

Ex. : 1(a)

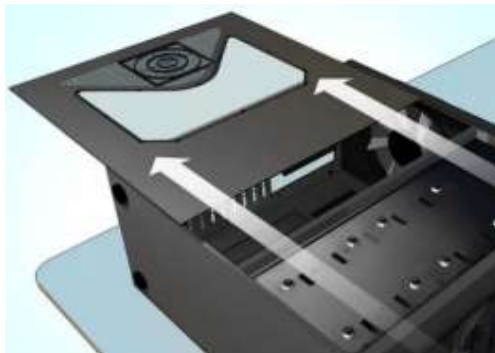
Date :

## ASSEMBLING THE COMPUTER SYSTEM

**Aim: To Assemble the Computer System**

**Steps:**

1. Grounding oneself can be done by using an antistatic wrist-strap cable to prevent electrostatic discharge (ESD) which can be deadly to computer electronics. Alternatively, a large metal body like a radiator can also be touched to discharge oneself.



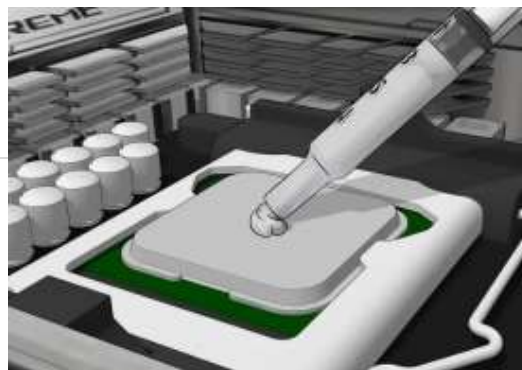
2. To Open the case. Unscrew the side panel (or slide it toward the back of the case) to do so.

3. Install the power supply. Some cases come with the power supply already installed, while others will require to purchase the power supply separately and install it ourself. Have to make sure that the power supply is installed in the correct orientation, and that nothing is blocking the power supply's fan.

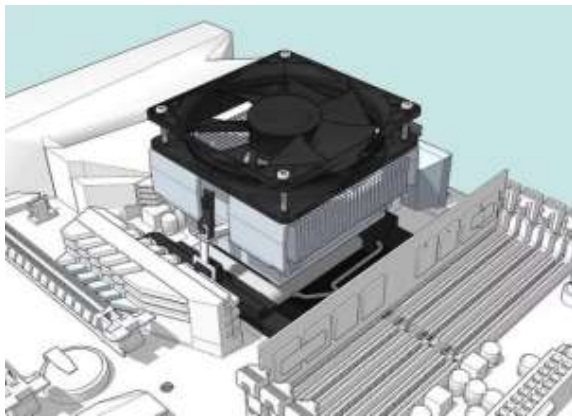
- a. The power supply will usually go near the top or the bottom rear of the case. Can determine where the power supply is supposed to sit by looking for a missing section on the back of the case.



4. Add components to the motherboard. This is usually easiest to do before installing the motherboard, as the case can limit ability to wire components:

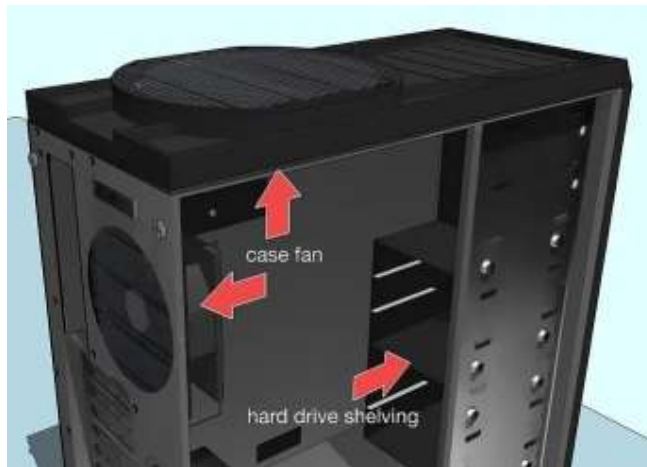


- a. Attach the processor to the motherboard by finding the processor port on the motherboard's surface. An indicator on CPU and motherboard will show the correct orientation.
  - b. Attach RAM to the motherboard by finding the RAM slots and inserting the RAM appropriately.
  - c. Attach power supply to the motherboard's power connectors.
  - d. Locate (but do not attach) the motherboard's hard drive SATA port. Can use this to connect the hard drive to the motherboard later.
5. Apply thermal paste to the processor if necessary. Put a small dot (around the size of a grain of rice or a pea) of thermal paste on the CPU. Adding too much thermal paste will create a mess, such as getting paste into the motherboard socket, which may short circuit components and decrease the motherboard's value if planning to sell it later.

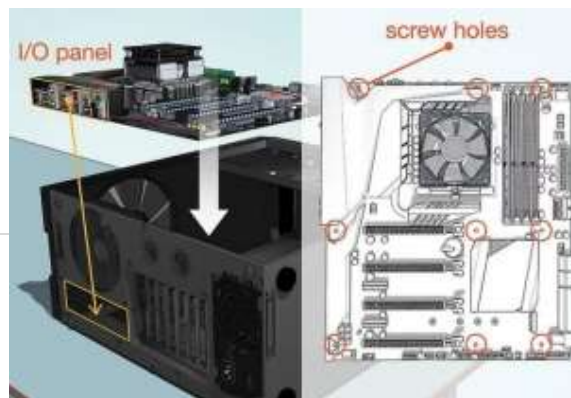


**Tip:** Some processors that come with heat sinks do not need thermal paste because the heat sink already has thermal paste applied by the factory. Check the bottom of the heat sink unit before applying paste to the processor.

6. Attach the heat sink. This varies from heat sink to heat sink, so read the instructions for the processor.
- a. Most stock coolers attach directly over the processor and clip into the motherboard.
  - b. Aftermarket heat sinks may have brackets that need to be attached underneath the motherboard.
  - c. Skip this step if processor has an installed heat sink.



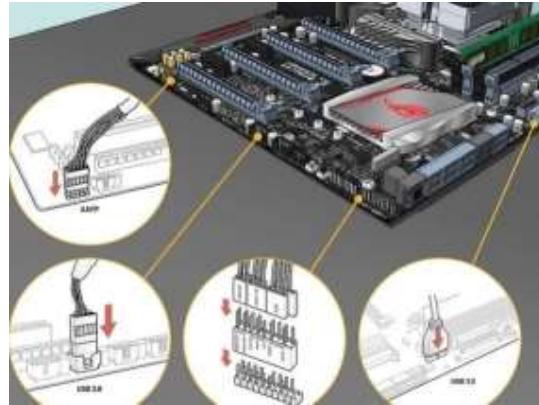
7. Prepare the case needs to knock the plates out of the back of the case in order to fit the components into the correct positions.
- a. If the case has separate shelving units to hold the hard drive, install the units using the included screws.



- b. May need to install and wire the case's fans before installing any components. If so, follow the case's fan installation instructions.

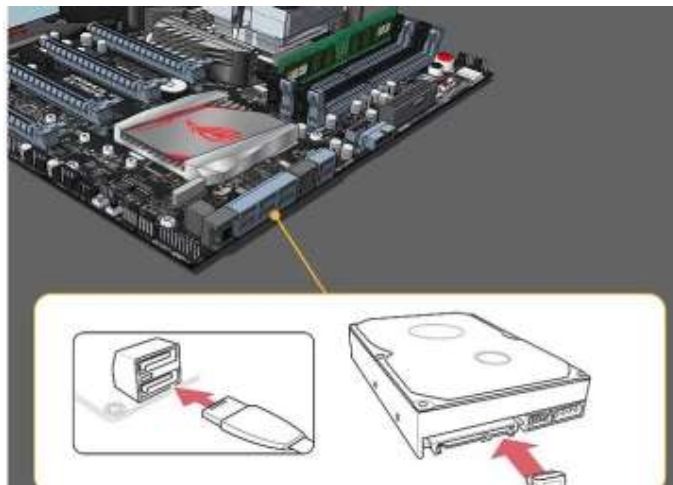
8. Secure the motherboard. Once the standoffs are installed, place the motherboard in the case and push it up against the backplate. All of the back ports should fit into the holes in the I/O backplate.

- a. Use the screws provided to secure the motherboard to the standoffs through the shielded screw holes on the motherboard.



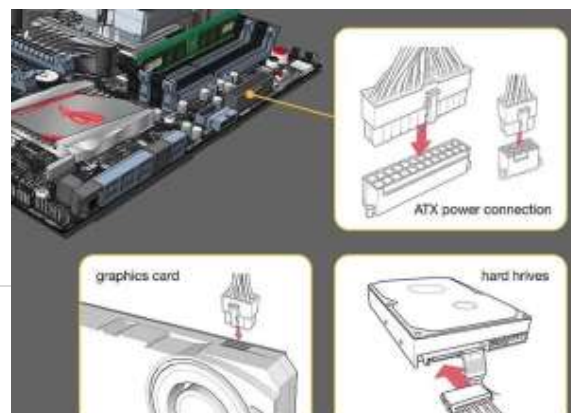
9. Plug in the case connectors. These tend to be located together on the motherboard near the front of the case. The order in which these are connected will depend on which is easiest. Make sure that the USB ports are connected, the Power and Reset switches, the LED power and hard drive lights, and the audiocable. The motherboard's documentation will show where on the motherboard these connectors attach.

- a. There is typically only one way that these connectors can attach to the motherboard. Do not try to force anything to fit.



10. Install the hard drive. This process will vary slightly depending on your case, but should typically go as follows:

- a. Remove any front panels on the case.
- b. Insert the hard drive into its slot (usually near the top of the case).
- c. Tighten any screws needed to hold the drive-in place.
- d. Plug the hard drive's SATA cable into the SATA slot on the motherboard.



