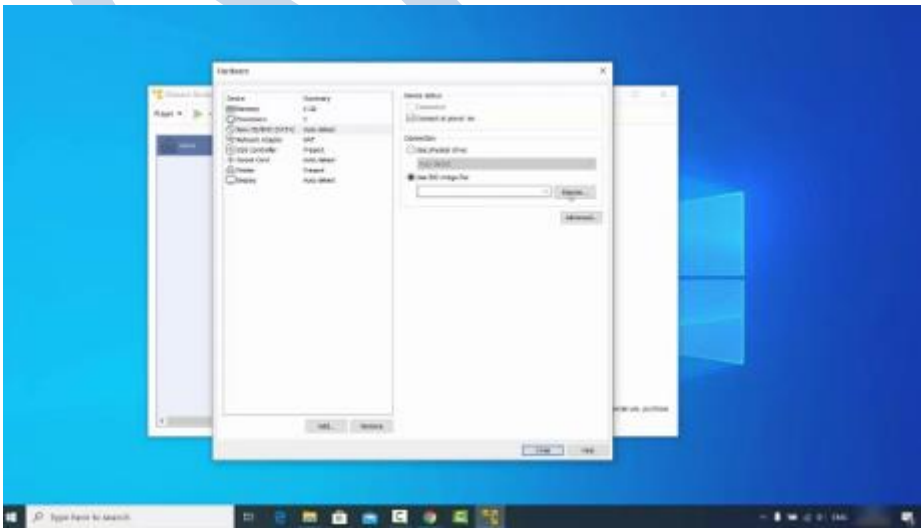
Step 9:



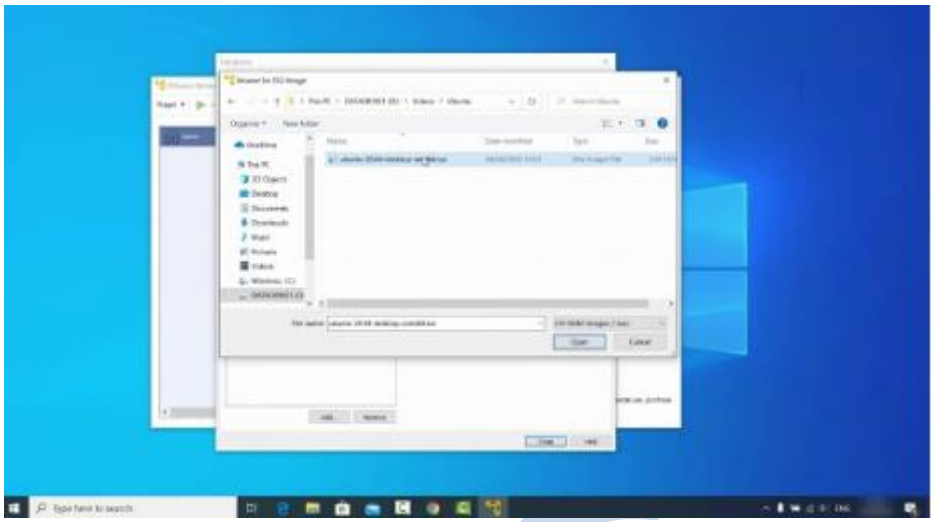
Step 10:



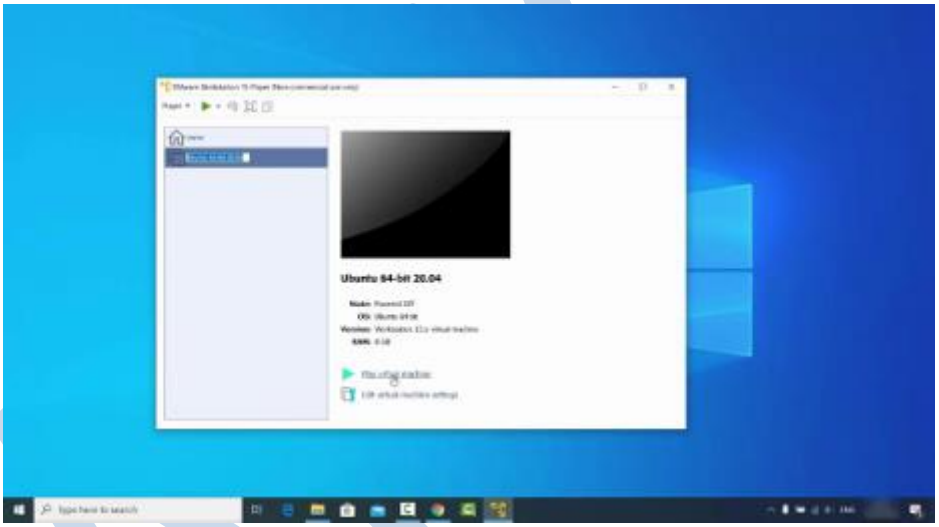
Step 11:



