

Exp No1:

Name: PONNURI ANIRUDDHA

Date: 16/08/2022

Reg No: RA2112704010015

Operating system Installation, Basic Linux commands

Aim:

Installation of OS and Practice of basic Unix Shell Commands

Procedure:

INSTALLATION OF:

1. VMware Player

To download VMware Player:

- Navigate to the VMware Download Center.
- Locate VMware Player under Desktop & End user Computing.
- Select the installer from the list according to our host operating system.
- Click Download.
- If prompted, log in to our Customer Connect profile. If we do not have a profile, create one. For more information, see How to create a Customer Connect profile (2007005)
- Ensure that our profile is complete and enter all mandatory fields.
- Review the End User License Agreement and click Yes if we agree.

If the installer fails to download during the download process:

- Delete the cache in our web browser.
- Disable the pop-up blocker in our web browser.
- Microsoft Internet Explorer: How to turn Internet Explorer Pop-up Blocker on or off on a Windows XP SP2-based computer
- Try to download using a different web browser application.
- Disable any local firewall software.
- Restart our machine.
- Download the installer from a different computer or network.

To install VMware Player on a

A. Windows Host:

- Log in to the Windows host.
- Open the folder where the VMware Player installer was downloaded. The default location is the Downloads folder for the user account on the Windows host.
- Right-click the installer and click Run as Administrator.
- Follow the on-screen instructions to finish the installation.
- Restart the host machine.

After Installation:

- The installer creates a desktop shortcut, a quick launch shortcut, or a combination of these options in addition to a Start Menu item.
- To start VMware Player on a Windows host system, select Start > Programs > VMware Player.

B. Linus Host:

- Log in to the Linux host with the user account information that is to be used in the VMware software.
- Open the terminal Interface.
- Change to root.
- Change the directories to the directory that contains the VMware Player bundle installer file. The default location is the download directory for the user.
- Run the appropriate player installer file for the host system.

There are a few command line operations available:

- --gtk
Opens the GUI-based VMware installer, which is the default option.
 - --console
Use the terminal for installation.
 - --custom
Use this option to customize the locations of the installation directories and set the hard limit for the number of open file descriptors.
 - --regular
Shows installation questions that have not been answered before or are required. This is the default option.
 - --ignore-errors or -I
Allows the installation to continue even if there is an error in one of the installer scripts. Because the section that has an error does not complete, the component might not be properly configured.
 - --required
Shows the license agreement only and then proceeds to install Player.
- Accept the license agreement.
- Follow the on-screen instructions or prompts to finish the installation.

- Restart the Linux host.

After Installation

- VMware Player can be started from the command line on all Linux distributions.
- On some Linux distributions, VMware Player can be started in the GUI from the System Tools menu under Applications.
- To start VMware Player on a Linux host system from the command line, run the `vmplayer` command in a terminal window

2.WINDOWS

Step 1 - Format the drive and set the primary partition as active

1. Connect the USB flash drive to our other PC.
2. Open Disk Management: Right-click on Start and choose Disk Management.
3. Format the partition: Right-click the USB drive partition and choose Format. Select the FAT32 file system to be able to boot either BIOS-based or UEFI-based PCs.
4. Set the partition as active: Right-click the USB drive partition and click Mark Partition as Active.

Step 2 - Copy Windows Setup to the USB flash drive

1. Use File Explorer to copy and paste the entire contents of the Windows product DVD or ISO to the USB flash drive.
2. Optional: add an unattended file to automate the installation process. For more information, see Automate Windows Setup.

Step 3 - Install Windows to the new PC

1. Connect the USB flash drive to a new PC.
2. Turn on the PC and press the key that opens the boot-device selection menu for the computer, such as the Esc/F10/F12 keys. Select the option that boots the PC from the USB flash drive.

Windows Setup starts. Follow the instructions to install Windows.

3. Remove the USB flash drive.

3.Linux

- Download .iso or the ISO files on a computer from the internet and store it in the CD-ROM or USB stick after making it bootable.
- We need to restart our computer after attaching pen drive into the computer. Press enter at the time of boot, here select the pen drive option to start the further boot process. Try for a manual boot setting by holding F12 key to start the boot process. This will allow we to select from various boot options before starting the system.
- Set the keyboard layout.

- Now we will be asked What apps would we like to install to start with Linux? The two options are ‘Normal installation’ and ‘Minimal installation’.
- Select the drive for installation of OS to be completed. Select “Erase Disk and install Ubuntu” in case we want to replace the existing OS otherwise select “Something else” option and click INSTALL NOW
- A small panel will ask for confirmation. Click Continue in case we don’t want to change any information provided. Select our location on the map and install Linux.
- Provide the login details.
- After the installation is complete we will see a prompt to restart the computer

3.Dual OS

Dual operating systems can be downloaded in many different ways. Some of those ways are as follows :

- Use linux in windows as a virtual machine : This runs a Linux OS like any other application within Windows. This is also one of the safest ways to get a feel of Linux.
- However, this will utilize our system resources and if we have less than 4Gb of RAM, I won’t advise using it extensively.
- Use a live version of Linux: In this method, we put Linux on a USB or DVD and we boot from it. This is usually slow and our changes done to the Linux system are (normally) not saved. This is particularly useful if we just want to see what Linux feels like.
- Remove Windows and Linux: If we have backed up our data and have a recovery or installation disk of Windows ready with us or if we are determined that we are not going back to Windows, we can remove Windows completely and use only Linux.
- Install Linux alongside Windows: This method is called dual booting Linux with Windows. Here, we install Linux on a system that already has Windows. And when our system powers up, we can choose if we want to use Windows or Linux. This involves touching the disk partition and sometimes boot order. Absolute beginners often find it complicated but this is the best way to use Linux and Windows together in one system. And in this article, we’ll see how to dual boot Linux Mint with Windows 10.

The steps that are to be followed for downloading linux as the second operating system are as follows:

1. Create a live USB or disk:

A live USB is a portable USB-attached external data storage device containing a full operating system that can be booted from. The term is reminiscent of USB flash drives but may encompass an external hard disk drive or solid-state drive, though they may be referred to as "live HDD" and "live SSD" respectively.

2. Make a new partition for the Linux meet:

In the windows system go to the start menu and click partition then a disk management utility is shown. Open the Disk Management System. In this disk management system carefully select the disk from which we should shrink the volume to create the required space for our required partition.

3. Boot in to live USB:

Plug the live USB or disk into the computer and restart the computer. While booting the computer press F10 or F12 function key (defers from computer to computer) to go to the boot menu. Now, choose the option to boot from USB or Removable Media.

4. Start the installation:

This step might take some time for completion. The first thing asked is the language in which this will operate. Then it does a few checks on the available space, battery and internet connection.

5. Prepare the partition:

The most important aspect in downloading the dual os is where the files are to be stored. The application gives a few options on where to store the files. The options given are as follows:

The last option should be selected.

6. Create root, swap and home:

Create a root partition. Choose the available free space and then press +

The next part is to create a swap partition. The next query is regarding the swap size that should be put. This depends on the available RAM size, the user needs, available disk space and whether the user uses hibernation or not. The suggested swap size for RAM available is as follows:

RAM less than 2 GB: Swap should be double the size of RAM

RAM between 2 to 4 GB: Swap should be RAM size + 2 GB

RAM between 6 GB to 8 GB: Swap should be size of RAM

RAM more than 8 GB: Swap should be half the size of RAM or less

The next step is to create Home. Try to allocate the maximum space available for home because this is where the files will be stored and saved.

7. Then click on the install now button.

8.Complete the rest of the process.

9.After completing the rest of the process restart the system

10.Dual Operating systems have been installed in the systems and can be used at any required time.

4. Linux in VMWare:

. There are a few prerequisites that need to be followed to install Ubuntu on VMWare. The requirements to do so are as follows:

1. A host system with minimum conditions of:

1. 8GB of memory
2. A Quad core CPU
3. 500 GB of hard disk space

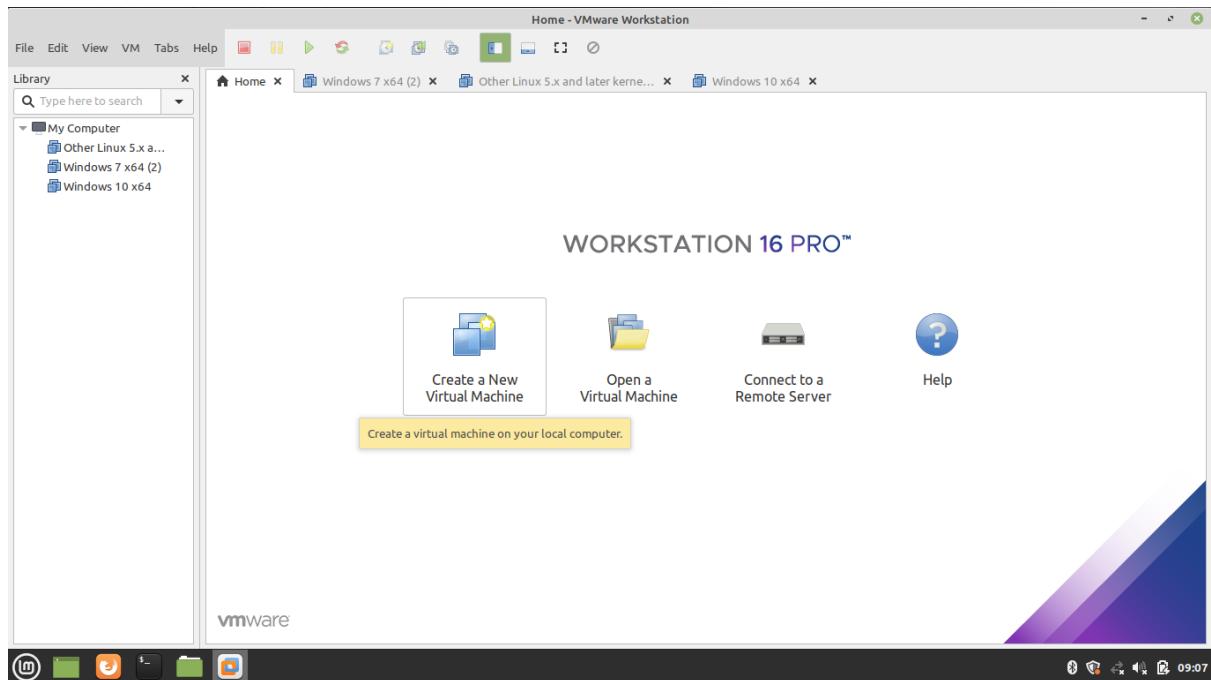
2. VMWare Workstation Pro or Player Application:

3. Ubuntu Operating system to install on VMWare Workstation

The steps to download linux in VMWare is as follows:

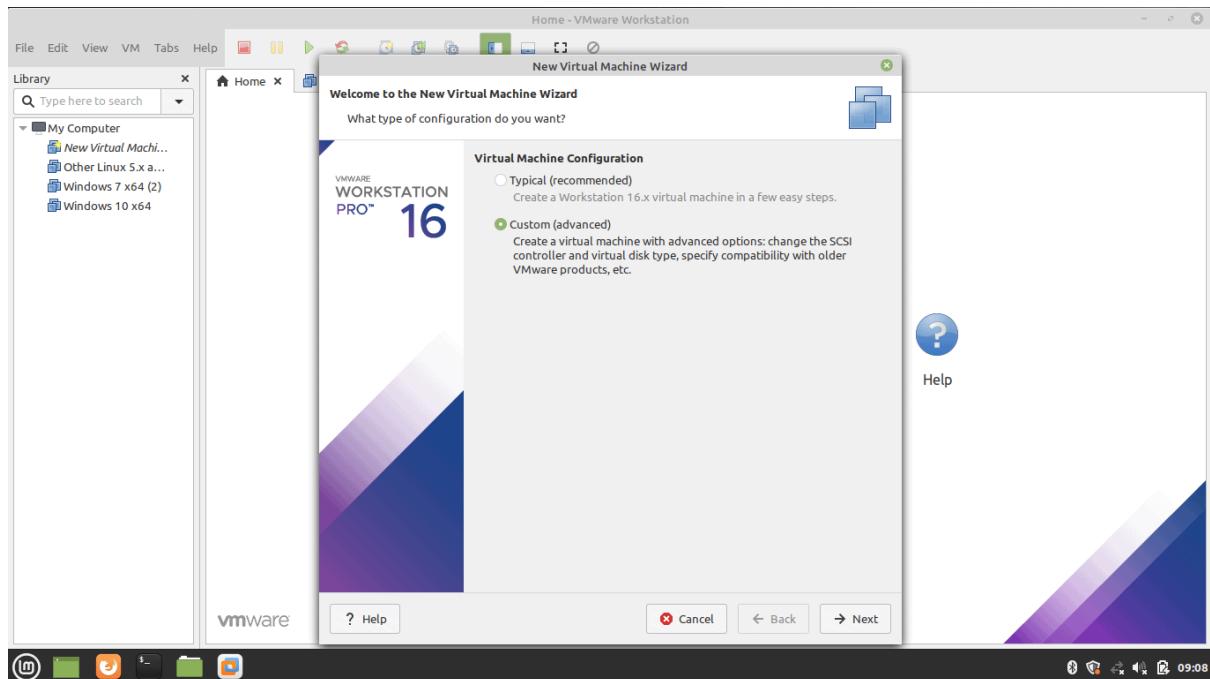
1. Fire up VMWare Workstation

Download the VMWare Workstation application for our host operating system and install it on our machine. The installation procedure is pretty simple and straight. Read the documentation for more details. Open the app after installation. Create a new Virtual Machine.