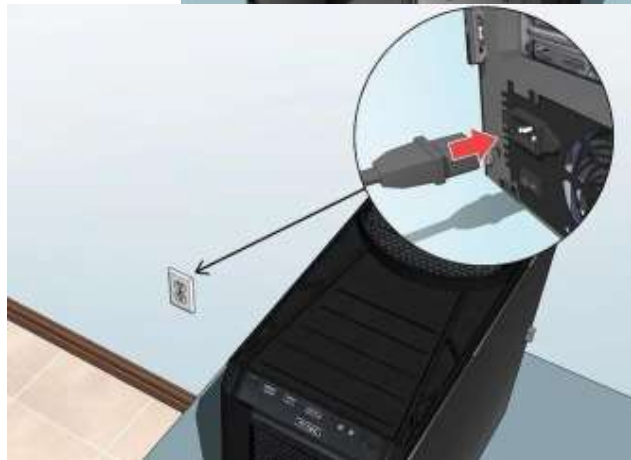
**11. Connect the power supply to any necessary components. If the power supply is not connected to components which need power, make sure that it is connected to the following locations:**

- a. Motherboard
- b. Graphics card(s)
- c. Hard drive(s)



**12. Finish the computer assembly. Once placed and connected the various internal components for the computer, all that's left to do is ensure that none of the wires interfere with circulation and close up the case.**

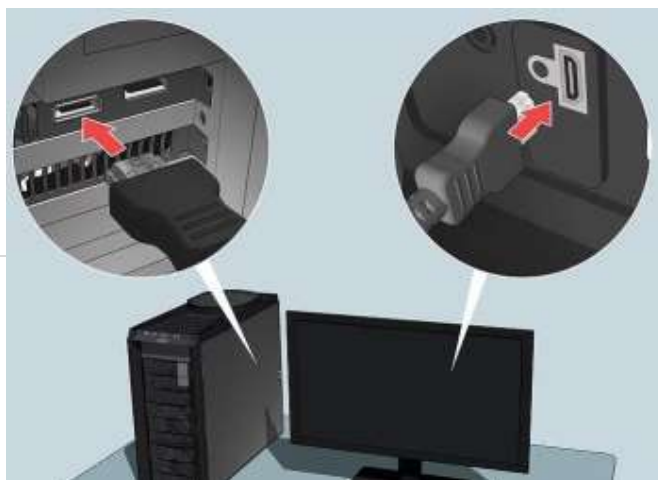
- a. If bought a cooling system, install it before proceeding. Refer to the cooling system's installation instructions in order to do so.
- b. Many cases will have a panel which either slides back into place or screws onto the side of the case.



## **RUNNING THE COMPUTER**

**1. Attach the computer to an outlet. Using the power source's power cable, plug the computer into a wall outlet or power strip.**

- a. May first have to attach the electrical cable to the power source input on the back of the computer's case.



2. **Plug a monitor into the computer.** Typically use the graphics card output that is near the bottom of the case, though some motherboards may have this port on the right or left side of the case.
  - a. The output here is usually a DisplayPort or HDMI port.



3. **Turn on the computer.** Press the computer's Power button on the front or back of the case. If everything's properly connected, the computer should start up.



4. **Install Windows or Linux.** Windows is compatible with all PCs and will make full use of their various features (e.g., Bluetooth), but will have to purchase a copy of Windows if not having a product key. Linux is free but may not be able to use all the computer's hardware.
    - a. If not having an installation USB drive, need to create one on another computer before can install the operating system.
-





- 5. Install the drivers.** Once the operating system is installed, need to install the drivers. Almost all the hardware that was purchased should come with discs that contain the driver software needed for the hardware to work.
- **Modern versions of Windows and Linux will install most drivers automatically when connected to the Internet.**

**Result : The Computer System has been successfully assembled**

Ex: 1 (b)

Date:

## Installing Ubuntu in VMware Player on Windows

**Aim :**

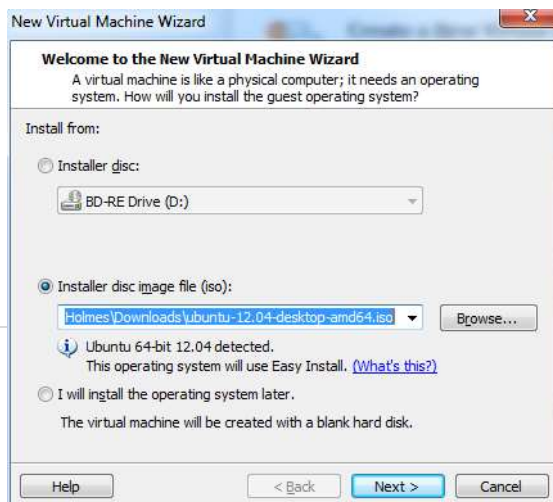
**To Install Ubuntu in VMware Player on Windows**

**Steps :**

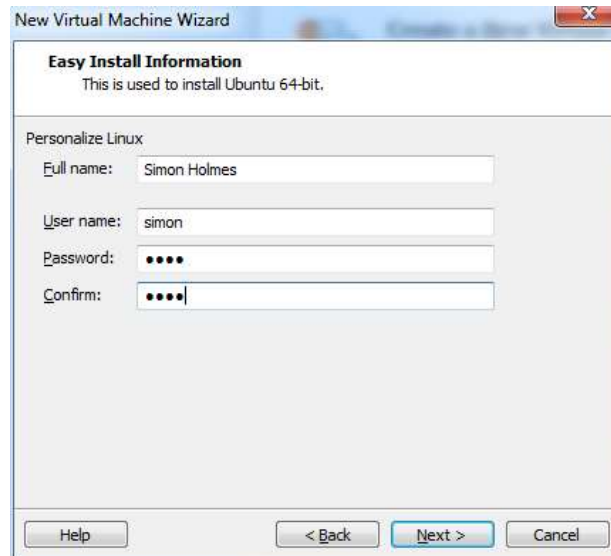
1. Download the Ubuntu iso (desktop not server) and the free VMware Player.
2. Install VMware Player and run it, can see something like this:



3. Select "Create a New Virtual Machine"
4. Select "Installer disc image file" and browse to the Ubuntu iso downloaded. Click next

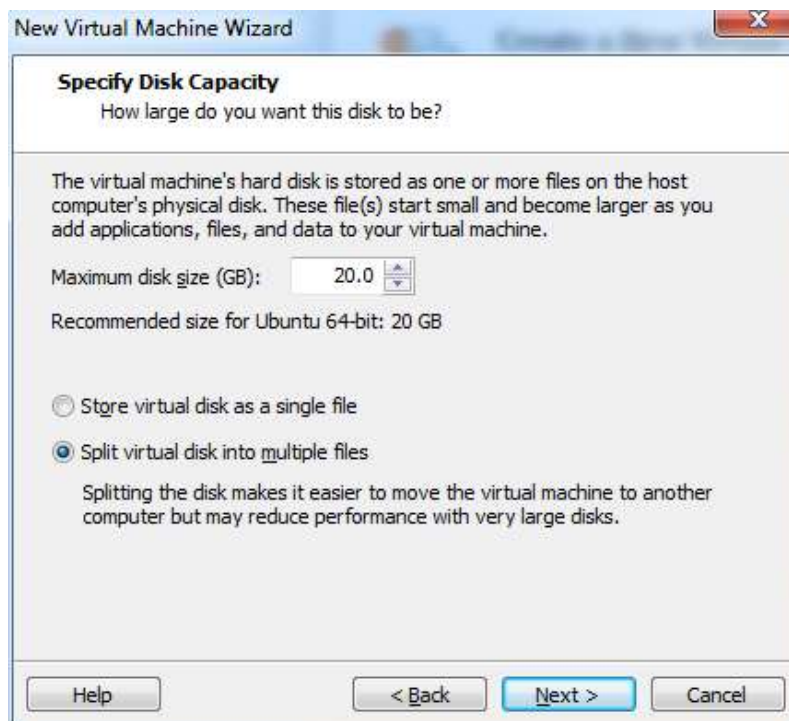


**5. Enter full name, username and password and hit next**



The screenshot shows the 'New Virtual Machine Wizard' window, specifically the 'Easy Install Information' step. The window title is 'New Virtual Machine Wizard'. Below the title bar, it says 'Easy Install Information' and 'This is used to install Ubuntu 64-bit.' The main section is titled 'Personalize Linux' and contains four input fields: 'Full name:' with the value 'Simon Holmes', 'User name:' with the value 'simon', 'Password:' with four dots, and 'Confirm:' with four dots. At the bottom, there are four buttons: 'Help', '< Back', 'Next >', and 'Cancel'.

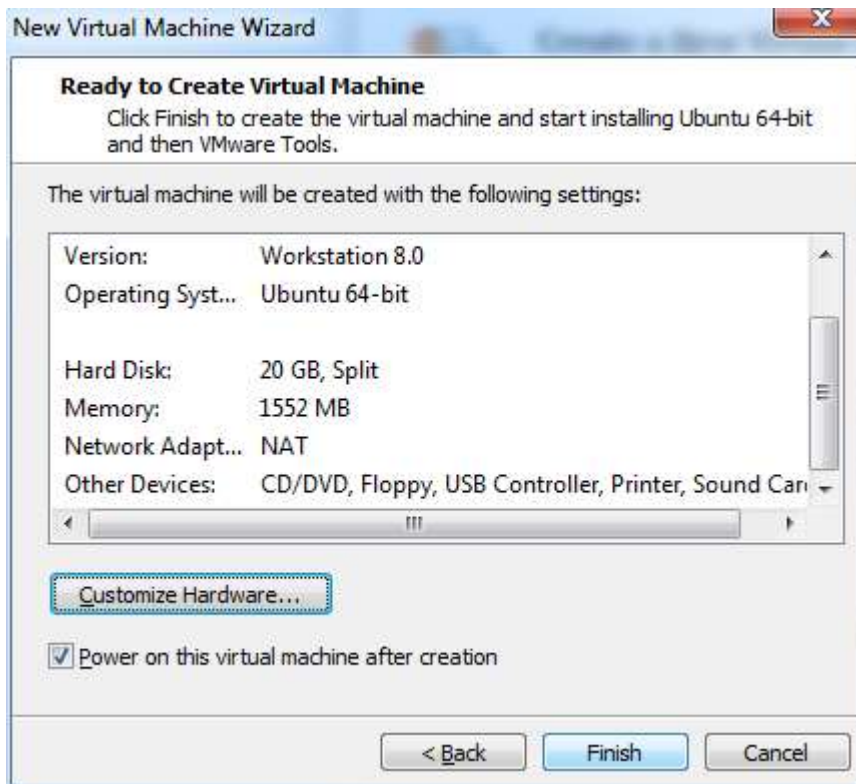
**6. Select the maximum disk size and type. Unless planning on some really CPU intensive work inside the VM, select the “Split virtual disk into multiple files” option. Hit next when comfortable with the settings.**