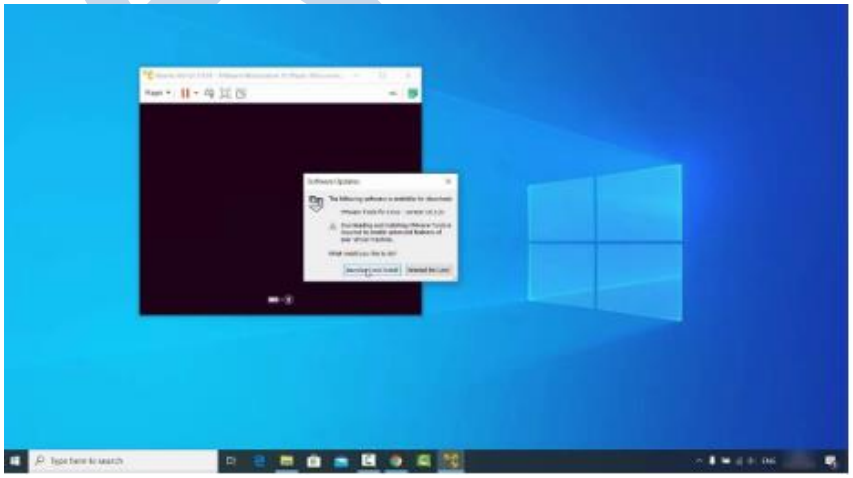
Step 12:



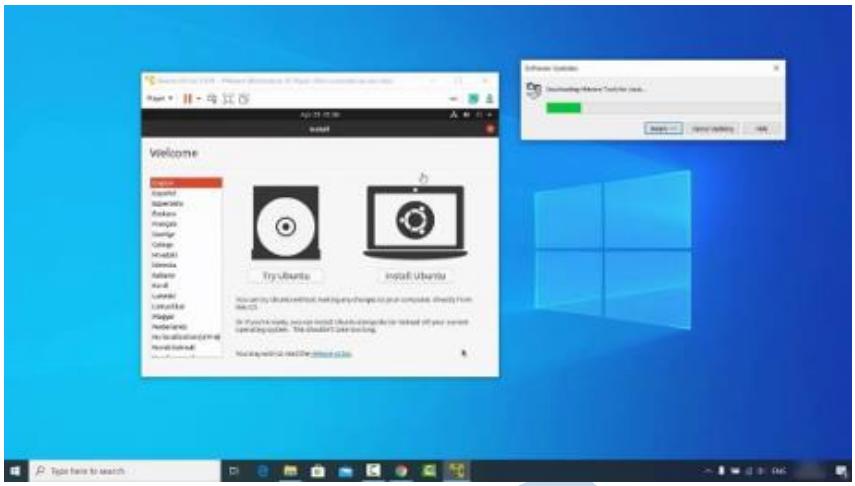
Step 13:



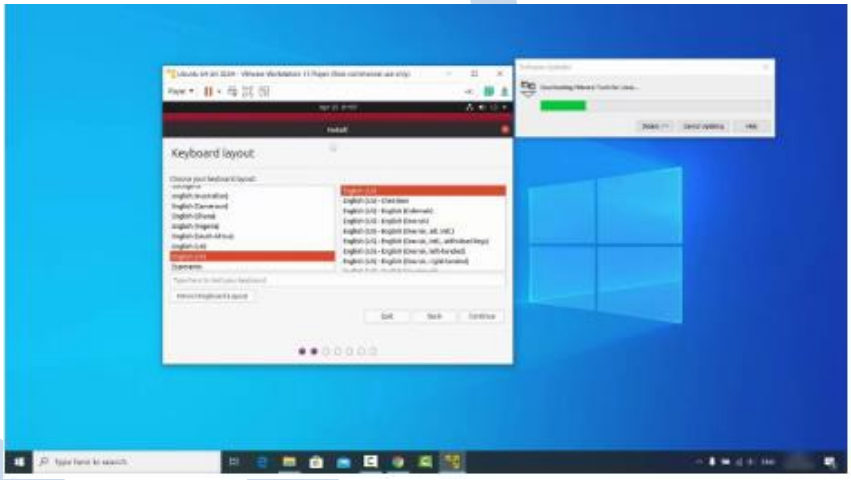
Step 14:



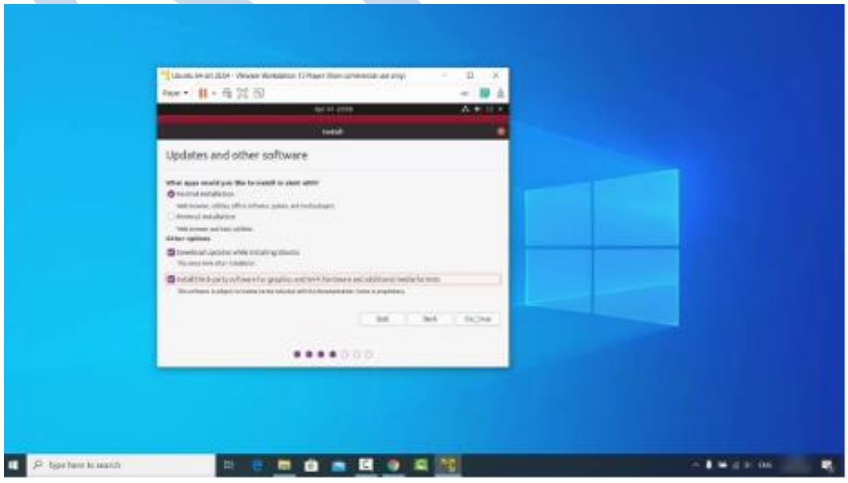
Step 15:



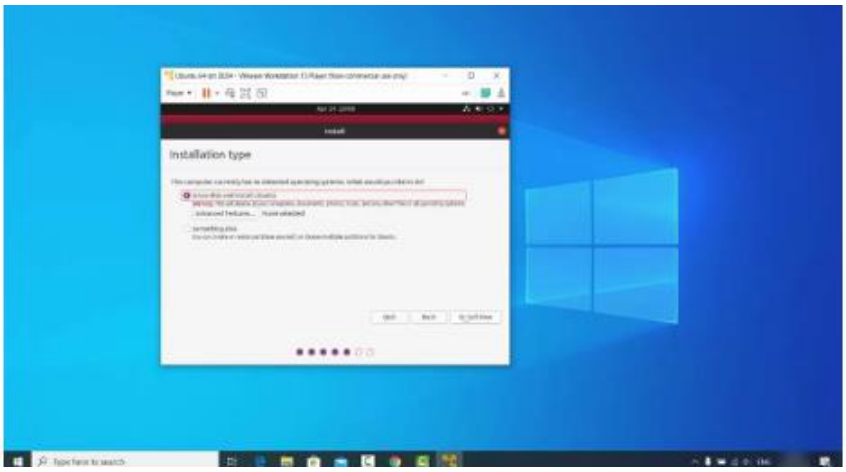
Step 16:



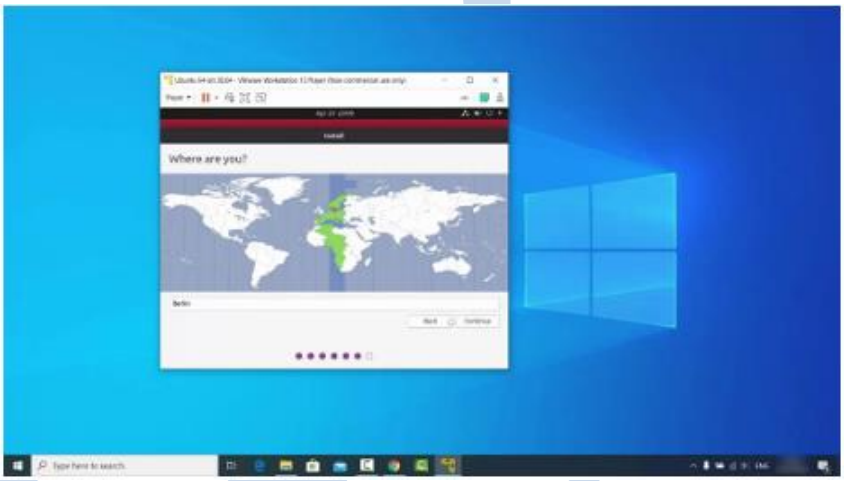
Step 17:



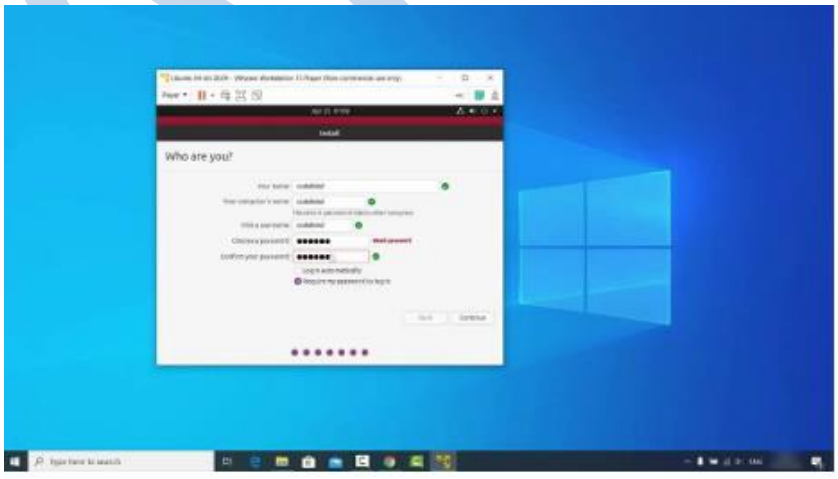
Step 18:



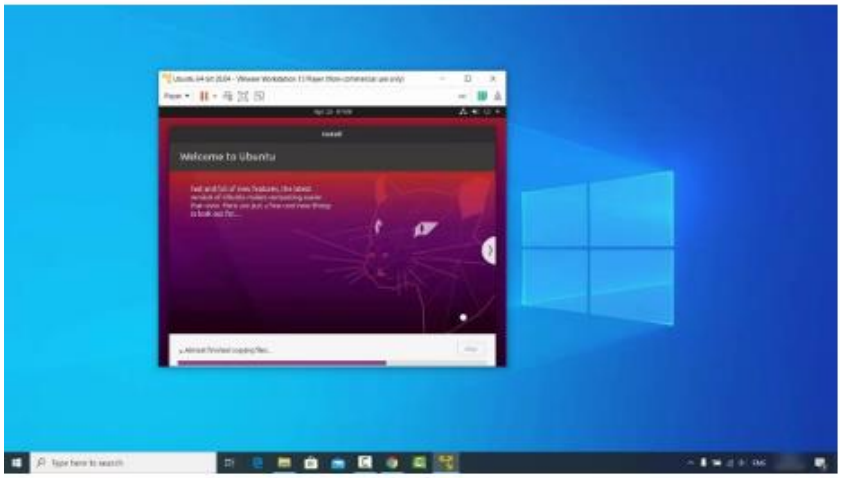
Step 19:



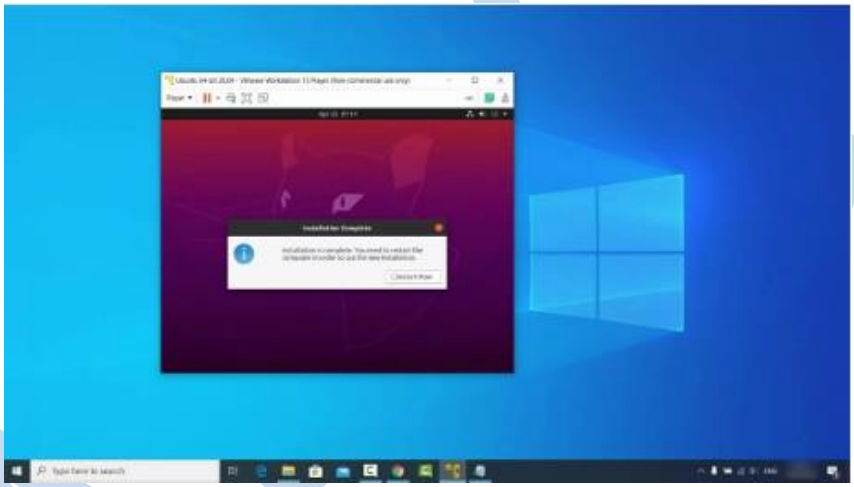
Step 20:



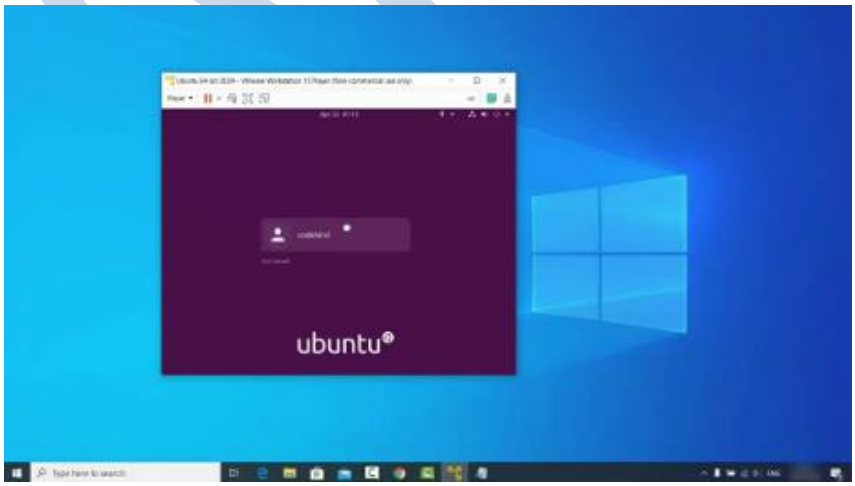
Step 21:



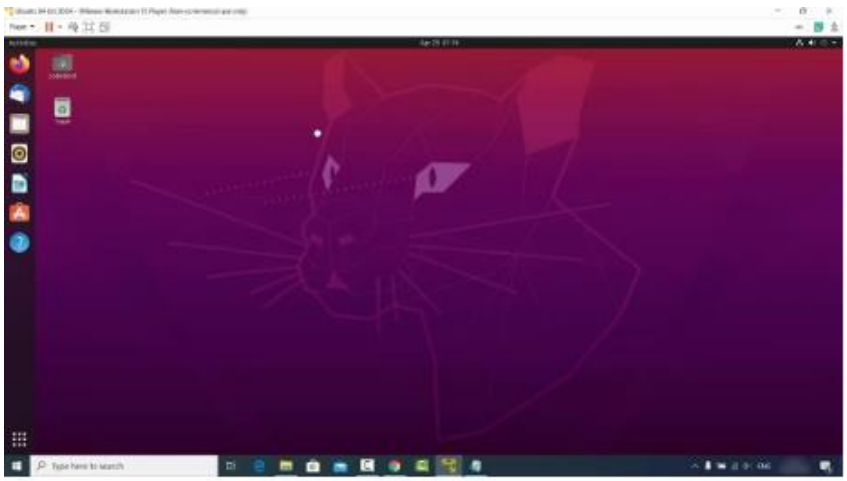
Step 22:



Step 23:



Step 24:



Result:

Thus the Installation of LINUX operating system is completed successfully.