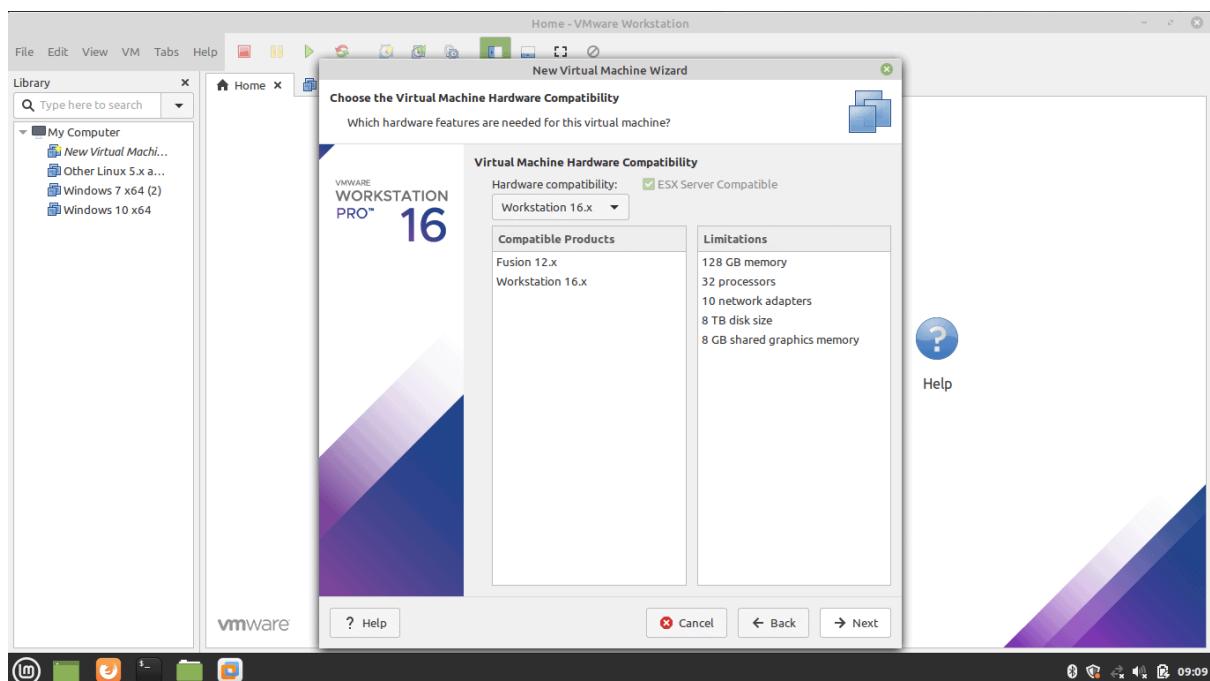
2. Select Custom Configuration Wizard

We can choose either Typical or Custom Wizard. We recommend selecting Custom if we want to install with all the configurations. If we are okay with default configurations then go ahead with Typical configurations.



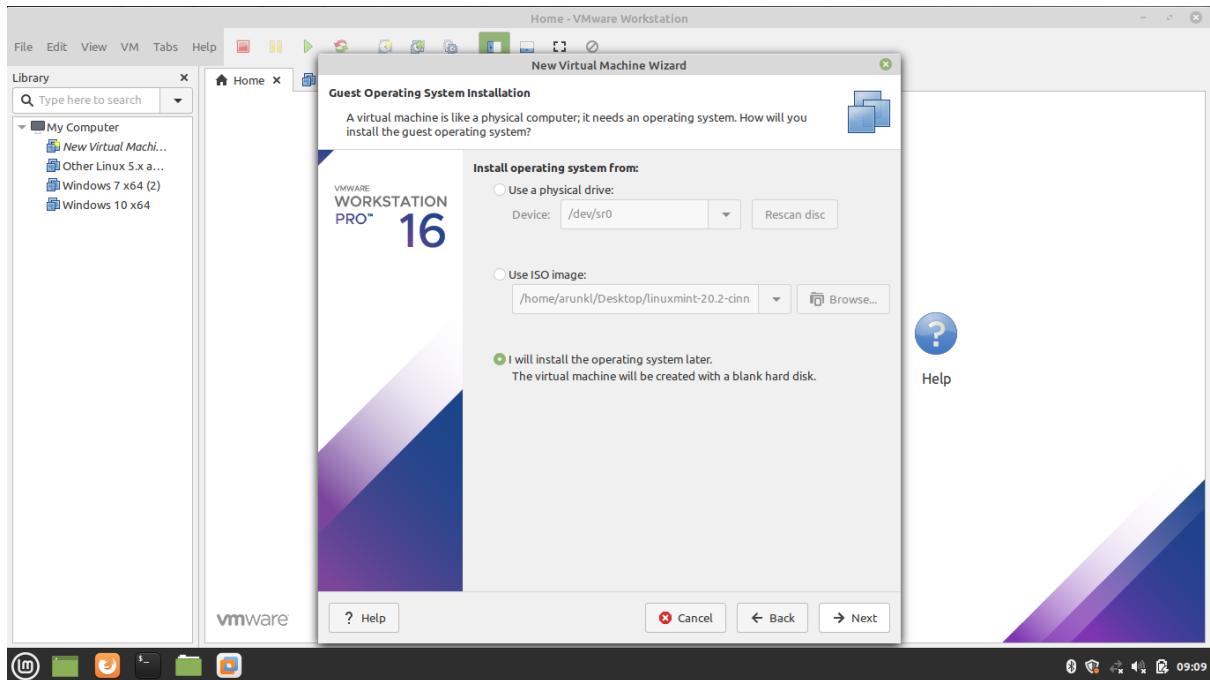
3. Select Virtual Machine Hardware Compatibility

Go with the default option if we don't have the choice.

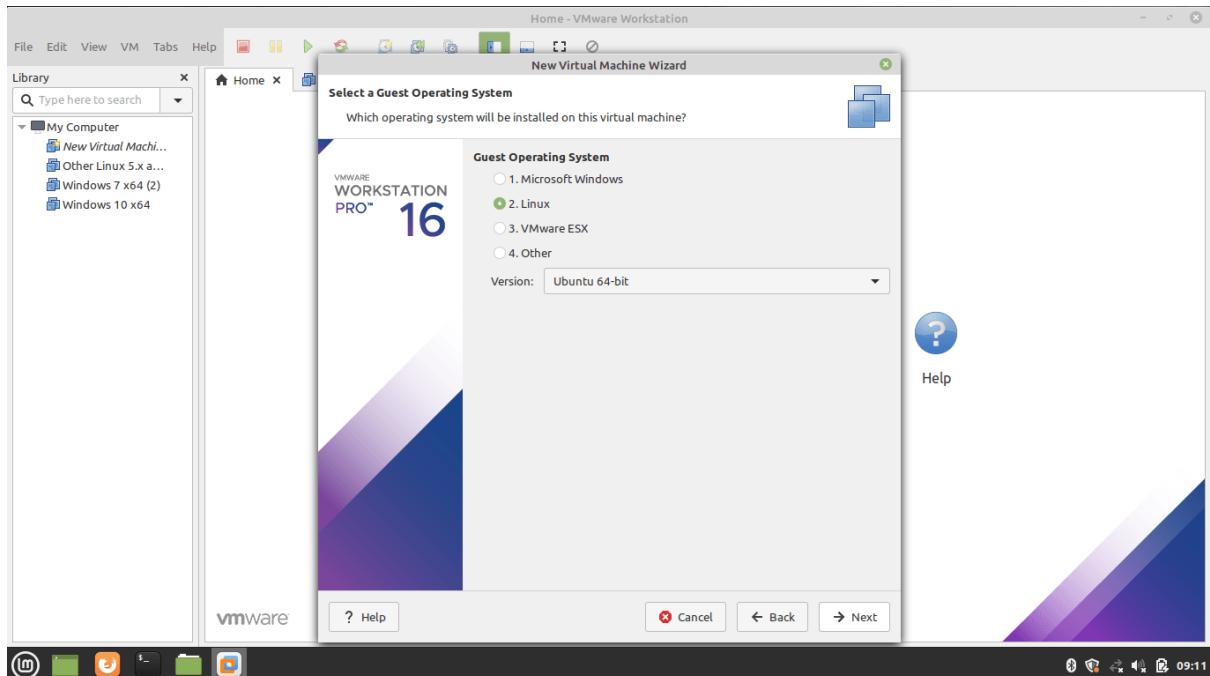


4. Select the Operating System Media

Select ‘I will install the operating system later’ for an interactive installation.

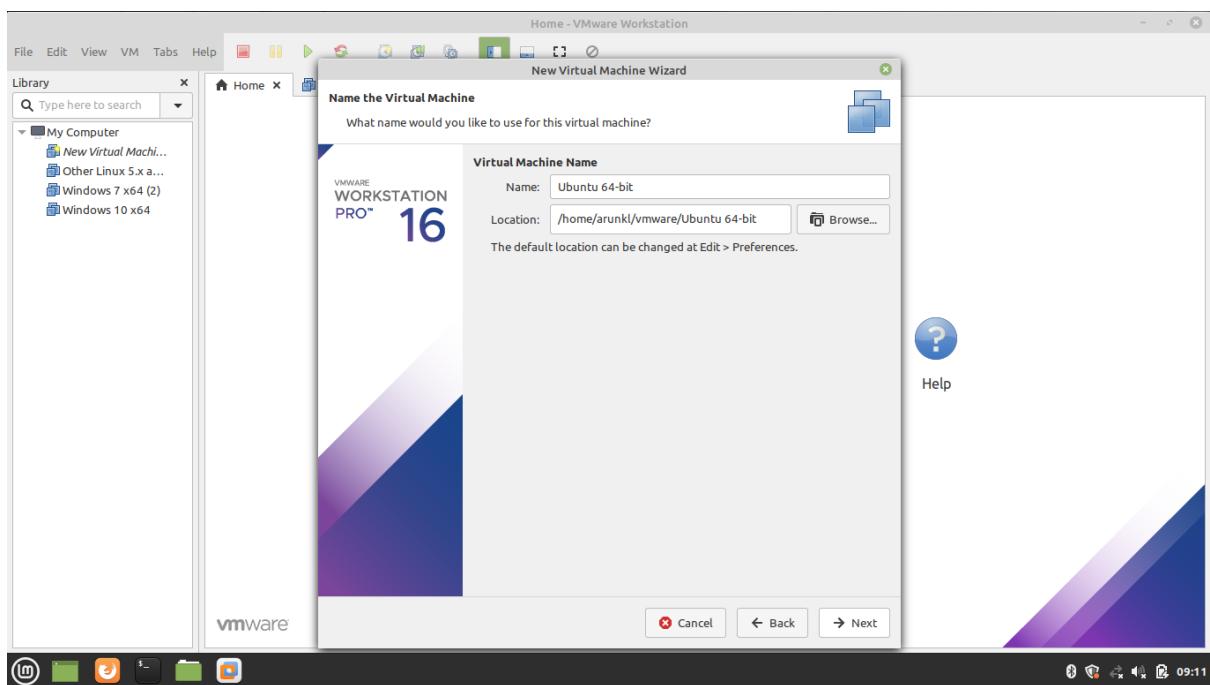


5. Select Guest Operating System



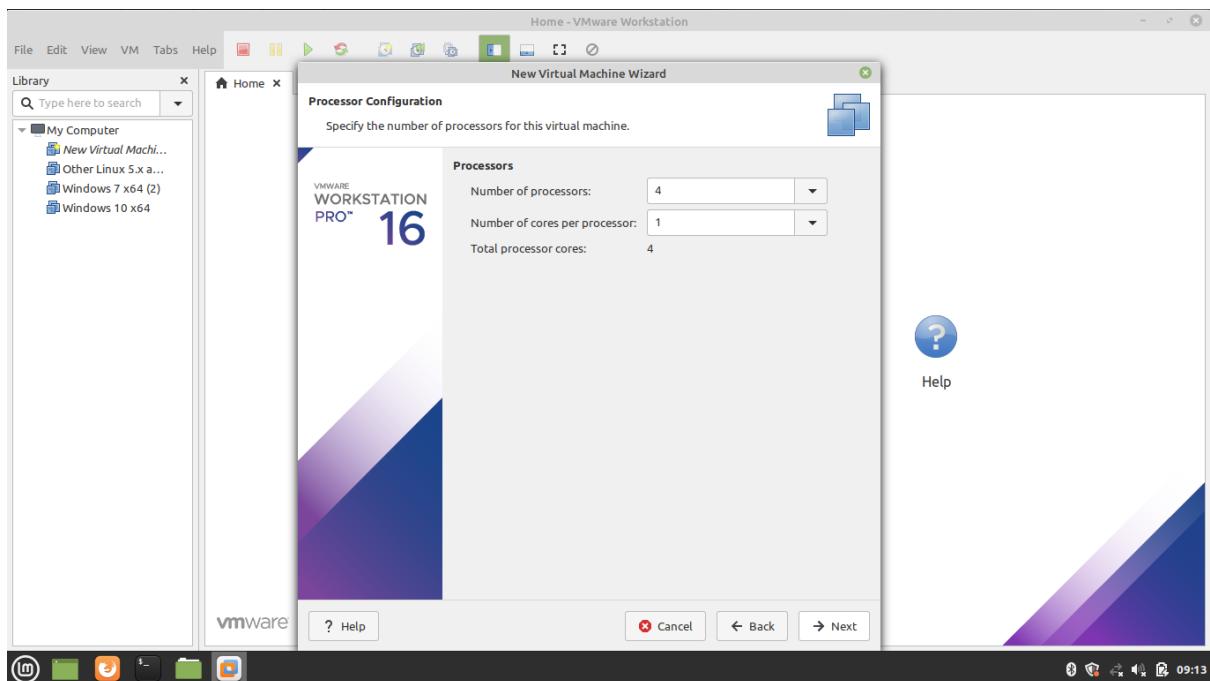
6. Name the Virtual Machine Name and location

Type a name and give the location details.