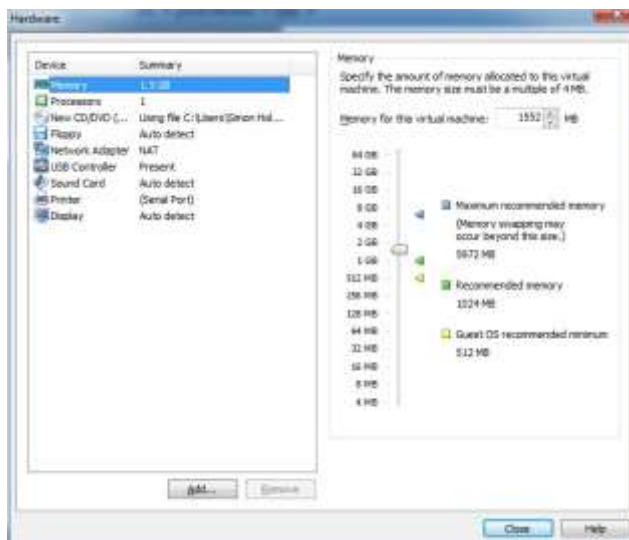
The screenshot shows the 'New Virtual Machine Wizard' window, specifically the 'Specify Disk Capacity' step. The window title is 'New Virtual Machine Wizard'. Below the title bar, it says 'Specify Disk Capacity' and 'How large do you want this disk to be?'. The main section contains a paragraph: 'The virtual machine's hard disk is stored as one or more files on the host computer's physical disk. These file(s) start small and become larger as you add applications, files, and data to your virtual machine.' Below this, there is a 'Maximum disk size (GB):' label followed by a spinner box set to '20.0'. Underneath, it says 'Recommended size for Ubuntu 64-bit: 20 GB'. There are two radio button options: 'Store virtual disk as a single file' (unselected) and 'Split virtual disk into multiple files' (selected). Below these options, a note states: 'Splitting the disk makes it easier to move the virtual machine to another computer but may reduce performance with very large disks.' At the bottom, there are four buttons: 'Help', '< Back', 'Next >', and 'Cancel'.

**7. This brings to the confirmation page. Click “Customize Hardware”**

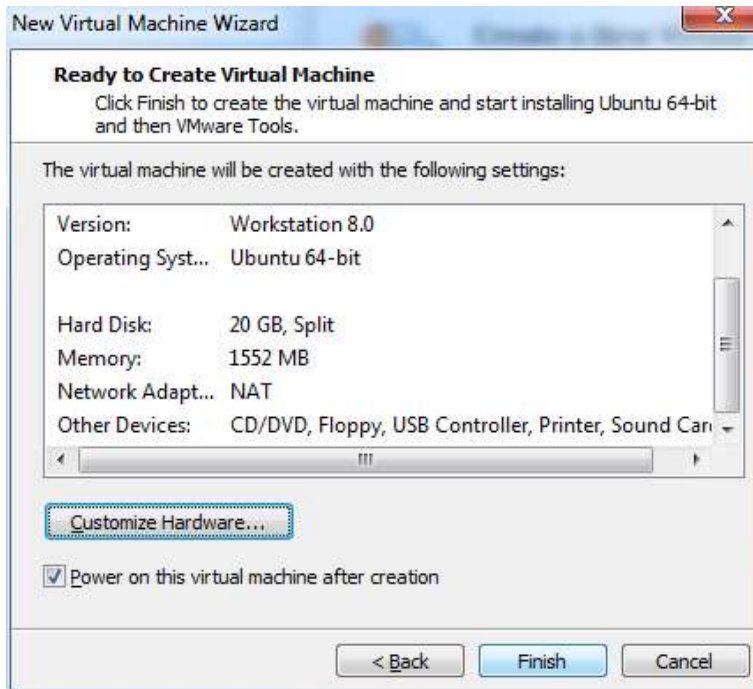




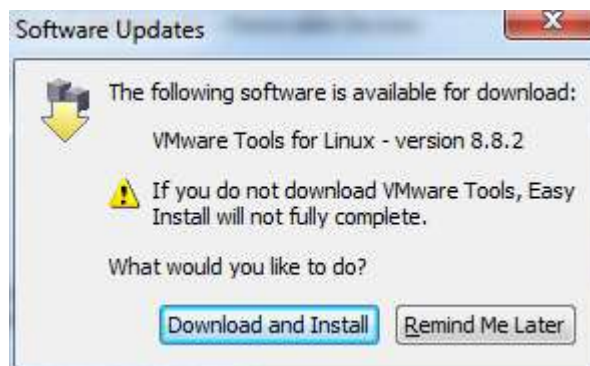
8. In the hardware options section select the amount of memory wanted by the VM to use. Click Close.



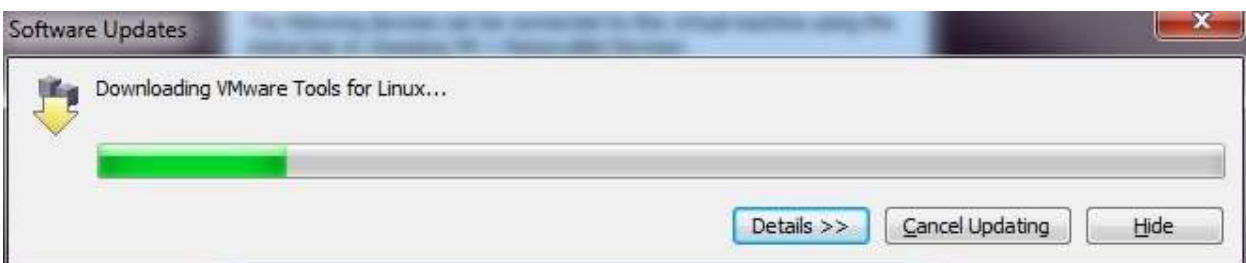
9. This brings back to the confirmation page. Click Finish this time.



**10. Will probably be prompted to download VMware Tools for Linux. Click “Download and Install”**



**11. Wait for it to install**



**12. Ubuntu will then start to install**



**13. When all is done will be presented with the Ubuntu login screen. Enter the password**



**14. Click the**



**clock in the top right to set your time and date settings**

**15. Once it is set that up, system is up and running with Ubuntu in VMware Player on the Windows machine.**

**Result ;**

**Ubuntu in VMware Player on Windows has been installed successfully**

	CPU SCHEDULING ALGORITHMS
Exp.No. 2(A)	IMPLEMENTATION OF FIRST COME FIRST SERVE SCHEDULING
Date	

**Aim:**

To implement the first come first serve scheduling algorithm

**Description:**

**Scheduling Criteria**

- **CPU utilization:** We want to keep the CPU as busy as possible. CPU utilization may range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).
- **Throughput:** If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes completed per time unit, called throughput. For long processes, this rate may be 1 process per hour; for short transactions, throughput might be 10 processes per second.
- **Turnaround time:** The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time:** Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time:** In an interactive system, turnaround time may not be the best criterion. Another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the amount of time it takes to start responding, but not the time that it takes to output that response.

**First-Come, First-Served Scheduling**

The process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. The average waiting time under the FCFS policy, however, is often quite long.

Example: Process Burst Time

P1	24
P2	3
P3	3

If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following

**Gantt chart:**



**Algorithm:**

1. Start the process
2. Get the number of processes to be inserted
3. Get the value for burst time of each process from the user
4. Having allocated the burst time(bt) for individual processes , Start with the first process from its initial position let other process to be in queue
5. Calculate the waiting time(wt) and turnaround time(tat) as
6.  $Wt(p_i) = wt(p_{i-1}) + tat(p_{i-1})$  (i.e. wt of current process = wt of previous process + tat of previous process)
7.  $tat(p_i) = wt(p_i) + bt(p_i)$  (i.e. tat of current process = wt of current process + bt of current process)
8. Calculate the total and average waiting time and turnaround time
9. Display the values
10. Stop the process

**Program:**

```
#include<stdio.h>
int main()
{
    int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp; float
        avg_wt,avg_tat;
    printf("Enter number of process:");
    scanf("%d",&n);
    printf("\nEnter Burst Time:\n");
    for(i=0;i<n;i++)
    {
        printf("p % d:",i+1);
        scanf("%d",&bt[i]);
        p[i]=i+1;          //contains process number
    }
    wt[0]=0;              //waiting time for first process will be zero
    //calculate waiting time
    for(i=1;i<n;i++)
    {
        wt[i]=0;
        for(j=0;j<i;j++)
            wt[i]+=bt[j];

        total+=wt[i];
    }
    avg_wt=(float)total/n;    //average waiting time
    total=0;
    printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time");
    for(i=0;i<n;i++)
    {
        tat[i]=bt[i]+wt[i];    //calculate turnaround time
        total+=tat[i];
        printf("\np%d\t\t %d\t\t\t %d\t\t\t %d",p[i],bt[i],wt[i],tat[i]);
    }

    avg_tat=(float)total/n;    //average turnaround time
    printf("\n\nAverage Waiting Time=%f",avg_wt);
    printf("\nAverage Turnaround Time=%f\n",avg_tat);
}
```



### **Output:**

```
[188r1a0501@localhost ~]$ vi w1.c
[188r1a0501@localhost ~]$ gcc w1.c
[188r1a0501@localhost ~]$ ./a.out
Enter number of process:3

Enter Burst Time:
p1:3
p2:4
p3:2

Process      Burst Time      Waiting Time      Turnaround Time
p1            3                0                  3
p2            4                3                  7
p3            2                7                  9

Average Waiting Time=3.33333
Average Turnaround Time=6.33333
[188r1a0501@localhost ~]$
```

### **Result:**

Thus, the first come first serve scheduling algorithm is implemented successfully.

Exp.No. 2(B)

## CPU SCHEDULING ALGORITHMS

Date

## IMPLEMENTATION OF SHORTEST JOB FIRST SCHEDULING

**Aim:**

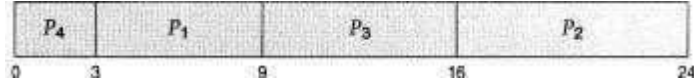
To implement the Shortest Job First scheduling algorithm

**Description:**

This algorithm associates with each process the length of the latter's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If two processes have the same length next CPU burst, FCFS scheduling is used to break the tie.

**Example**

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

**Gantt Chart**

Average waiting time:

$$P1 : 10 - 1 = 9$$

$$P2 : 1 - 1 = 0$$

$$P3 : 17 - 2 = 15$$

$$P4 : 5 - 3 = 2$$

$$AWT = (9+0+15+2) / 4 = 6.5 \text{ ms}$$

**Algorithm :**

1. Start the process
2. Get the number of processes to be inserted
3. Sort the processes according to the burst time and allocate the one with shortest burst to execute first
4. If two process have same burst length then FCFS scheduling algorithm is used
5. Calculate the total and average waiting time and turn around time
6. Display the values
7. Stop the process

**Program:**

```
#include<stdio.h>

int main()
{
    int bt[20],p[20],wt[20],tat[20],i,j,n,total=0,pos,temp; float
        avg_wt,avg_tat;
    printf("Enter number of process:");
    scanf("%d",&n);

    printf("\nEnter Burst Time:\n");
    for(i=0;i<n;i++)
    {
        printf("p%d:",i+1);
        scanf("%d",&bt[i]);
        p[i]=i+1;          //contains process number
    }
    //sorting burst time in ascending order using selection sort
    for(i=0;i<n;i++)
    {
        pos=i;
        for(j=i+1;j<n;j++)
        {
            if(bt[j]<bt[pos])
                pos=j;
        }
        temp=bt[i];
        bt[i]=bt[pos];
        bt[pos]=temp;

        temp=p[i];
        p[i]=p[pos];
        p[pos]=temp;
    }
    wt[0]=0;          //waiting time for first process will be zero

    //calculate waiting time
    for(i=1;i<n;i++)
    {
        wt[i]=0;
```

```

for(j=0;j<i;j++)
wt[i]+=bt[j];
    total+=wt[i];
}

avg_wt=(float)total/n;        //average waiting time
    total=0;

    printf("\nProcess\t Burst Time \tWaiting Time\tTurnaround Time");
for(i=0;i<n;i++)
{
    tat[i]=bt[i]+wt[i];        //calculate turnaround time
    total+=tat[i];
    printf("\np%d\t\t %d\t\t\t %d\t\t\t\t %d",p[i],bt[i],wt[i],tat[i]);
}

avg_tat=(float)total/n;        //average turnaround time
    printf("\n\nAverage Waiting Time=%f",avg_wt);
    printf("\n\nAverage Turnaround Time=%f\n",avg_tat);
}

```

### Output:

```
[188r1a0501@localhost ~]$ vi w1a.c
[188r1a0501@localhost ~]$ gcc w1a.c
[188r1a0501@localhost ~]$ ./a.out
Enter number of process:3

Enter Burst Time:
p1:5
p2:3
p3:7

Process      Burst Time      Waiting Time      Turnaround Time
p2           3              0                3
p1           5              3                8
p3           7              8               15

Average Waiting Time=3.666667
Average Turnaround Time=8.666667
[188r1a0501@localhost ~]$ █
```

### Result:

Thus, the Shortest Job First scheduling algorithm is implemented successfully.



Exp.No. 2(C)

## CPU SCHEDULING ALGORITHMS

Date

## IMPLEMENTATION OF PRIORITY SCHEDULING

**Aim:**

To implement priority scheduling algorithm

**Description:**

The SJF algorithm is a special case of the general priority-scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority.

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

**Gantt chart:**

AWT = 8.2 ms

It can be either preemptive or non-preemptive. Problem with priority scheduling algorithms is indefinite blocking or starvation. A solution to the problem of indefinite blockage of low priority process is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 0 (low) to 127 (high), we could increment the priority of a waiting process by 1 every 15 mins.

**Algorithm:**

1. Start the process
2. Get the number of processes to be inserted
3. Get the corresponding priority of processes
4. Sort the processes according to the priority and allocate the one with highest priority to execute first
5. If two process have same priority then FCFS scheduling algorithm is used
6. Calculate the total and average waiting time and turnaround time
7. Display the values
8. Stop the process

**Program:**

```
#include<stdio.h>

int main()
{
    int bt[20],p[20],wt[20],tat[20],pri[20],i,j,k,n,total=0,pos,temp; float
        avg_wt,avg_tat;
    printf("Enter number of process:");
    scanf("%d",&n);

    printf("\nEnter Burst Time:\n");
    for(i=0;i<n;i++)
    {
        printf("p%d:",i+1);
        scanf("%d",&bt[i]);
        p[i]=i+1;          //contains process number
    }
    printf(" enter priority of the process ");
    for(i=0;i<n;i++)
    {
        p[i] = i;
        //printf("Priority of Process");
        printf("p%d ",i+1);
        scanf("%d",&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;
            }
}
```

```

wt[0]=0;           //waiting time for first process will be zero

//calculate waiting time
for(i=1;i<n;i++)
{
    wt[i]=0;
    for(j=0;j<i;j++)
        wt[i]+=bt[j];

    total+=wt[i];
}
avg_wt=(float)total/n;    //average waiting time
total=0;

printf("\nProcess\t      Burst Time \tPriority \tWaiting Time\tTurnaround Time");
for(i=0;i<n;i++)
{
    tat[i]=bt[i]+wt[i];    //calculate turnaround time
    total+=tat[i];
    printf("\np%d\t\t %d\t\t %d\t\t %d\t\t %d",p[i],bt[i],pri[i],wt[i],tat[i]);
}

avg_tat=(float)total/n;    //average turnaround time
printf("\n\nAverage Waiting Time=%f",avg_wt);
printf("\n\nAverage Turnaround Time=%f\n",avg_tat);
}

```

### Output:

```
[188r1a0501@localhost ~]$ vi wlb.c
[188r1a0501@localhost ~]$ gcc wlb.c
[188r1a0501@localhost ~]$ ./a.out
Enter number of process:3

Enter Burst Time:
p1:5
p2:6
p3:7
Enter priority of the process p1 1
p2 3
p3 2

Process      Burst Time      Priority      Waiting Time      Turnaround Time
p0            5                1              0                  5
p2            7                2              5                 12
p1            6                3             12                 18

Average Waiting Time=5.666667
Average Turnaround Time=11.666667
```

### Result:

Thus, priority scheduling algorithm is implemented successfully.

Exp.No. 2(D)	CPU SCHEDULING ALGORITHMS
Date	IMPLEMENTATION OF ROUND-ROBIN SCHEDULING

### Aim:

To implement round robin scheduling algorithm

### Description:

The round-robin (RR) scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a time quantum (or time slice), is defined. The ready queue is treated as a circular queue.

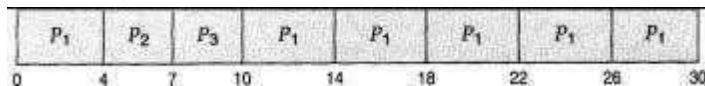
Process Burst Time **Quantum Time = 4 ms**

P1            24

P2            3

P3            3

### Gantt chart



The average waiting time is  $17/3 = 5.66$  milliseconds.

Waiting time for P1 =  $26 - 20 = 6$  P2 = 4 P3 = 7 ( $6+4+7 / 3 = 5.66$  ms)

The performance of the RR algorithm depends heavily on the size of the time– quantum. If time-quantum is very large (infinite) then RR policy is same as FCFS policy. If time quantum is very small, RR approach is called processor sharing and appears to the users as though each of n process has its own processor running at  $1/n$  the speed of real processor.

### Algorithm:

1. Start the process
2. Get the number of elements to be inserted
3. Get the value for burst time for individual processes
4. Get the value for time quantum
5. Make the CPU scheduler go around the ready queue allocating CPU to each process for the time interval specified
6. Make the CPU scheduler pick the first process and set time to interrupt after quantum. And after it's expiry dispatch the process
7. If the process has burst time less than the time quantum then the process is released by the CPU
8. If the process has burst time greater than time quantum then it is interrupted by the OS and the process is put to the tail of ready queue and the schedule selects next process from head of the queue
9. Calculate the total and average waiting time and turnaround time
10. Display the results

**Program:**

```
#include<stdio.h>

main()
{
    int st[10],bt[10],wt[10],tat[10],n,tq; int
        i,count=0,swt=0,stat=0,temp,sq=0;
    float awt,atat;
    printf("enter the number of processes");
    scanf("%d",&n);
    printf("enter the burst time of each process /n");
    for(i=0;i<n;i++)
    {
        printf(("p%d",i+1);
        scanf("%d",&bt[i]);
        st[i]=bt[i];
    }
    printf("enter the time quantum");
    scanf("%d",&tq);
    while(1)
    {
        for(i=0,count=0;i<n;i++)
        {
            temp=tq;
            if(st[i]==0)
            {
                count++;
                continue;
            }
            if(st[i]>tq)
                st[i]=st[i]-tq;
            else
                if(st[i]>=0)
```



```

        {
            temp=st[i];
            st[i]=0;
        }
        sq=sq+temp;
        tat[i]=sq;
    }
    if(n==count)
        break;
}
for(i=0;i<n;i++)
{
    wt[i]=tat[i]-bt[i];
    swt=swt+wt[i];
    stat=stat+tat[i];
}
awt=(float)swt/n;
atat=(float)stat/n;
printf("process no\t burst time\t waiting time\t turnaround time\n");
for(i=0;i<n;i++)
printf("%d\t %d\t %d\t %d\n",i+1,bt[i],wt[i],tat[i]); printf("avg
wt time=%f,avg turn around time=%f",awt,atat);
}

```

### **Output:**

```
[188r1a0501@localhost ~]$ vi wld.c
[188r1a0501@localhost ~]$ gcc wld.c
[188r1a0501@localhost ~]$ ./a.out
enter the number of processes 3
enter the burst time of each process
p1 7
p2 2
p3 8
enter the time quantum 2
process no      burst time      waiting time      turnaround time
1                7                8                15
2                2                2                4
3                8                9                17
avg wt time=6.333333,avg turn around time=12.000000[188r1a0501@localhost ~]$
```

### **Result:**

Thus, round robin scheduling algorithm implemented successfully.