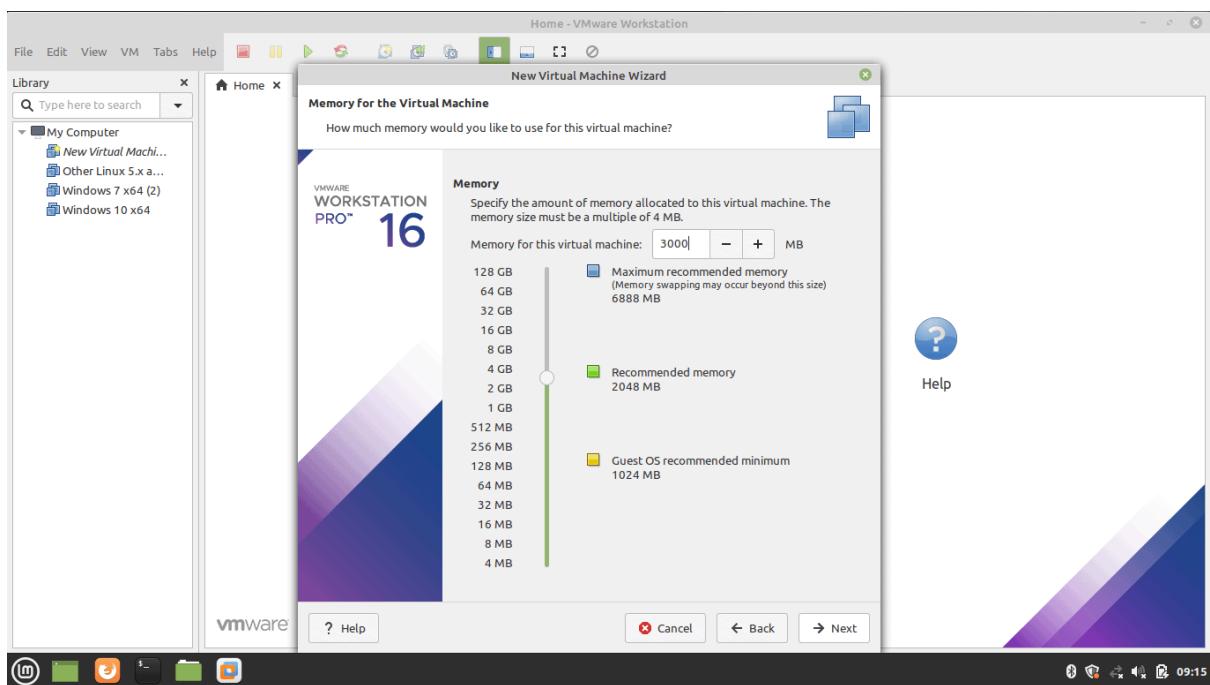
7. Allocate the Processors

Assign the processors, Calculate the processor required to run the host machine. Assign the leftover resources to the virtual machine.



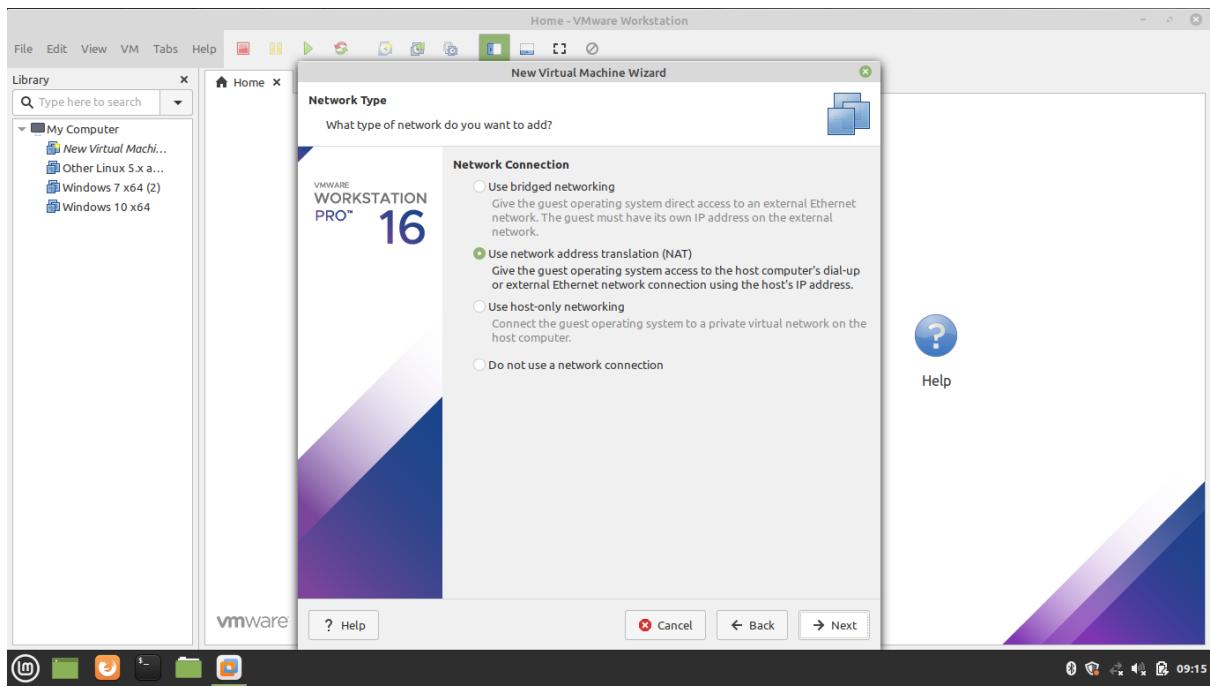
8. Allocate the Memory for Virtual Machine

Memory allocation calculation is the same as the processor allocation. Leave sufficient memory for the host system and allocate the remaining memory for the virtual machine.

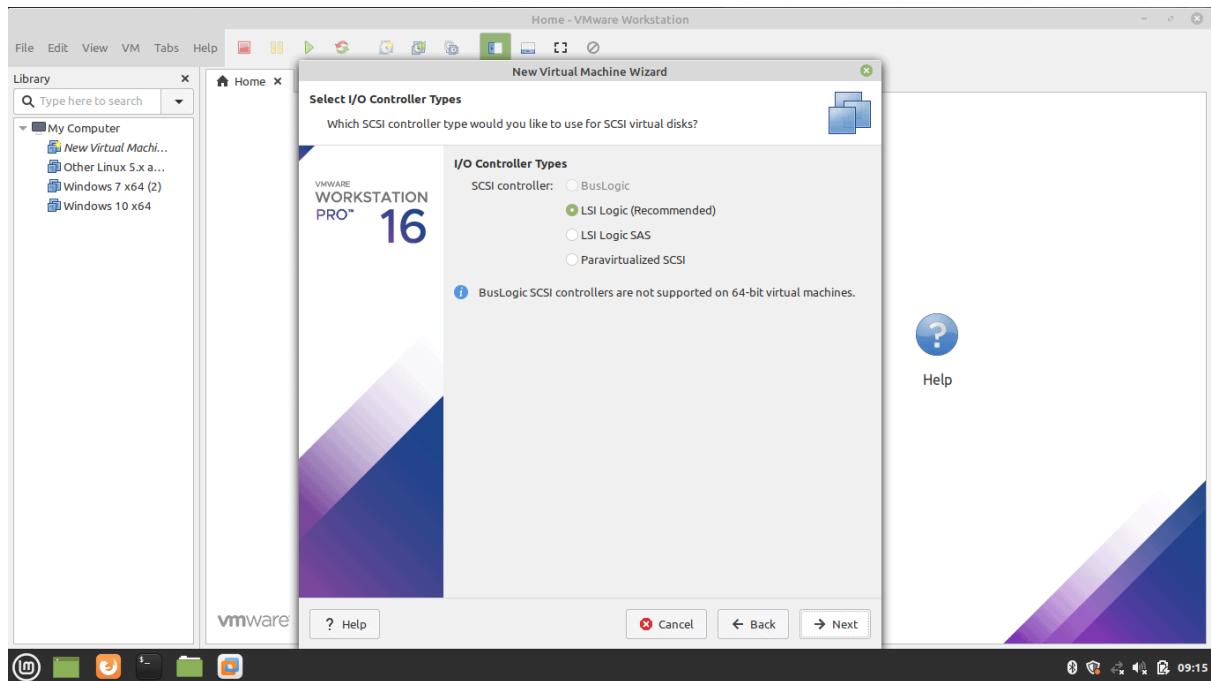


9. Choose the Network Configuration

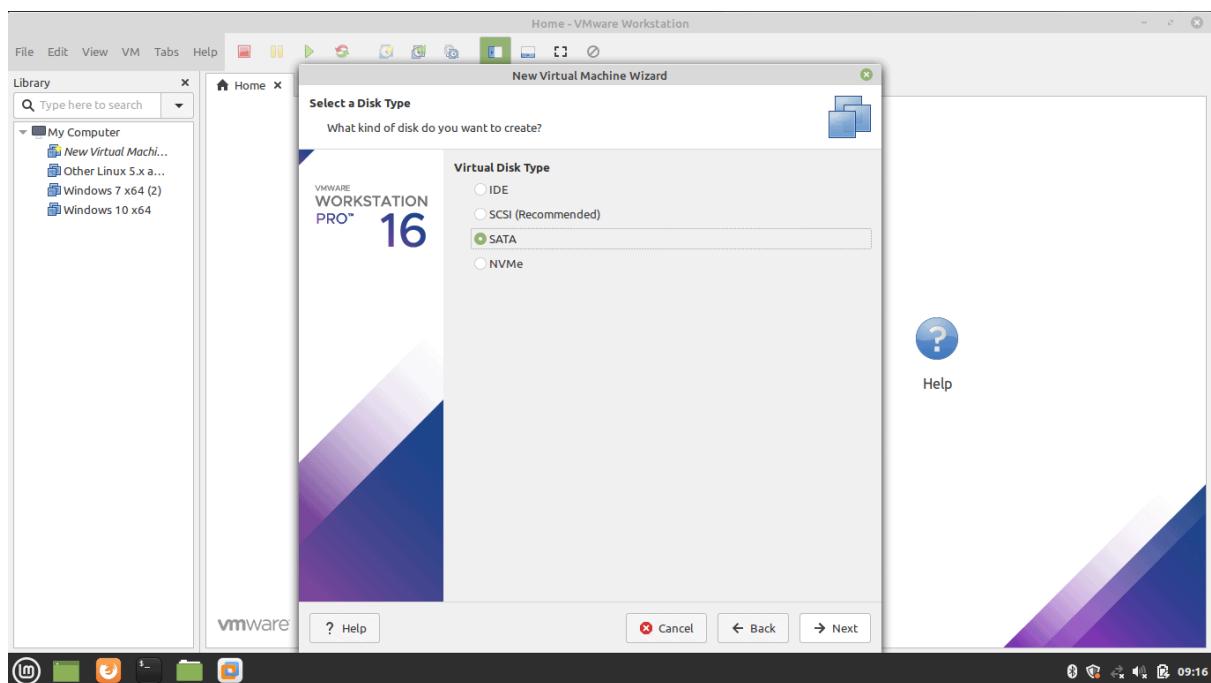
Select any one of the network configurations as per our requirement.



10. Select the I/O Controller Type

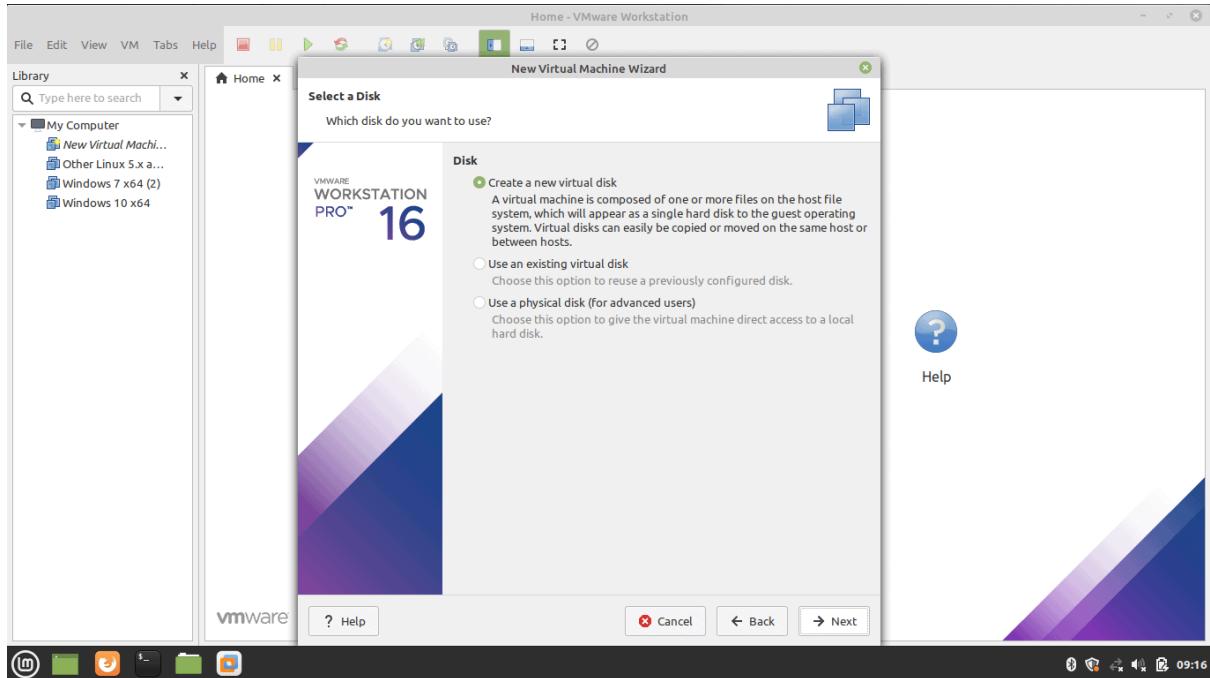


11. Select Disk Type



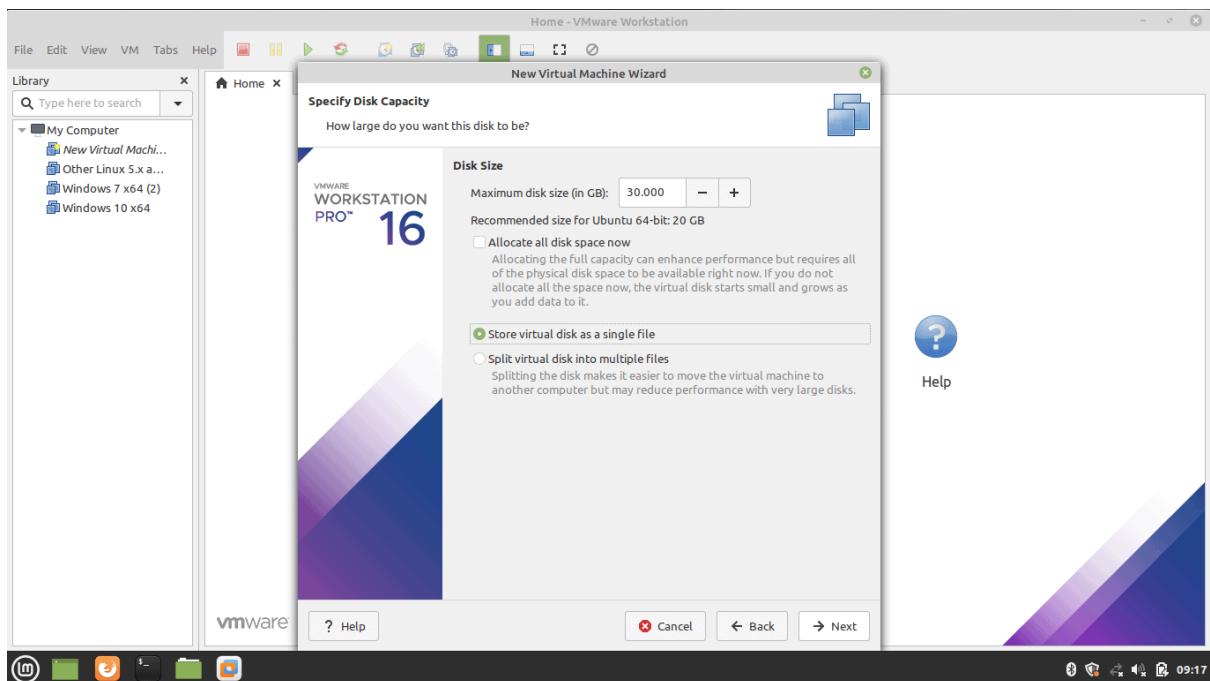
12. Select Virtual Disk

Select the Virtual Disk if we have or create one.

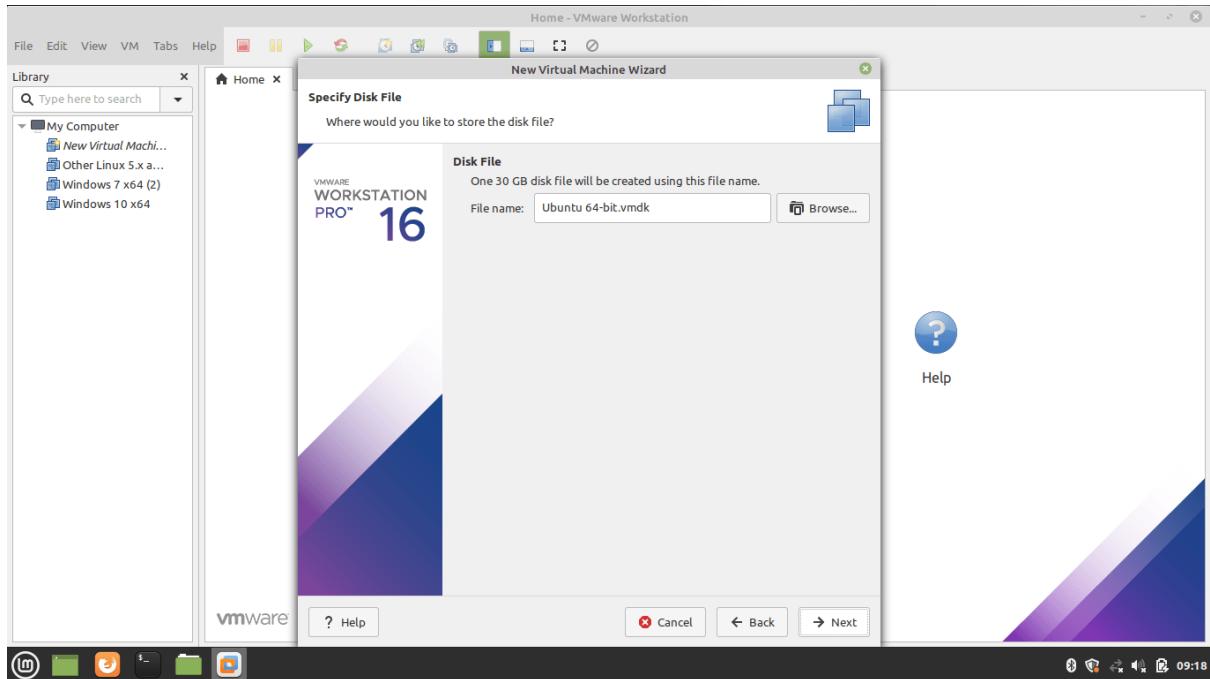


13. Select Disk Capacity

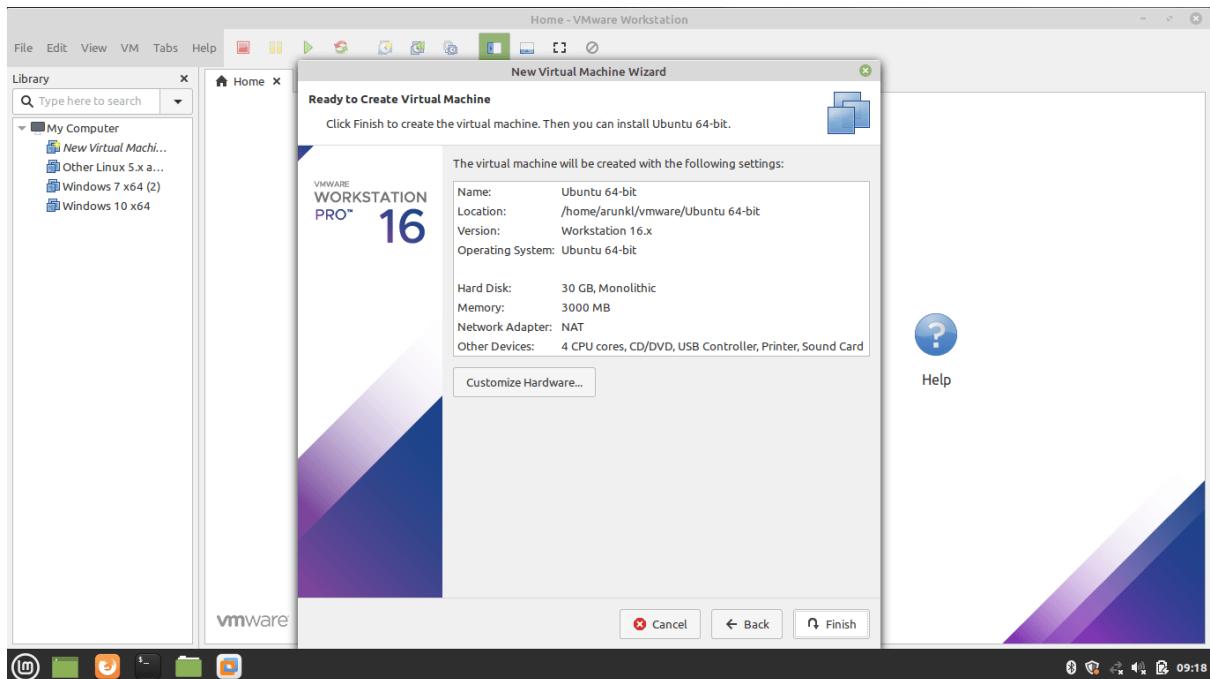
Select the disk size. Selecting a single disk will increase the performance. However, selecting a split disk will help in the disk transfer scenario.