<b>EX.no:3a)</b>	<b><u>INTER PROCESS COMMUNICATIONS</u></b>
<b>DATE:</b>	

**Aim:** write C programs to illustrate IPC using pipes mechanisms

**Algorithm: IPC using pipes**

1. Create a child process using fork()
2. Create a simple pipe with C, we make use of the pipe() syscall.
3. Create two file descriptor fd[0] is set up for reading, fd[1] is set up for writing
4. Close the read end of parent process using close() and perform write operation
5. Close the write end of child process and perform reading
6. Display the text.

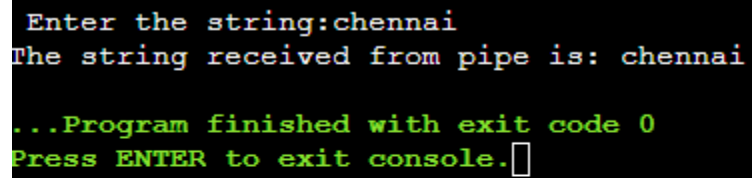
**PROGRAM:**

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

{
    int fd[2], child; char a[10];
    printf("\n Enter the string:");
    scanf("%s", a);
    pipe(fd);
    child=fork();
    if(!child)
    {
        close(fd[0]);
        write(fd[1], a, 5); wait(0);
    }
    else
    {
        close(fd[1]);
```

```
read(fd[0],a,5);  
printf("The string received from pipe is: %s",a);  
}  
return 0;  
}
```

### **OUTPUT:**



```
Enter the string:chennai  
The string received from pipe is: chennai  
  
...Program finished with exit code 0  
Press ENTER to exit console.□
```

**Result:** Thus, IPC using pipes mechanisms is illustrated using c program successfully.

<b>EX.no:3b)</b>	<p style="text-align: center;"><b><u>SYSTEM CALLS</u></b></p> <p style="text-align: center;"><b><u>(READ &amp; WRITE, CREATE &amp;FORK, OPEN</u></b></p> <p style="text-align: center;"><b><u>&amp;CLOSE)</u></b></p>
<b>DATE:</b>	

**Aim:** C program using open, read, write, close , create , fork() system calls.

**Theory:**

There are 5 basic system calls that Unix provides for file I/O.

1. **Create:** Used to Create a new empty file

**Syntax:** int creat(char \*filename, mode\_t mode)

filename : name of the file which you want to create

mode : indicates permissions of new file.

2. **open:** Used to Open the file for reading, writing or both.

**Syntax:** int open(char \*path, int flags [ , int mode ] );

Path : path to file which you want to use

flags : How you like to use

O\_RDONLY: read only, O\_WRONLY: write only, O\_RDWR: read and write, O\_CREAT: create file if it doesn't exist, O\_EXCL: prevent creation if it already exists

3. **close:** Tells the operating system you are done with a file descriptor and Close the file which pointed by fd.

**Syntax:** int close(int fd); fd :file descriptor

4. **read:** From the file indicated by the file descriptor fd, the read() function reads cnt bytes of input into the memory area indicated by buf. A successful read() updates the access time for the file.

**Syntax:** int read(int fd, char \*buf, int size);

fd: file descriptor

buf: buffer to read data from

cnt: length of buffer

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

**Syntax:** int write(int fd, char \*buf, int size); fd: file descriptor

buf: buffer to write data to

cnt: length of buffer

**\*File descriptor** is integer that uniquely identifies an open file of the process.

**Algorithm:**

1. Start the program.
2. Open a file for O\_RDWR for R/W, O\_CREAT for creating a file, O\_TRUNC for truncate a file.
3. Using getchar(), read the character and stored in the string[] array.
4. The string [] array is write into a file close it.
5. Then the first is opened for read only mode and read the characters and displayed it and close the file.
6. Use Fork().
7. Stop the program.

**Program:**

```
#include<sys/stat.h>
#include<stdio.h>
#include<fcntl.h>
#include<sys/types.h>
int main()
{
    int n,i=0;
    int f1,f2;
    char c, strin[100];
    f1=open("data",O_RDWR|O_CREAT|O_TRUNC);
    while((c=getchar())!='\n')
    {
        strin[i++]=c;
    }
    strin[i]='\0';
    write(f1,strin,i);
    close(f1);
    f2=open("data",O_RDONLY);
    read(f2,strin,0);
    printf("\n%s\n",strin);
    close(f2);
    fork();
    return 0;
}
```



**Output:**

```
hai
hai

...Program finished with exit code 0
Press ENTER to exit console. □
```

**RESULT:**

Thus, open, read, write, close , create , fork() system calls implemented successfully using c program.

<b>EX.no:</b> <b>3(C)</b>	<b><u>IMPLEMENT BANKERS' ALGORITHM FOR DEAD LOCK AVOIDANCE</u></b>
<b>DATE:</b>	

**AIM:**

To write a C program to implement Bankers Algorithm to avoid Deadlock.

**PROBLEM DESCRIPTION:**

There are possibilities of side effects of preventing deadlocks and low device utilization reduced system throughput. When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system.

When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If will, the resource are allocated, otherwise the process must wait until some other process releases enough resources.

**ALGORITHM:**

- 1:** Get the no of processes.
- 2:** Get the process numbers.
- 3:** Get the no of resources types and instances of it.
- 4:** Get Max demand of each process of  $n \times m$  matrices.
- 5:** Get the  $n \times m$  matrices the number of resources of each type currently allocated to each process.
- 6:** Calculate the  $n \times m$  of the remaining resource need of each process.
- 7:** Initialize work as available resource and array of finish to false.
- 8:** Check the Needed resource is lesser than the available resource if not display the System not in safe state and if it is lesser than system in safe state.
- 9:** Initialize work as sum of work and allocation, check if array of finish is true go to step 7 again if not go to step 8.
- 10:** Check that request can be immediately granted.
- 11:** If single request is lesser than or equal to available if true means arrive to new state.
- 12:** Print the sequence if it is in safe state or print not in safe state.

**Program:**

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

```
int main ()
{
    int allocated[15][15], max[15][15], need[15][15],
        avail[15], tres[15], work[15], flag[15];
    int pno, rno, i, j, prc,
        count, t, total; count =
        0;
    //clrscr ();

    printf ("\n Enter number of
    process:");
    scanf ("%d", &pno);
    printf ("\n Enter number
    of resources:");
    scanf ("%d", &rno);
    for (i = 1; i <= pno; i++)
    {
        flag[i] = 0;
    }
    printf ("\n Enter total numbers of
    each resources:");
    for (i = 1; i <= rno; i++)
        scanf ("%d", &tres[i]);

    printf ("\n Enter Max resources for
    each process:");
    for (i = 1; i <= pno; i++)
    {

printf ("\n for process %d:",
i);
for (j = 1; j <= rno; j++)
    scanf ("%d", &max[i][j]);
    }
```

```
printf ("\n Enter allocated resources for each  
process:"); for (i = 1; i <= pno; i++)  
{  
printf ("\n for process %d:",  
i); for (j = 1; j <= rno; j++)  
scanf ("%d", &allocated[i][j]);
```

```
}  
printf ("\n available  
resources:\n");  
for (j = 1; j <= rno; j++)  
{  
    avail[j] = 0;  
    total = 0;  
    for (i = 1; i <= pno; i++)  
    {  
        total += allocated[i][j];  
    }  
    avail[j] =  
    tres[j] -  
    total;  
    work[j] =  
    avail[j];  
    printf ("    %d \t", work[j]);  
}
```

```
do  
{  
  
    for (i = 1; i <= pno; i++)  
    {  
        for (j = 1; j <= rno; j++)  
        {  
            need[i][j] = max[i][j] - allocated[i][j];  
        }  
    }  
}
```

```

printf("\n Allocated matrix Max need");
for (i = 1; i <= pno; i++)
    {
        printf ("\n");
        for (j = 1; j <= rno; j++)
            {
                printf ("%4d", allocated[i][j]);
            }
        printf ("|");
        for (j = 1; j <= rno; j++)
            {
                printf ("%4d", max[i][j]);
            }
        printf ("|");
        for (j = 1; j <= rno; j++)
            {
                printf ("%4d", need[i][j]);
            }
    }

    prc = 0;

for (i = 1; i <= pno; i++){
    if (flag[i] == 0){
        prc = i;

        for (j = 1; j <= rno; j++)
            {
                if (work[j] < need[i][j])
                    {
                        prc = 0; break;
                    }
            }
    }

    if (prc != 0)
        break;
}

```

```

if (prc != 0){
    printf ("\n Process %d completed",
i);
    count++;
    printf ("\n Available
matrix:")
    for (j = 1; j <= rno; j++)
        {
            work[j] +=
            allocated[prc][
            j];
            allocated[prc][
            j] = 0;
            max[prc][j] = 0;
            flag[prc] = 1;
            printf (" %d", work[j]);
        }
    }

while (count != pno && prc != 0);

    if (count == pno)
        printf ("\nThe system is in a safe
state!!");
    else
        printf ("\nThe system is in an unsafe
state!!");
return 0;

}

```

### **Output:**

```
[188r1a0501@localhost ~]$ vi dp.c
[188r1a0501@localhost ~]$ gcc dp.c
[188r1a0501@localhost ~]$ ./a.out

Enter number of process:5

Enter number of resources:3

Enter total numbers of each resources:10      5      7

Enter Max resources for each process:
for process 1:7      5      3
for process 2:3      2      2
for process 3:9      0      2
for process 4:2      2      2
for process 5:4      3      3

Enter allocated resources for each process:
for process 1:0      1      0
for process 2:2      0      0
for process 3:3      0      2
for process 4:2      1      1
for process 5:0      0      2
```

### **Result:**

Thus, implement Bankers Algorithm to avoid Deadlock is implemented successfully using c program.

<b>Ex: 4(a)</b>	<b>PAGING TECHNIQUE OF MEMORY MANAGEMENT</b>
<b>Date:</b>	

**AIM:**

To write a c program to implement Paging technique for memory management.

**DESCRIPTION:**

Paging is a memory management scheme which permits the physical address space of a process to be noncontiguous. In this scheme physical memory is broken into fixed sized blocks called FRAMES. The logical memory is broken into blocks of same size called PAGES. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. Every address generated by the CPU is divided into two parts

- page number( p )
- page offset ( d )

The size of a page is power of 2 . The selection of power of 2 as the page size makes the translation of logical address into a page number and page offset easy. The size of a page lies between 512 bytes and 16mb per page depending on the computer architecture. When we use the paging scheme, we have no external fragmentation.

**ALGORITHM:**

1. Read all the necessary input from the keyboard.
2. Pages - Logical memory is broken into fixed - sized blocks.
3. Frames – Physical memory is broken into fixed – sized blocks.
4. Calculate the physical address using the logical address
5. Physical address = (Frame number \* Frame size) + offset
6. Display the physical address.
7. Stop the process

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