14. Specify Virtual Disk File

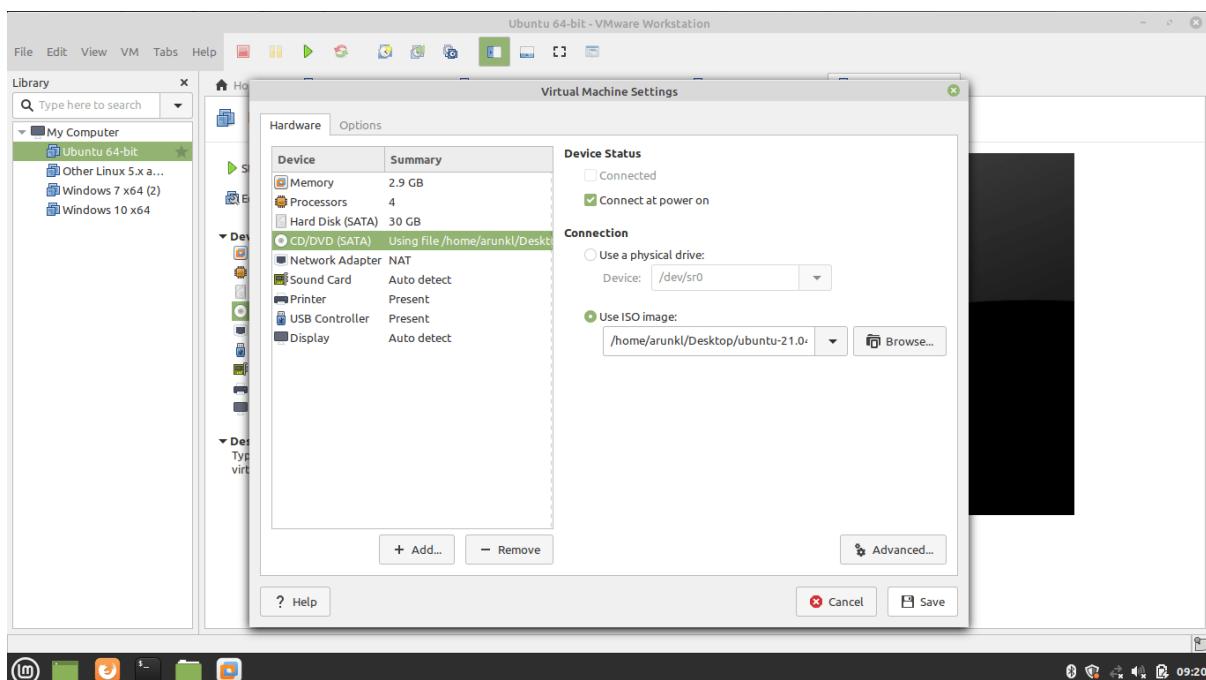


15. Create Virtual Machine

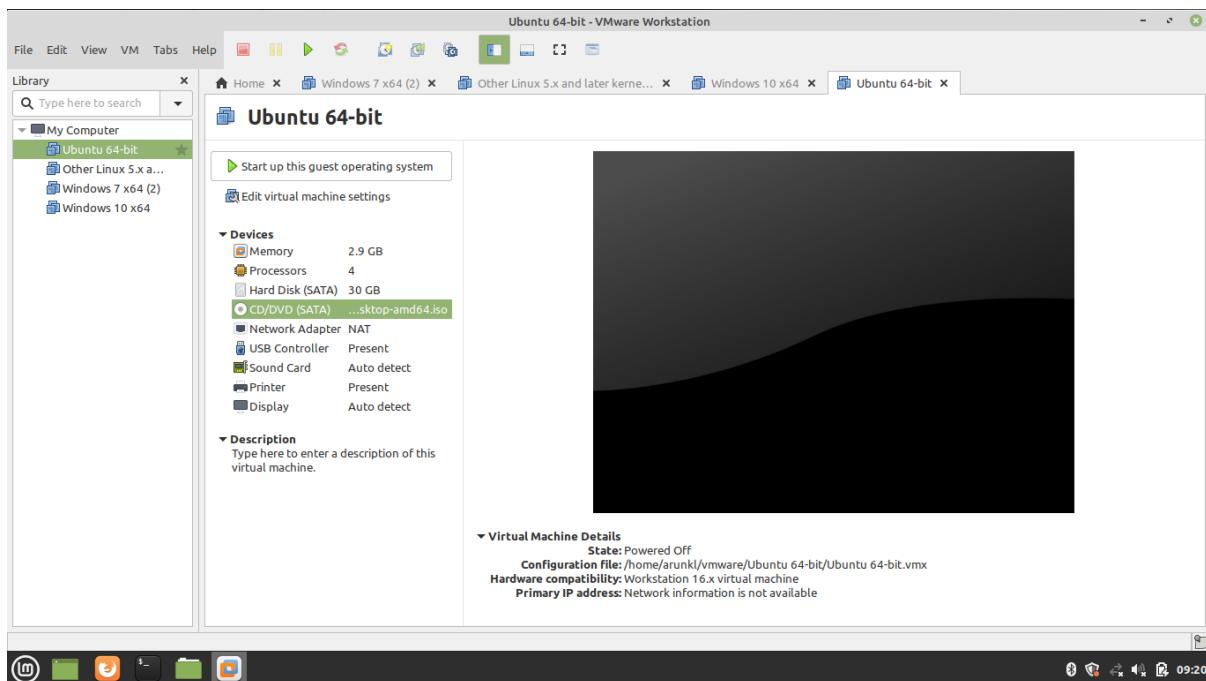


16. Supply Ubuntu ISO Image to Virtual Machine

Download Ubuntu image. Edit the CD/DVD settings and import the downloaded Ubuntu image.

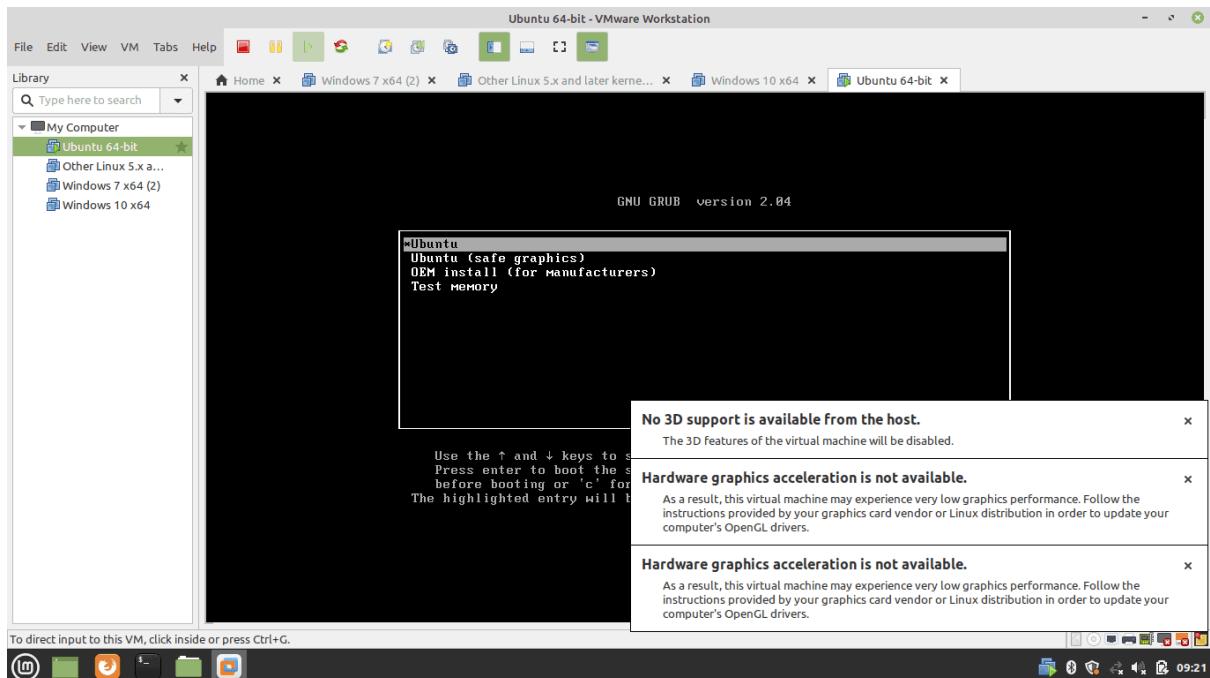


17. Install Ubuntu Linux on VMWare Workstation



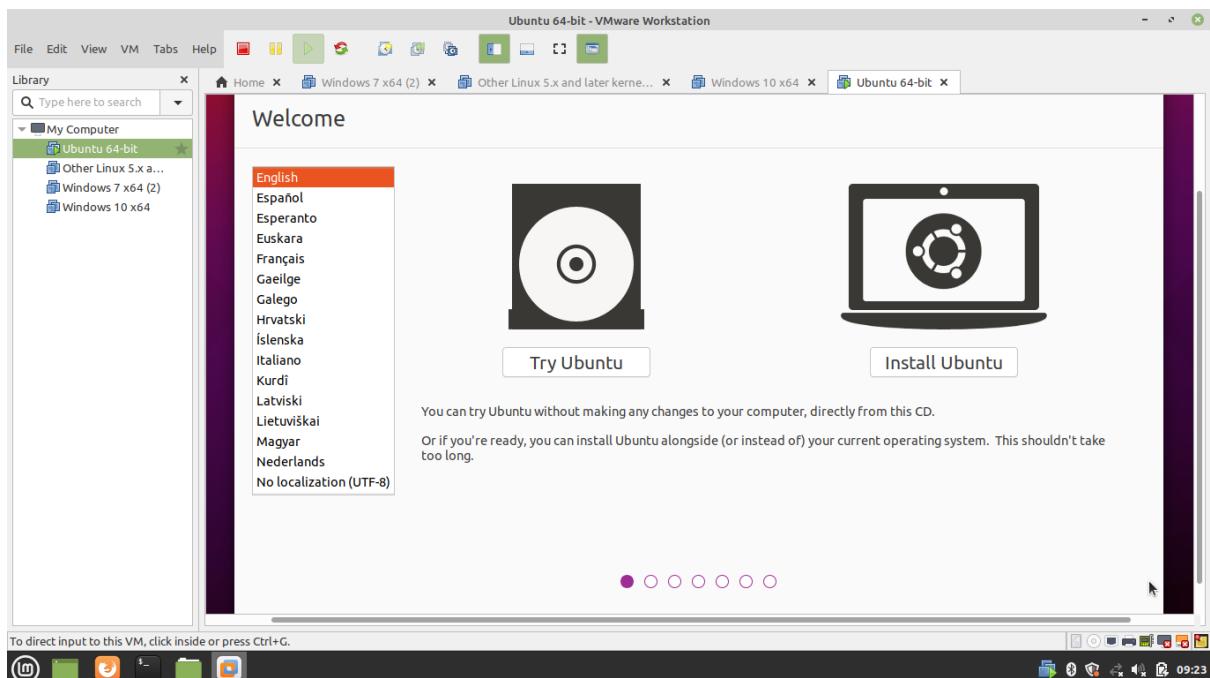
18. Power On the Virtual Machine

Press the Play button to power on the Virtual Machine.

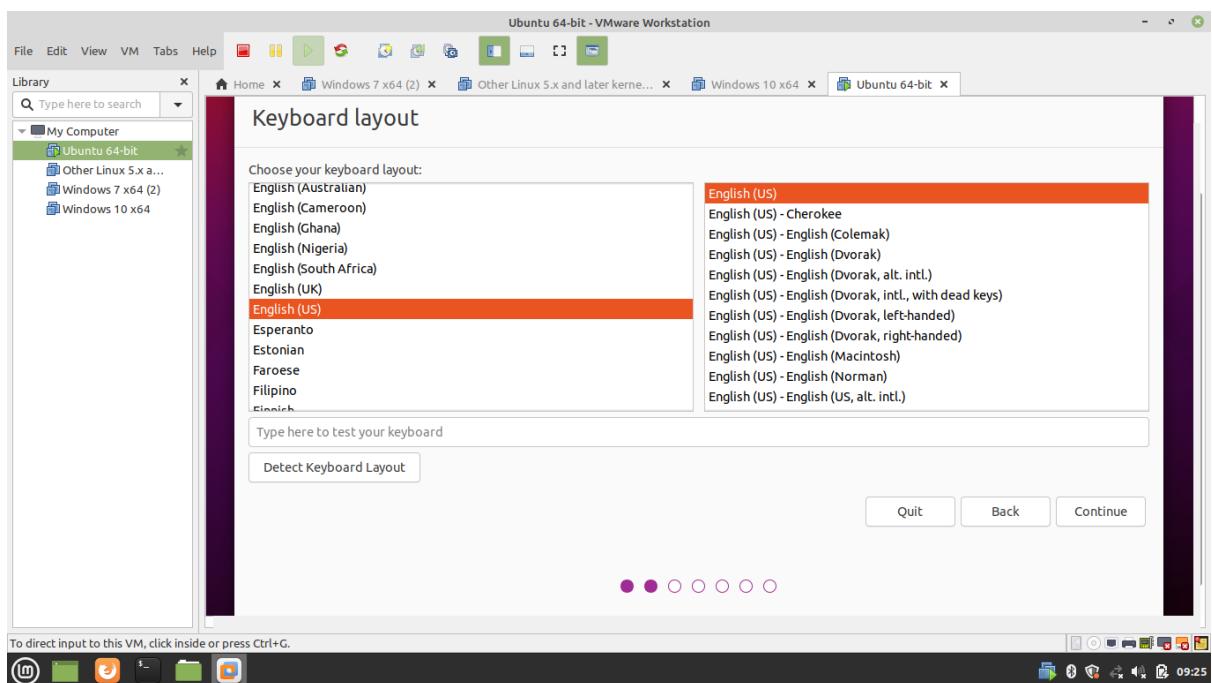


19. Welcome Ubuntu Virtual Machine

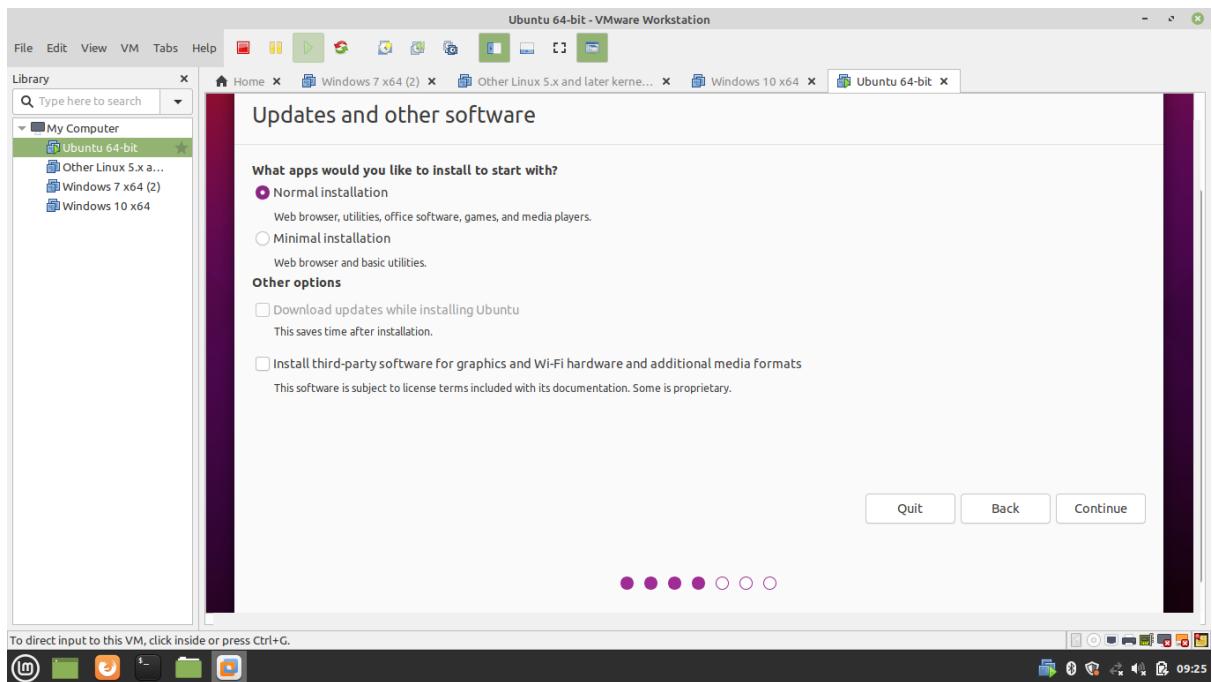
After powering on the Virtual machine, we will be treated with a welcome screen on which we will see two options: Try Ubuntu and Install Ubuntu. Select Try Ubuntu if we want to run Ubuntu in live mode. Select Install to continue the installation process.



20. Select Keyboard Lawet

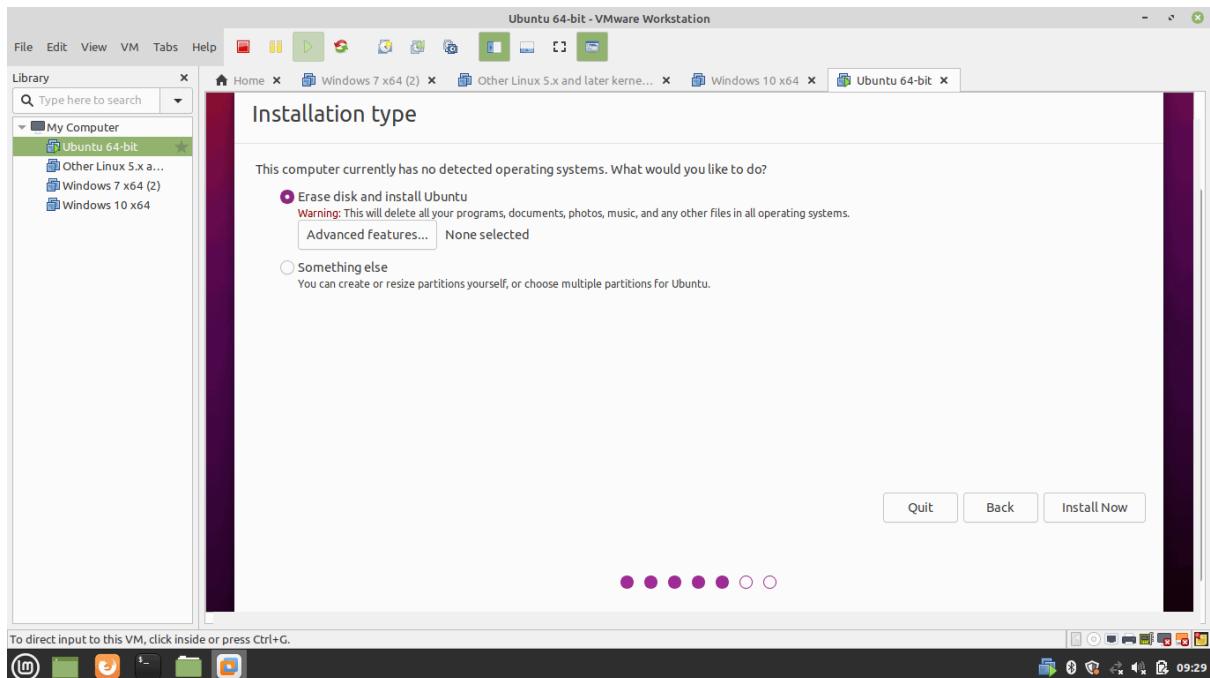


21. Software Update and Package Selection in Virtual Machine

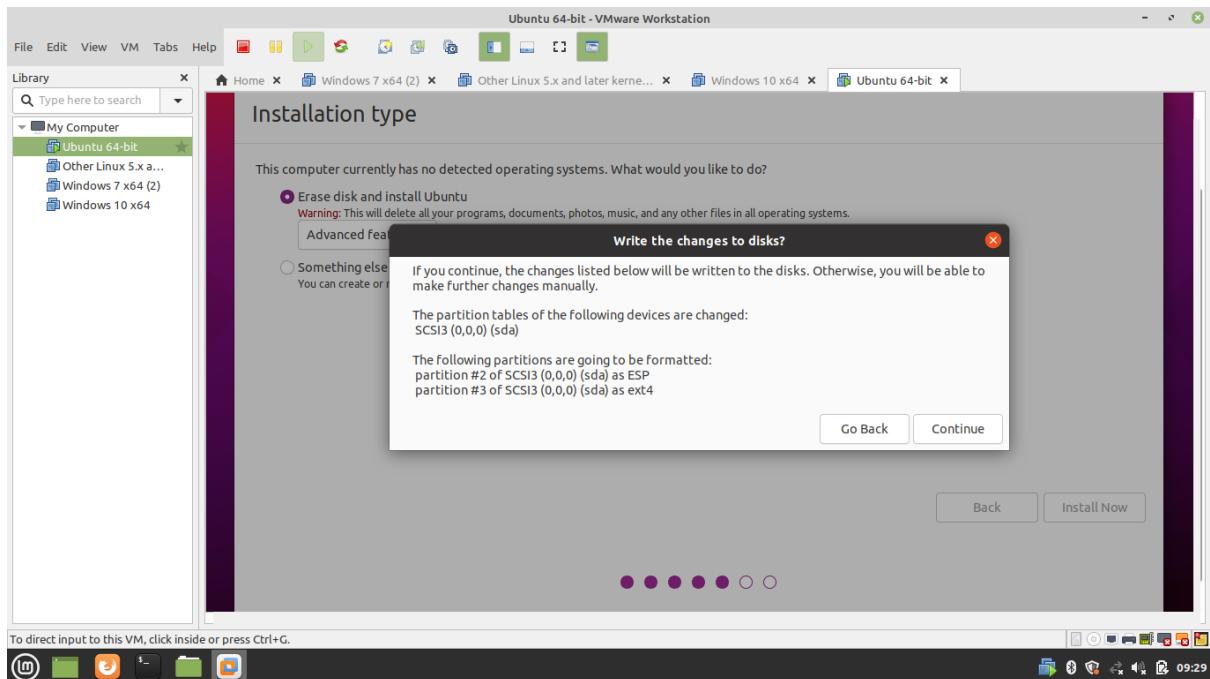


22. Partition the Disk

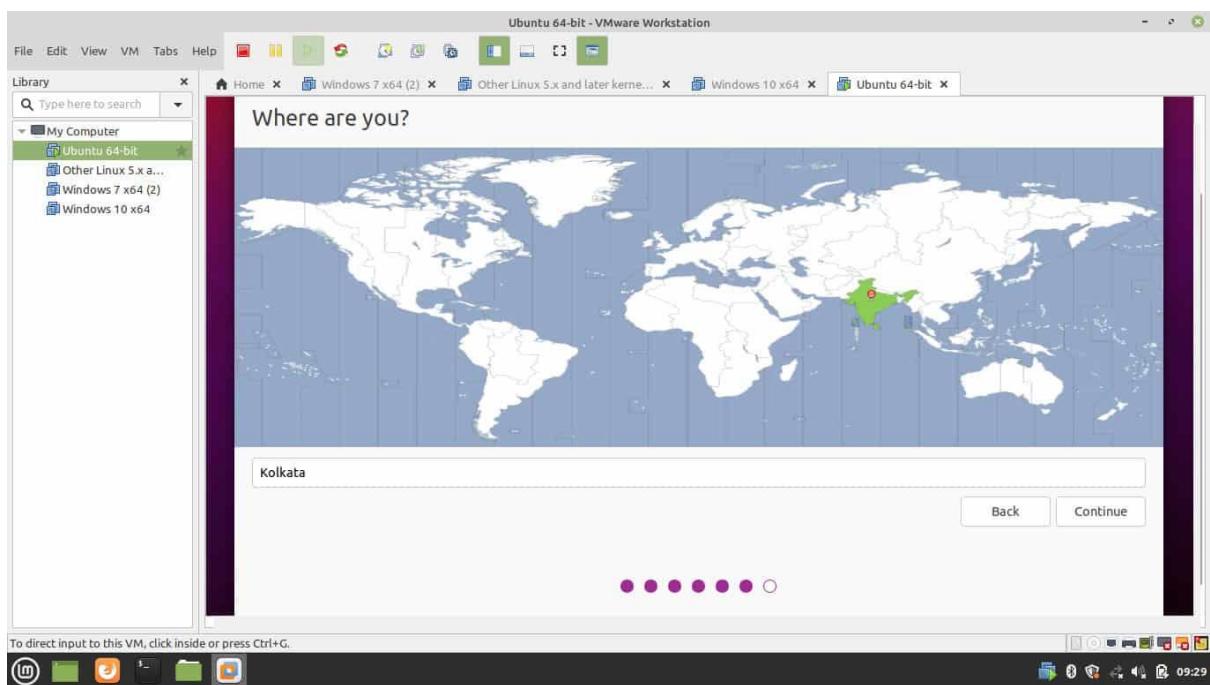
Select Erase the Disk for auto partition. Or Select the Advance option to create the custom partition.