```

nop = ms/ps;
printf("\nThe no. of pages available in memory are -- %d ",nop);
printf("\nEnter number of processes -- ");
scanf("%d",&np);
rempages = nop;
for(i=1;i<=np;i++)
{
printf("\nEnter no. of pages required for p[%d]-- ",i);
scanf("%d",&s[i]);
if(s[i] >rempages)
{
printf("\nMemory is Full");
break;
}
rempages = rempages - s[i];
printf("\nEnter pagetable for p[%d] --- ",i);
for(j=0;j<s[i];j++)
scanf("%d",&fno[i][j]);
}
printf("\nEnter Logical Address to find Physical Address ");
printf("\nEnter process no. and page number and offset -- ");
scanf("%d %d %d",&x,&y, &offset);
if(x>np || y>=s[i] || offset>=ps)
printf("\nInvalid Process or Page Number or offset");
else
{ pa=fno[x][y]*ps+offset;
printf("\nThe Physical Address is -- %d",pa);
}
getch();
}

```

## OUTPUT:

```
Enter the memory size -- 1000
Enter the page size -- 100
The no. of pages available in memory are -- 10
Enter number of processes -- 3
Enter no. of pages required for p[1]-- 4
Enter pagetable for p[1] --- 8 6 9 5
Enter no. of pages required for p[2]-- 5
Enter pagetable for p[2] --- 1 4 5 7 3
Enter no. of pages required for p[3]-- 5
Memory is Full
Enter Logical Address to find Physical Address
Enter process no. and page number and offset -- 2 3 60
The Physical Address is -- 760
...Program finished with exit code 0
Press ENTER to exit console.
```

## RESULT:

Thus the implementation of paging technique for memory management is executed successfully.

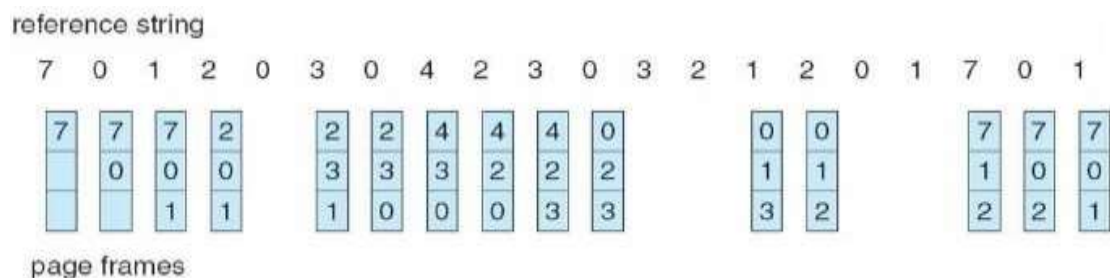
<b>Ex: 4(b)</b>	<b>PAGE REPLACEMENT ALGORITHM</b>  <b>(FIFO)</b>
<b>Date:</b>	

**AIM:**

To write a C program to implement Page Replacement technique using FIFO

**DESCRIPTION:**

**FIFO Page Replacement** – replaces the oldest page in the memory. The page loaded first is removed first. FIFO Page replacement algorithm can be implemented using a FIFO queue. When a page is brought into the memory, we insert it at the tail of the queue. We replace the page at the head of the queue



**ALGORITHM:**

1. Start the program.
2. Get the number of pages and their sequence from the user
3. Get the number of available page frames from the user.
4. In FIFO, on the basics of first in first out, replace the pages respectively, then find number of page faults occurred.
5. Compare all frames with incoming page-
6. If the incoming page is already available in page frame, set the match flag to indicate 'no need of page replacement'.
7. If the incoming page is not available in all frames, then remove the page which is loaded into the memory long back and give space for new incoming page.
8. Increment the 'number of Page faults counter
9. Print the number of page faults.
10. Stop the program.

## PROGRAM:

```
#include<stdio.h>
int main()
{
    int i,j,n,a[50],frame[10],no,k,avail,count=0;
    printf("\n ENTER THE NUMBER OF PAGES:\n");
    scanf("%d",&n);
    printf("\n ENTER THE PAGE NUMBER :\n");
    for(i=1;i<=n;i++)
        scanf("%d",&a[i]);
    printf("\n ENTER THE NUMBER OF FRAMES :");
    scanf("%d",&no);
    for(i=0;i<no;i++)
        frame[i]= -1;
    j=0;
    printf("\tRef string\t Page Frames\n");
    for(i=1;i<=n;i++)
    {
        printf("%d\t\t\t",a[i]);
        avail=0;
        for(k=0;k<no;k++)
            if(frame[k]==a[i])
                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("\nPage Fault Is %d",count);
    return 0;
}
```

## OUTPUT:

```
/tmp/MjeCnIz8jz.o
ENTER THE NUMBER OF PAGES:
8
ENTER THE PAGE NUMBER :
3 5 2 5 7 8 3 5
ENTER THE NUMBER OF FRAMES :3
Ref string    Page Frames
3             3  -1  -1
5             3  5  -1
2             3  5  2
5
7             7  5  2
8             7  8  2
3             7  8  3
5             5  8  3

Page Fault Is 7
```

## RESULT:

Thus the implementation of FIFO page replacement is successfully executed.

Ex: 4(c)	<p style="text-align: center;"><b>PAGE REPLACEMENT ALGORITHM</b></p> <p style="text-align: center;"><b>(LRU)</b></p>
Date:	

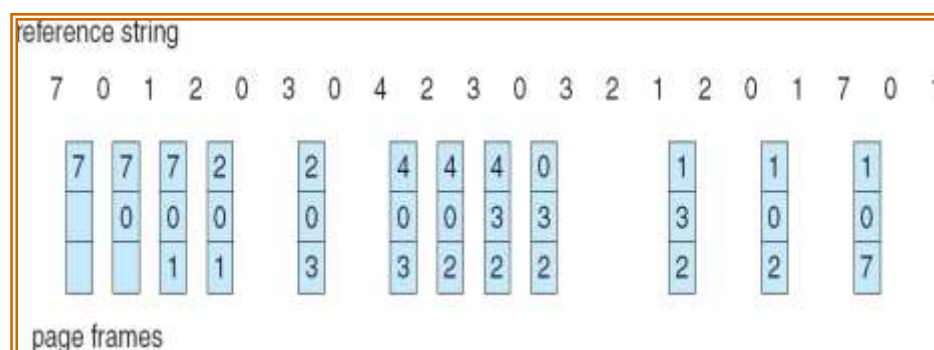
**AIM:**

To write a C program to implement Page Replacement technique using LRU

**DESCRIPTION:**

**LRU page replacement** - If we use the recent past as an approximation of the near future, then we will replace the page that *has not been used* for the longest period of time. This approach is the least-recently-used (LRU) algorithm.

- LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time. This strategy is the optimal page-replacement algorithm looking backward in time, rather than forward.
- The result of applying LRU replacement to our example reference string produces 12 faults. The first faults are the same as the optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently.



- The most recently used page is page 0, and just before that page 3 was used. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3 since, of the three pages in memory {0, 3, 4}, page 3 is the least recently used. Despite these problems, LRU replacement with 12 faults is still much better than FIFO replacement with 15.
- The LRU policy is often used as a page-replacement algorithm and is considered to be good. The major problem is *how* to implement LRU replacement.
- An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use.

## ALGORITHM:.

1. Start the program
2. Get the number of pages and their sequence from the user
3. Get the number of available page frames from the user.
4. In LRU replace the page that *has not been used* for the longest period of time.
5. Compare all frames with incoming page-
6. If the incoming page is already available in page frame, set the match flag to indicate 'no need of page replacement'.
7. If the incoming page is not available in all frames, then remove the page which has not been used for the longest period of time.
8. Increment the 'number of Page faults' counter
9. Print the number of page faults.
10. Stop the program.

## 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: \n");
scanf("%d",&n);
printf("Enter the reference string: \n");
for(i=0;i<n;i++)
scanf("%d",&p[i]);
printf("Enter no of frames: \n");
scanf("%d",&f);
q[k]=p[k];
printf("\t\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:

```
/tmp/MjeCnIz8jz.o
Enter no of pages:
8
Enter the reference string:
3 5 2 5 7 8 3 5
Enter no of frames:
3
3
3
  3  5
  3  5  2
  7  5  2
  7  5  8
  7  3  8
  5  3  8

The no of page faults is 7|
```

## RESULT:

Thus the implementation of LRU page replacement is successfully executed.

<b>Ex: 4(d)</b>	<b>PAGE REPLACEMENT ALGORITHM</b>  <b>(OPR)</b>
<b>Date:</b>	

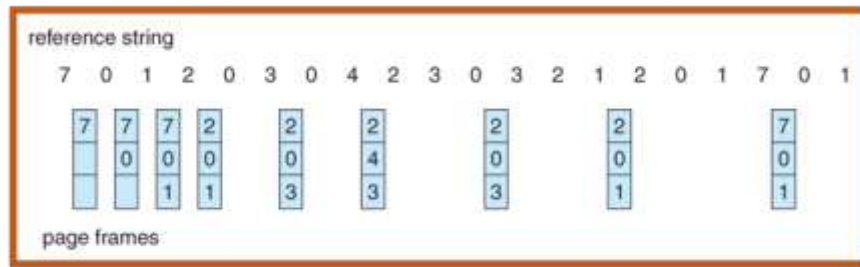
### AIM:

To write a C program to implement Page Replacement technique using OPR

### DESCRIPTION:

**Optimal Replacement-** In optimal we do opposite and look for right further most. Now, this is done so that there are lesser page faults as the element will not use for the longest duration of time in the future.

- The result of the discovery of Belady's Anamoly
- Lowest page fault rate of all algorithm's and will never suffer from belady's Anamoly.
- Simply it replaces the pages that won't be used for longest period of time.
- Optimal page replacement is perfect, but not possible in practice as operating system cannot know future requests.
- The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.



### ALGORITHM:

1. Start the program.
2. Take the input of pages as an array.
3. Look for the page allocated is present in near future, if no then replace that page in the memory with new page,
4. If page already present increment hit, else increment miss.
5. Repeat till we reach the last element of the array.
6. Print the number of hits and misses.
7. Stop the program.

## PROGRAM:

```
#include<stdio.h>
int main()
{
    int i,j,n,a[50],frame[10],no,k,avail,count=0;
    printf("\n ENTER THE NUMBER OF PAGES:\n");
    scanf("%d",&n);
    printf("\n ENTER THE PAGE NUMBER :\n");
    for(i=1;i<=n;i++)
        scanf("%d",&a[i]);
    printf("\n ENTER THE NUMBER OF FRAMES :");
    scanf("%d",&no);
    for(i=0;i<no;i++)
        frame[i]= -1;
    j=0;
    printf("\ntref string\t page frames\n");
    for(i=1;i<=n;i++)
    {
        printf("%d\t\t",a[i]);
        avail=0;
        for(k=0;k<no;k++)
            if(frame[k]==a[i])
                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:

```
/tmp/MjeCnIz8jz.o
ENTER THE NUMBER OF PAGES:
8
ENTER THE PAGE NUMBER :
3 5 2 5 7 8 3 5
ENTER THE NUMBER OF FRAMES :3
ref string    page frames
3          3   -1  -1
5          3   5  -1
2          3   5   2
5
7          7   5   2
8          7   8   2
3          7   8   3
5          5   8   3
Page Fault Is 7
```

## RESULT:

Thus the implementation of OPR page replacement is successfully executed.

**EX. 5a)**

**Date:**

## **DISK SCHEDULING**

### **FIRST COME FIRST SERVE**

#### **AIM:**

To write a program for the first come first serve method of disc scheduling.

#### **DESCRIPTION:**

Disk scheduling is schedule I/O requests arriving for the disk.

It is important because: -

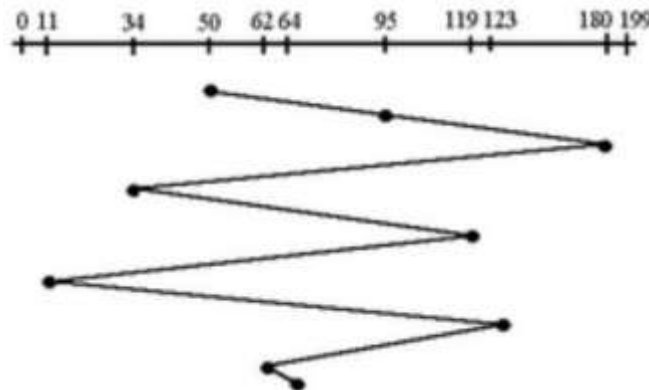
Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by the disk controller. Thus other I/O requests need to wait in the waiting queue and need to be scheduled.

Two or more request may be far from each other so can result in greater disk head movement.

Hard drives are one of the slowest parts of the computer system and thus need to be accessed in an efficient manner

FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.

**Example:** Given the following queue -- 95, 180, 34, 119, 11, 123, 62, 64 with the Read-write head initially at the track 50 and the tail track being at 199.



#### **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);
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:



```

/tmp/FJD1CYE7c0.o
Enter the number of Requests
5
Enter the Requests sequence
9 2 15 20 6
Enter initial head position
10
Total head moment is 40

```

### RESULT:

Thus the implementation of the program for first come first serve disc scheduling has been successfully executed.

**EX. 5b)****DISK SCHEDULING****Date:****SHORTEST SEEK TIME FIRST****AIM:**

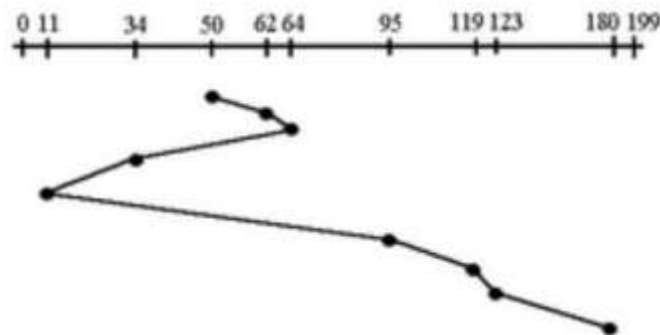
To write a program for the first come first serve method of disc scheduling.

**DESCRIPTION:**

Shortest seek time first (SSTF) algorithm

Shortest seek time first (SSTF) algorithm selects the disk I/O request which requires the least disk arm movement from its current position regardless of the direction. It reduces the total seek time as compared to FCFS.

**Example:-** Given the following queue -- 95, 180, 34, 119, 11, 123, 62, 64 with the Read-write head initially at the track 50 and the tail track being at 199.

**PROGRAM:**

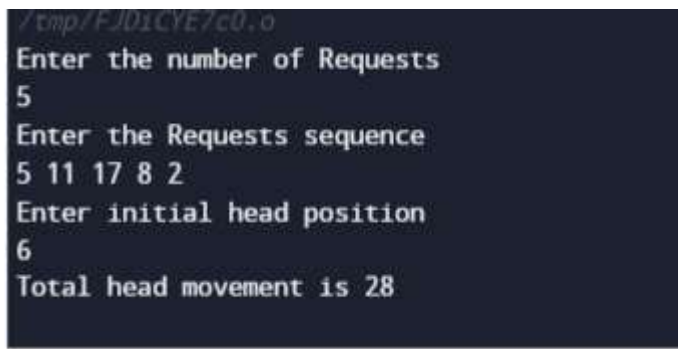
```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int RQ[100],i,n,TotalHeadMoment=0,initial,count=0;
    printf("Enter the number of Requests\n");
    scanf("%d",&n);
    printf("Enter the Requests sequence\n");
    for(i=0;i<n;i++)
        scanf("%d",&RQ[i]);
    printf("Enter initial head position\n");
    scanf("%d",&initial);
    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];
RQ[index]=1000;
    count++;
}
printf("Total head movement is %d",TotalHeadMoment);
return 0;
}

```

### OUTPUT:



```

/tmp/FJD1CYE7c0.o
Enter the number of Requests
5
Enter the Requests sequence
5 11 17 8 2
Enter initial head position
6
Total head movement is 28

```

### RESULT:

Thus the implementation of the program for shortest seek time first disc scheduling has been successfully executed.



**EX. 5c)****DISK SCHEDULING****Date:****SCAN****AIM:**

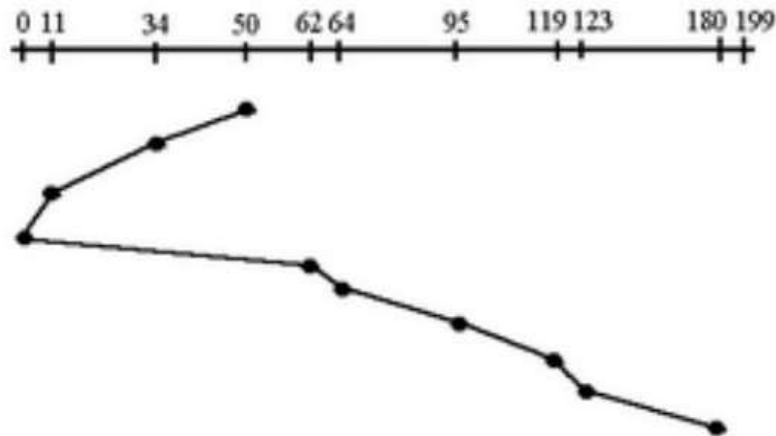
To write a program for the first come first serve method of disc scheduling.

**DESCRIPTION:****SCAN**

It is also called as Elevator Algorithm. In this algorithm, the disk arm moves into a particular direction till the end, satisfying all the requests coming in its path, and then it turns back and moves in the reverse direction satisfying requests coming in its path.

It works in the way an elevator works, elevator moves in a direction completely till the last floor of that direction and then turns back.

**Example:-** Given the following queue -- 95, 180, 34, 119, 11, 123, 62, 64 with the Read-write head initially at the track 50 and the tail track being at 199. head movement is towards low value.

**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];
    }
    // last movement for max size
    TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);
    initial = size-1;
    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];
    }
    TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);
}

```

```

        initial =0;
        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:

```

/tmp/FJD1CYE7c0.o
Enter the number of Requests
5
Enter the Requests sequence
6 14 4 17 9
Enter initial head position
7
Enter total disk size
100
Enter the head movement direction for high 1 and for low 0
1
Total head movement is 187

```

### RESULT:

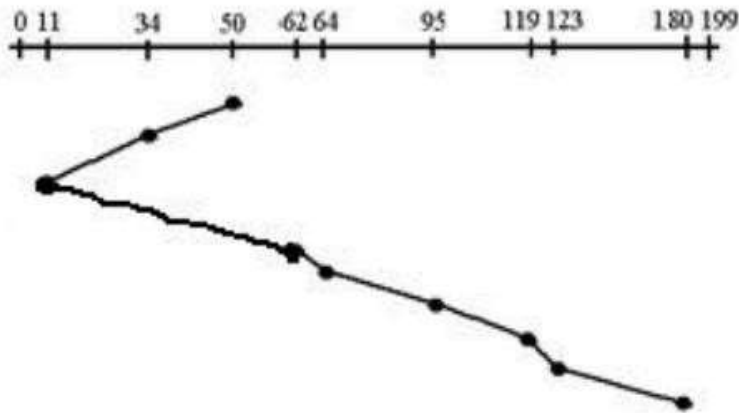
Thus the implementation of the program for SCAN disc scheduling has been successfully executed.

**EX. 5d)****DISK SCHEDULING****Date:****LOOK****AIM:**

To write a program for the first come first serve method of disc scheduling.