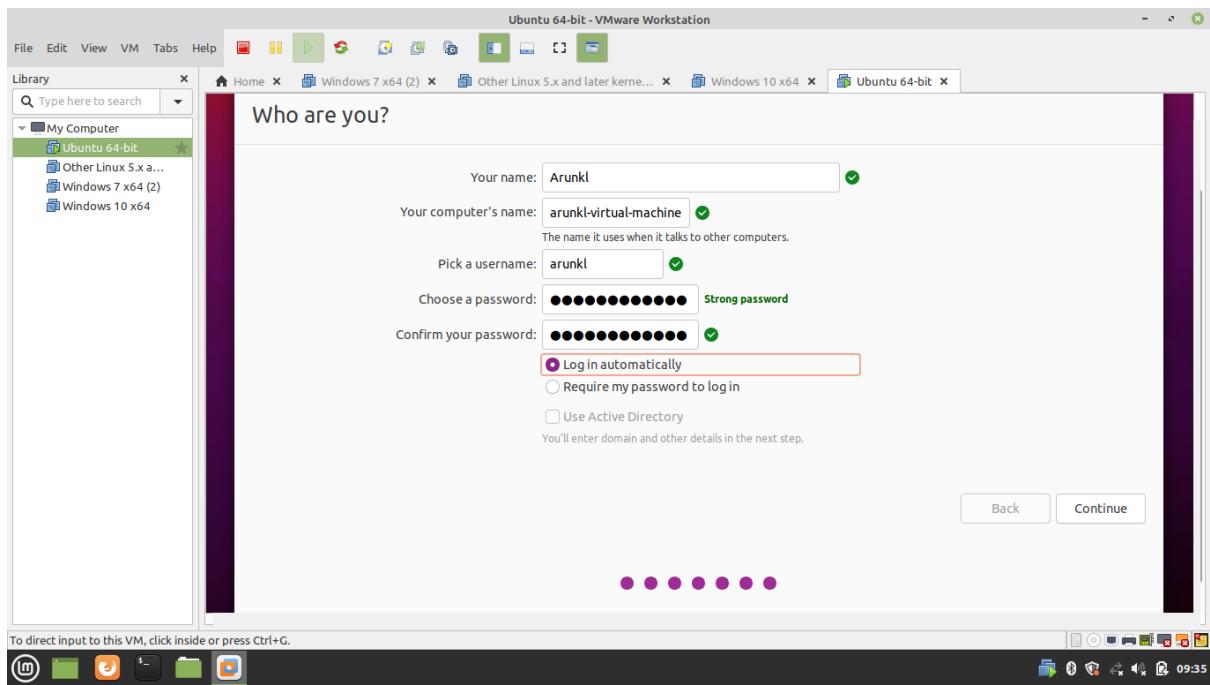
23. Write Changes to Disk



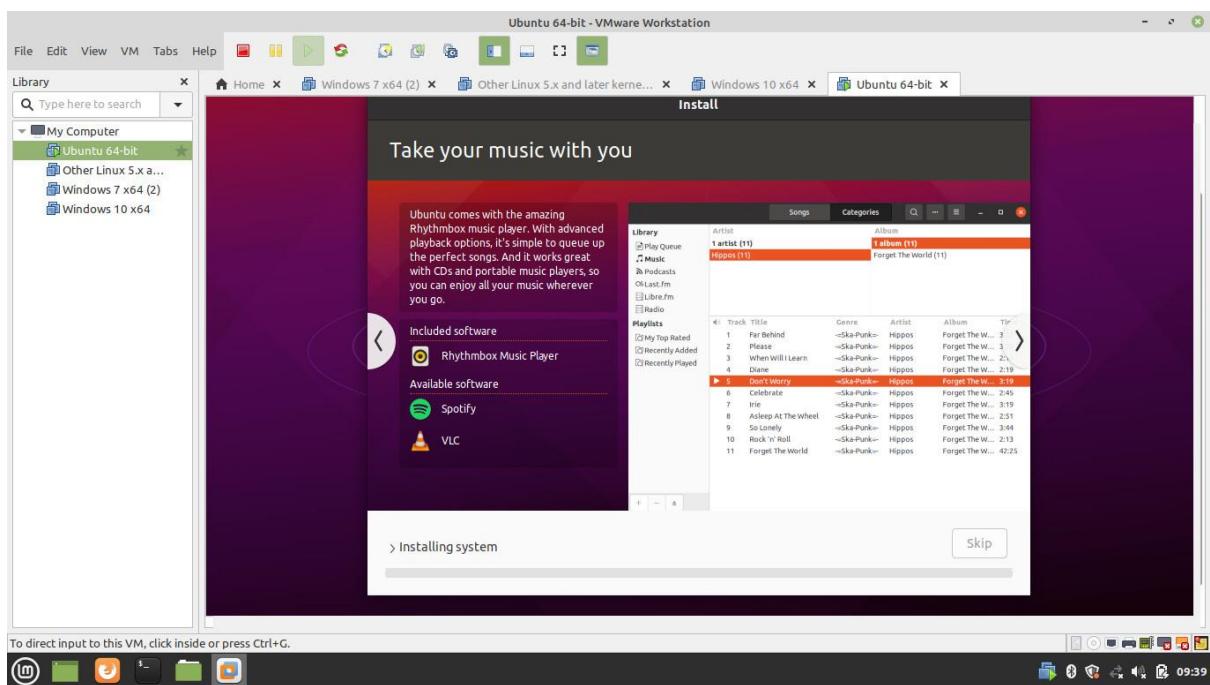
24. Select Time Zone



25. Create an Admin Account

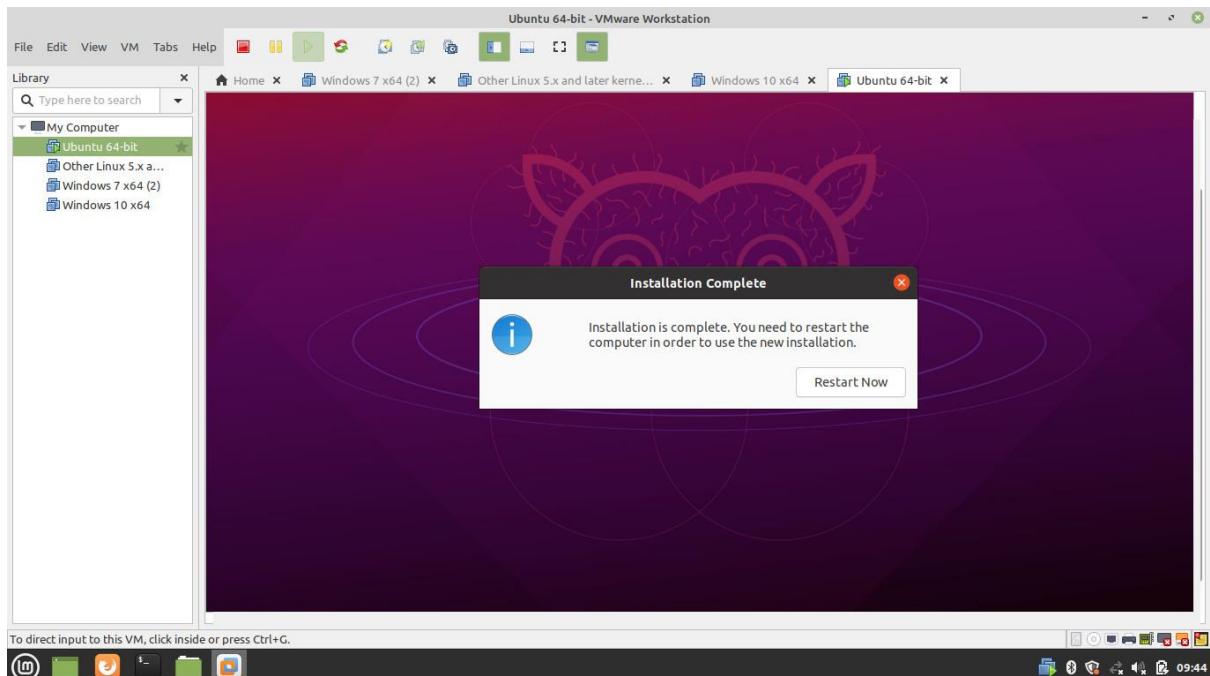


26. Installation of Ubuntu in Progress



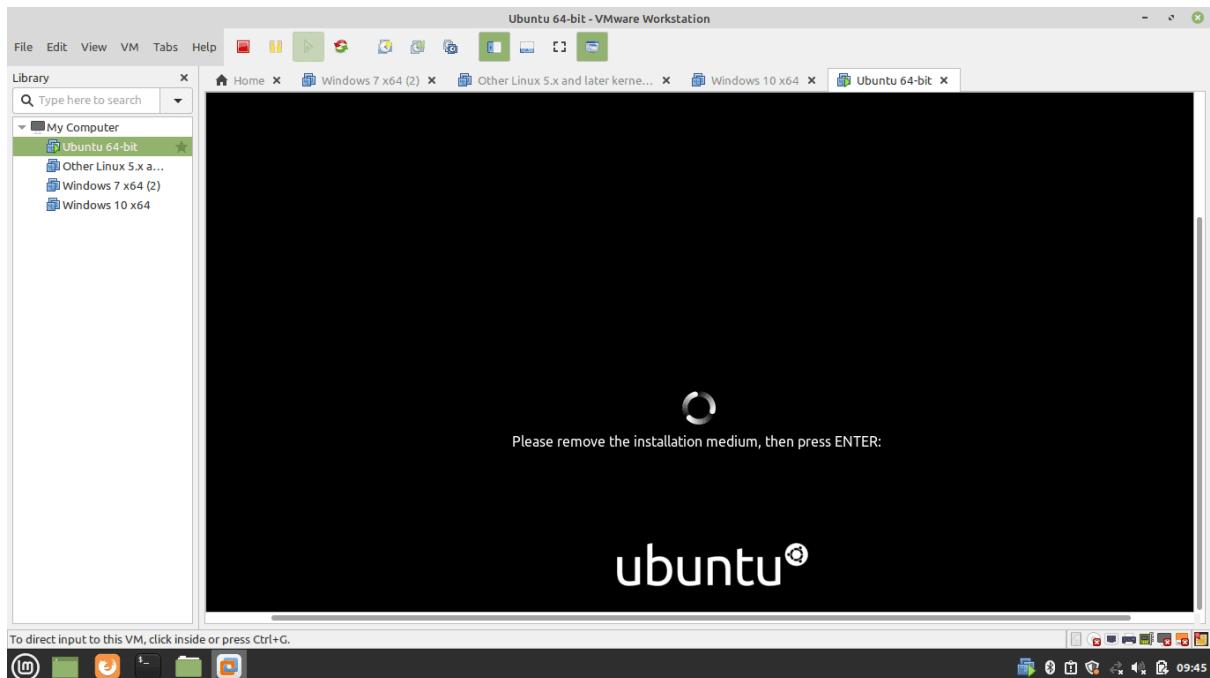
27. Reboot Virtual Machine

Reboot the machine after installation.

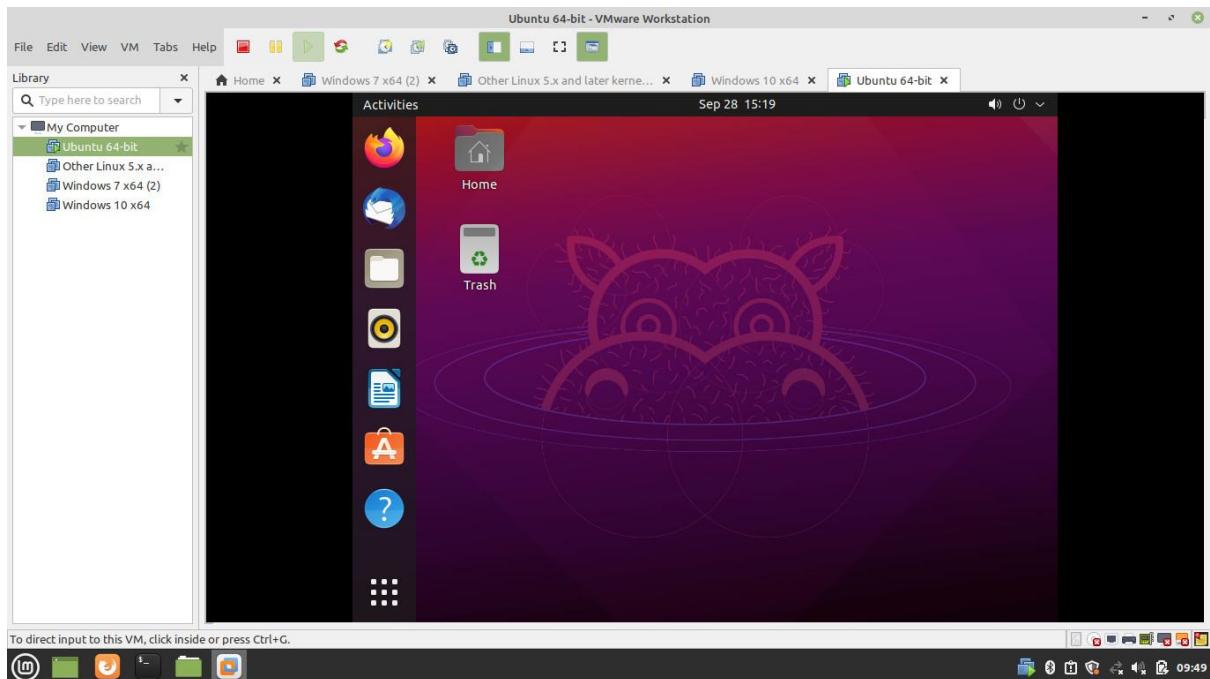


28. Remove the installation media

Remove the installation media before reboot.

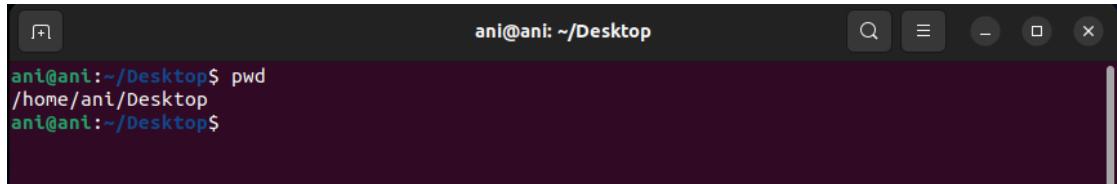


29. Boot Ubuntu



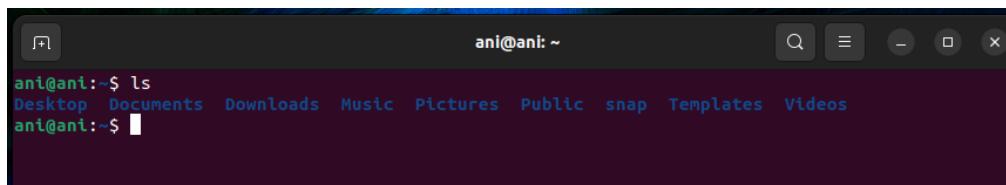
Basic 25 Unix shell commands:

1. **pwd** — When we first open the terminal, we are in the home directory of our user. To know which directory we are in, we can use the “**pwd**” command. It gives us the absolute path, which means the path that starts from the root. The root is the base of the Linux file system. It is denoted by a forward slash(/). The user directory is usually something like “/home/username”.