**DESCRIPTION****Look**

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

**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",TotalHeadMoment);
    return 0;
}

```

## OUTPUT:

```
/tmp/FJD1CYE7c0.o
Enter the number of Requests
5
Enter the Requests sequence
6 19 4 11 7
Enter initial head position
10
Enter total disk size
100
Enter the head movement direction for high 1 and for low 0
0
Total head movement is 21
```

## RESULT:

Thus the implementation of the program for LOOK disc scheduling has been successfully executed.

<b>EX. 6a)</b>	<b><u>FILE MANAGEMENT USING SEQUENTIAL ALLOCATION</u></b>
<b>Date:</b>	

**AIM:**

To implement file management using sequential list.

**ALGORITHM:**

Step 1: Start the program.

Step 2: Get the number of memory partition and their sizes.

Step 3: Get the number of processes and values of block size for each process.

Step 4: First fit algorithm searches all the entire memory block until a hole which is big enough is encountered. It allocates that memory block for the requesting process.

Step 5: Best-fit algorithm searches the memory blocks for the smallest hole which can be allocated to requesting process and allocates it.

Step 6: Worst fit algorithm searches the memory blocks for the largest hole and allocates it to the process.

Step 7: Analyses all the three memory management techniques and display the best algorithm which utilizes the memory resources effectively and efficiently.

Step 8: Stop the program.

**PROGRAM:**

```
#include <stdio.h>
#include<conio.h>
void main()
{
int f[50], i, st, len, j, c, k, count = 0;
clrscr();
for(i=0;i<50;i++)
f[i]=0;
printf("Files Allocated are : \n");
x: count=0;
printf("Enter starting block and length of files: ");
scanf("%d%d", &st,&len);
for(k=st;k<(st+len);k++)
if(f[k]==0)
count++;
if(len==count)
```

```

{
for(j=st;j<(st+len);j++)
if(f[j]==0)
{
f[j]=1;
printf("%d\t%d\n",j,f[j]);
}
if(j!=(st+len-1))
printf(" The file is allocated to disk\n");
}
else
printf(" The file is not allocated \n");
printf("Do you want to enter more file(Yes - 1/No - 0)");
scanf("%d", &c);
if(c==1)
goto x;
else
exit();
getch();
}

```

### **OUTPUT:**

```

Files Allocated are :
Enter starting block and length of files: 14 3
14 1
15 1
16 1
The file is allocated to disk
Do you want to enter more file(Yes - 1/No - 0)1
Enter starting block and length of files: 14 1
The file is not allocated
Do you want to enter more file(Yes - 1/No - 0)1
Enter starting block and length of files: 14 4
The file is not allocated
Do you want to enter more file(Yes - 1/No - 0)0

```

**RESULT:** Thus, file management using sequential list is implemented successfully.



<b>EX. 6b)</b>	<b><u>FILE MANAGEMENT USING INDEXED ALLOCATION</u></b>
<b>Date:</b>	

**Aim:**

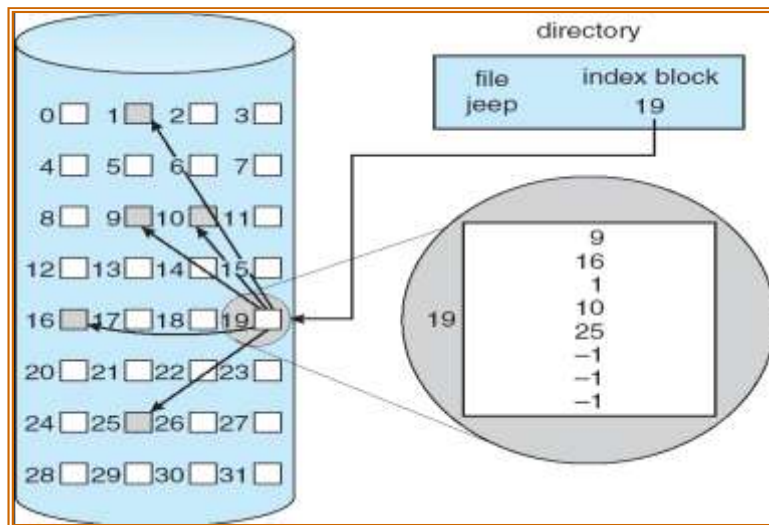
To implement file management using Indexed list.

**Description:**

- Indexed allocation brings all the block pointers together into one location: called the index block.
- Each file has its own index block, which is an array of disk-block addresses. The *i*th entry in the index block points to the *i*th block of the file.
- The directory contains the address of the index block (Figure 5.8).
- To read the *i*th block, we use the pointer in the *i*th index-block entry to find and read the desired block.
- When the file is created, all pointers in the index block are set to nil. When the *i*th block is first written, a block is obtained from the free-space manager, and its address is put in the *i*th index-block entry.
- Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk may satisfy a request for more space.
- If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue:

Linked scheme: An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we may link together several index blocks.

Multilevel index: A variant of the linked representation is to use a first level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block, and that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size.



### **Program:**

```
#include<stdio.h>

#include<conio.h>
#include<stdlib.h>
void main()
{
int f[50], index[50],i, n, st, len, j, c, k, ind,count=0;
clrscr();
for(i=0;i<50;i++)
f[i]=0;
x:printf("Enter the index block: ");
scanf("%d",&ind);
if(f[ind]!=1)
{
printf("Enter no of blocks needed and no of files for the index %d on the disk : \n", ind);
scanf("%d",&n);
}
else
{
printf("%d index is already allocated \n",ind);
goto x;
}
y: count=0;
for(i=0;i<n;i++)
{
```

```
scanf("%d", &index[i]);
if(f[index[i]]==0)
count++;
}
if(count==n)
{
for(j=0;j<n;j++)
f[index[j]]=1;
printf("Allocated\n");
printf("File Indexed\n");
for(k=0;k<n;k++)
printf("%d ----->%d : %d\n",ind,index[k],f[index[k]]);
}
else
{
printf("File in the index is already allocated \n");
printf("Enter another file indexed");
goto y;
}
printf("Do you want to enter more file(Yes - 1/No - 0)");
scanf("%d", &c);
if(c==1)
goto x;
else
exit(0);
getch();
}
```

## **OUTPUT:**

```
Enter the index block: 5
Enter no of blocks needed and no of files for the index 5 on the disk :
4
1 2 3 4
Allocated
File Indexed
5----->1 : 1
5----->2 : 1
5----->3 : 1
5----->4 : 1
Do you want to enter more file(Yes - 1/No - 0)1
Enter the index block: 4
4 index is already allocated
Enter the index block: 6
Enter no of blocks needed and no of files for the index 6 on the disk :
2
7 8
Allocated
File Indexed
6----->7 : 1
6----->8 : 1
Do you want to enter more file(Yes - 1/No - 0)0
```

**RESULT:** Thus, file management using Indexed list is implemented successfully.

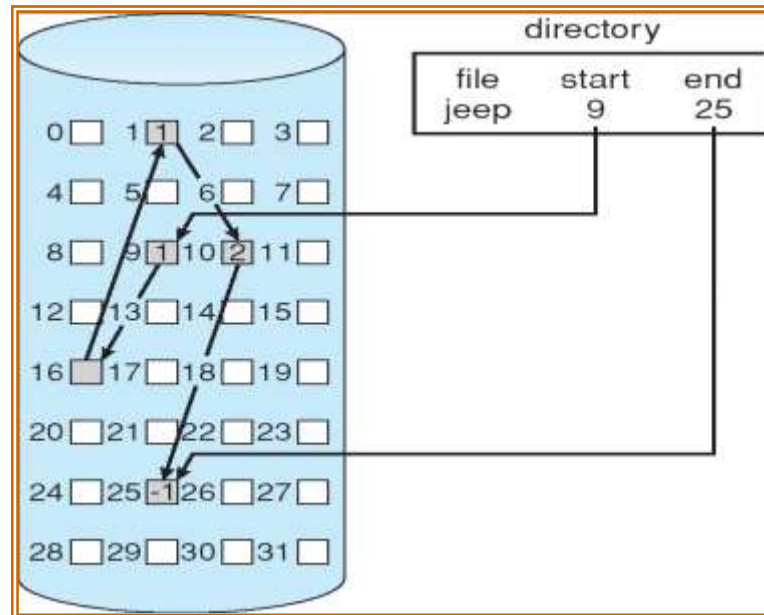
EX. 6c)

Date:

## FILE MANAGEMENT USING LINKED ALLOCATION

### Aim:

To implement file management using Linked list.



### Description:

- **Linked allocation** solves all problems of contiguous 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. These pointers are not made available to the user. Thus, if each block is 512 bytes, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508bytes.
- To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file.
- This pointer is initialized to *nil* (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes a free block to be found via the free-space-management system, and this new block is then written to, and is linked to the end of the file.
- To read a file, we simply read blocks by following the pointers from block to block. There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request.
- The size of a file does not need to be declared when that file is created. A file can

- continue to grow as long as free blocks are available.
- Consequently, it is never necessary to compact disk space.

### **PROGRAM:**

```
#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

void recursivePart(int pages[]){
    int st, len, k, c, j;

    printf("Enter the index of the starting block and its length: ");
    scanf("%d%d", &st, &len);
    k = len;
    if (pages[st] == 0){
        for (j = st; j < (st + k); j++){
            if (pages[j] == 0){
                pages[j] = 1;
                printf("%d ----->%d\n", j, pages[j]);
            }
            else {
                printf("The block %d is already allocated \n", j);
                k++;
            }
        }
    }
    else
        printf("The block %d is already allocated \n", st);
    printf("Do you want to enter more files? \n");
```

```
printf("Enter 1 for Yes, Enter 0 for No: ");
scanf("%d", &c);
if (c==1)
    recursivePart(pages);
else
    exit(0);
return;
}

int main(){
    int pages[50], p, a;

    for (int i = 0; i < 50; i++)
        pages[i] = 0;
    printf("Enter the number of blocks already allocated: ");
    scanf("%d", &p);
    printf("Enter the blocks already allocated: ");
    for (int i = 0; i < p; i++){
        scanf("%d", &a);
        pages[a] = 1;
    }

    recursivePart(pages);
    getch();
    return 0;
}
```

### **OUTPUT:**

```
Enter the number of blocks already allocated: 3
Enter the blocks already allocated: 1 3 5
Enter the index of the starting block and its length: 2 2
2----->1
The block 3 is already allocated
4----->1
Do you want to enter more files?
Enter 1 for Yes, Enter 0 for No: 0

...Program finished with exit code 0
Press ENTER to exit console.□
```

**RESULT:** Thus, file management using Linked list is implemented successfully.