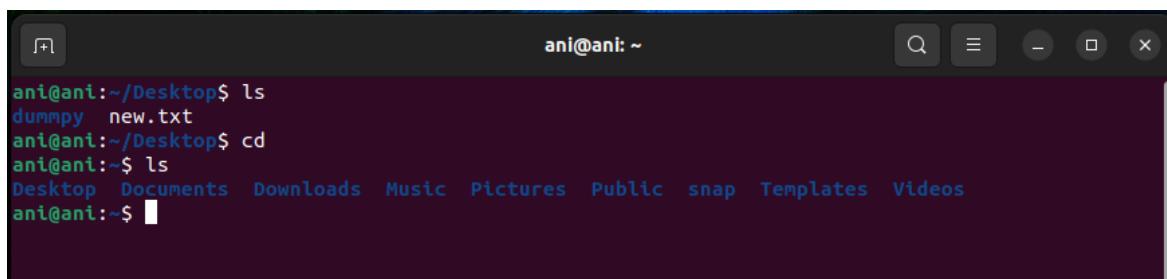
```
ani@ani:~/Desktop$ pwd
/home/ani/Desktop
ani@ani:~/Desktop$
```

2. **ls** — Use the “**ls**” command to know what files are in the directory we are in. We can see all the hidden files by using the command “**ls -a**”.



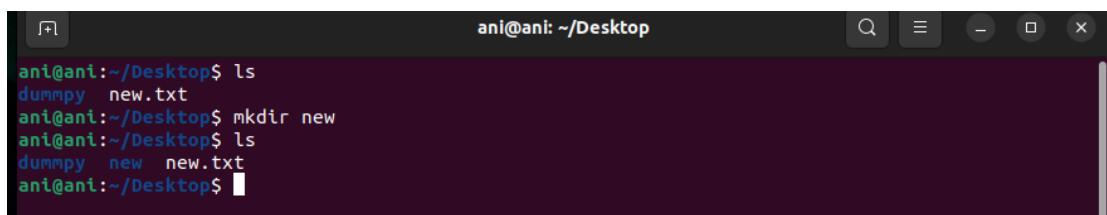
```
ani@ani:~$ ls
Desktop Documents Downloads Music Pictures Public snap Templates Videos
ani@ani:~$
```

3. **cd** — Use the “**cd**” command to go to a directory. For example, if we are in the home folder, and we want to go to the downloads folder, then we can type in “**cd Downloads**”. Remember, this command is case sensitive, and we have to type in the name of the folder exactly as it is.



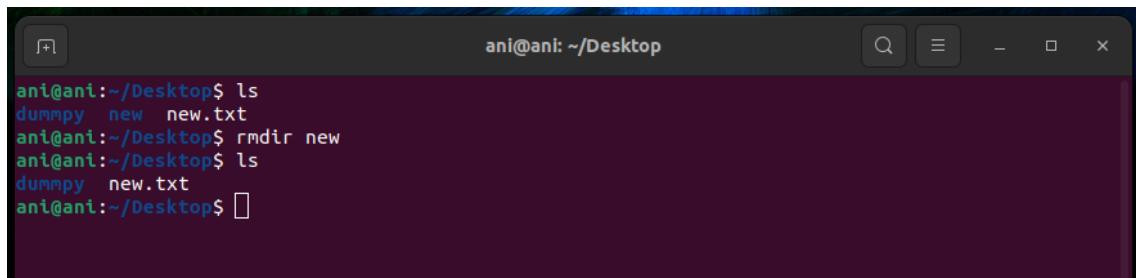
```
ani@ani:~/Desktop$ ls
dummpy new.txt
ani@ani:~/Desktop$ cd
ani@ani:~$ ls
Desktop Documents Downloads Music Pictures Public snap Templates Videos
ani@ani:~$
```

4. **mkdir** — Use the **mkdir** command when we need to create a folder or a directory. For example, if we want to make a directory called “DIY”, then we can type “**mkdir DIY**”. Remember, as told before, if we want to create a directory named “DIY Hacking”, then we can type “**mkdir DIY\ Hacking**”.



```
ani@ani:~/Desktop$ ls
dummpy new.txt
ani@ani:~/Desktop$ mkdir new
ani@ani:~/Desktop$ ls
dummpy new new.txt
ani@ani:~/Desktop$
```

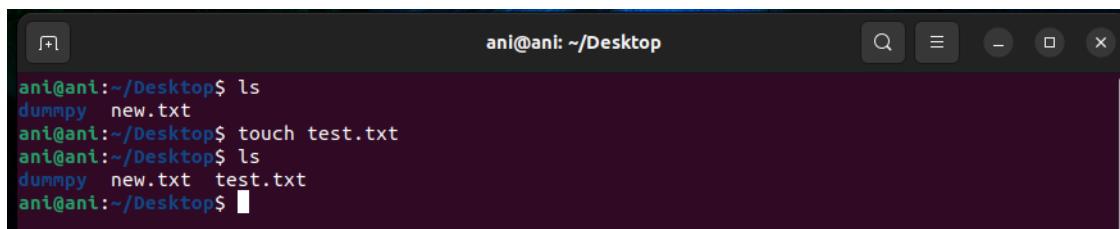
5. rmdir — Use rmdir to delete a directory. But rmdir can only be used to delete an empty directory. To delete a directory containing files, use rm.



A screenshot of a terminal window titled "ani@ani: ~/Desktop". The terminal shows the following session:

```
ani@ani:~/Desktop$ ls
dummpy new new.txt
ani@ani:~/Desktop$ rmdir new
ani@ani:~/Desktop$ ls
dummpy new.txt
ani@ani:~/Desktop$
```

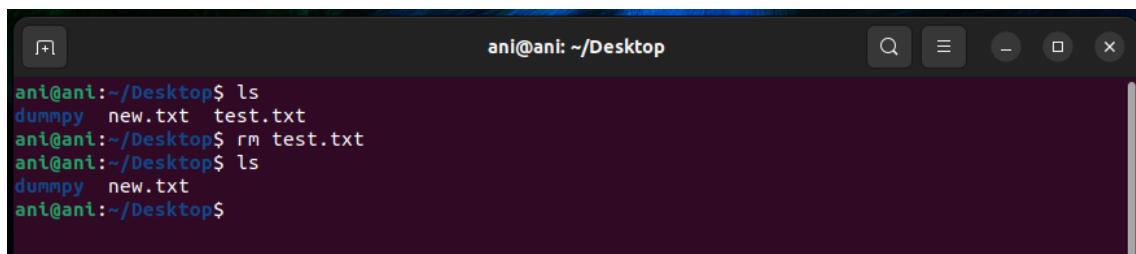
6. touch — The touch command is used to create a file. It can be anything, from an empty txt file to an empty zip file. For example, “touch new.txt”.



A screenshot of a terminal window titled "ani@ani: ~/Desktop". The terminal shows the following session:

```
ani@ani:~/Desktop$ ls
dummpy new.txt
ani@ani:~/Desktop$ touch test.txt
ani@ani:~/Desktop$ ls
dummpy new.txt test.txt
ani@ani:~/Desktop$
```

7. rm - Use the rm command to delete files and directories. Use "rm -r" to delete just the directory. It deletes both the folder and the files it contains when using only the rm command.



A screenshot of a terminal window titled "ani@ani: ~/Desktop". The terminal shows the following session:

```
ani@ani:~/Desktop$ ls
dummpy new.txt test.txt
ani@ani:~/Desktop$ rm test.txt
ani@ani:~/Desktop$ ls
dummpy new.txt
ani@ani:~/Desktop$
```

8. man — To know more about a command and how to use it, use the man command. It shows the manual pages of the command. For example, “man rm” shows the manual pages of the rm command.

```
ani@ani: ~/Desktop
User Commands
RM(1)

NAME
    rm - remove files or directories

SYNOPSIS
    rm [OPTION]... [FILE]...

DESCRIPTION
    This manual page documents the GNU version of rm. rm removes each
    specified file. By default, it does not remove directories.

    If the -I or --interactive=once option is given, and there are more
    than three files or the -r, -R, or --recursive are given, then rm
    prompts the user for whether to proceed with the entire operation. If
    the response is not affirmative, the entire command is aborted.

    Otherwise, if a file is unwritable, standard input is a terminal, and
    the -f or --force option is not given, or the -i or --interactive=al-
    ways option is given, rm prompts the user for whether to remove the
    file. If the response is not affirmative, the file is skipped.

OPTIONS
    Remove (unlink) the FILE(s).
```

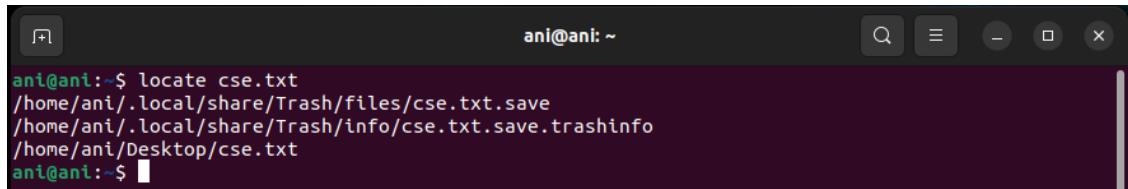
9. cp — Use the cp command to copy files through the command line. It takes two arguments: The first is the location of the file to be copied, the second is where to copy.

```
ani@ani:~$ ls
Desktop Documents Downloads Music Pictures Public snap Templates Videos
ani@ani:~$ cd Desktop
ani@ani:~/Desktop$ ls
dummpy new.txt
ani@ani:~/Desktop$ sudo cp new.txt /dummpy
[sudo] password for ani:
ani@ani:~/Desktop$ sudo ls /dummpy
/dummpy
ani@ani:~/Desktop$ s
```

10. mv — Use the mv command to move files through the command line. We can also use the mv command to rename a file. For example, if we want to rename the file “text” to “new”, we can use “mv text new”.

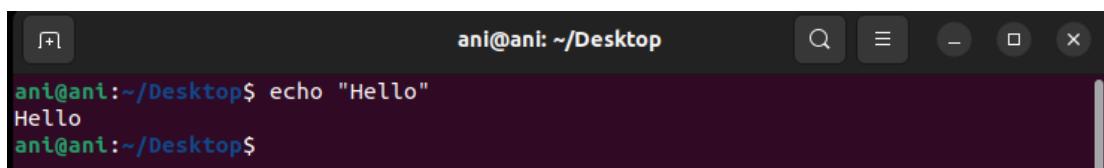
```
ani@ani:~/Desktop$ ls
cse.txt
ani@ani:~/Desktop$ mv cse.txt new.txt
ani@ani:~/Desktop$ ls
new.txt
ani@ani:~/Desktop$ s
```

11. locate — The locate command is used to locate a file in a Linux system, just like the search command in Windows. This command is useful when we don't know where a file is saved or the actual name of the file. Using the -i argument with the command helps to ignore the case (it doesn't matter if it is uppercase or lowercase). So, if we want a file that has the word "hello", it gives the list of all the files in our Linux system containing the word "hello" when we type in "locate -i hello"



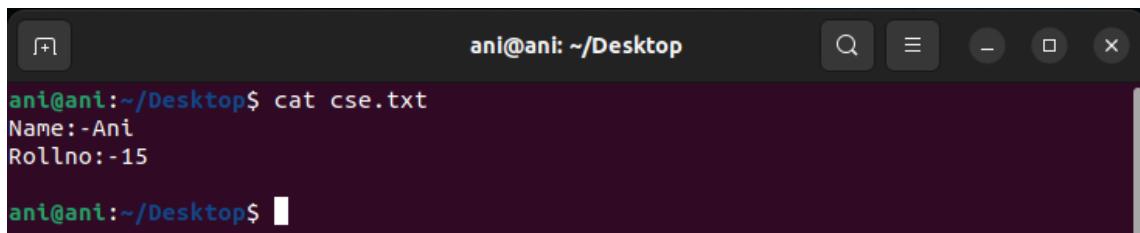
```
ani@ani:~$ locate cse.txt
/home/ani/.local/share/Trash/files/cse.txt.save
/home/ani/.local/share/Trash/info/cse.txt.save.trashinfo
/home/ani/Desktop/cse.txt
ani@ani:~$
```

12. echo — The "echo" command helps us move some data, usually text into a file.



```
ani@ani:~/Desktop$ echo "Hello"
Hello
ani@ani:~/Desktop$
```

13. cat — Use the cat command to display the contents of a file. It is usually used to easily view programs.

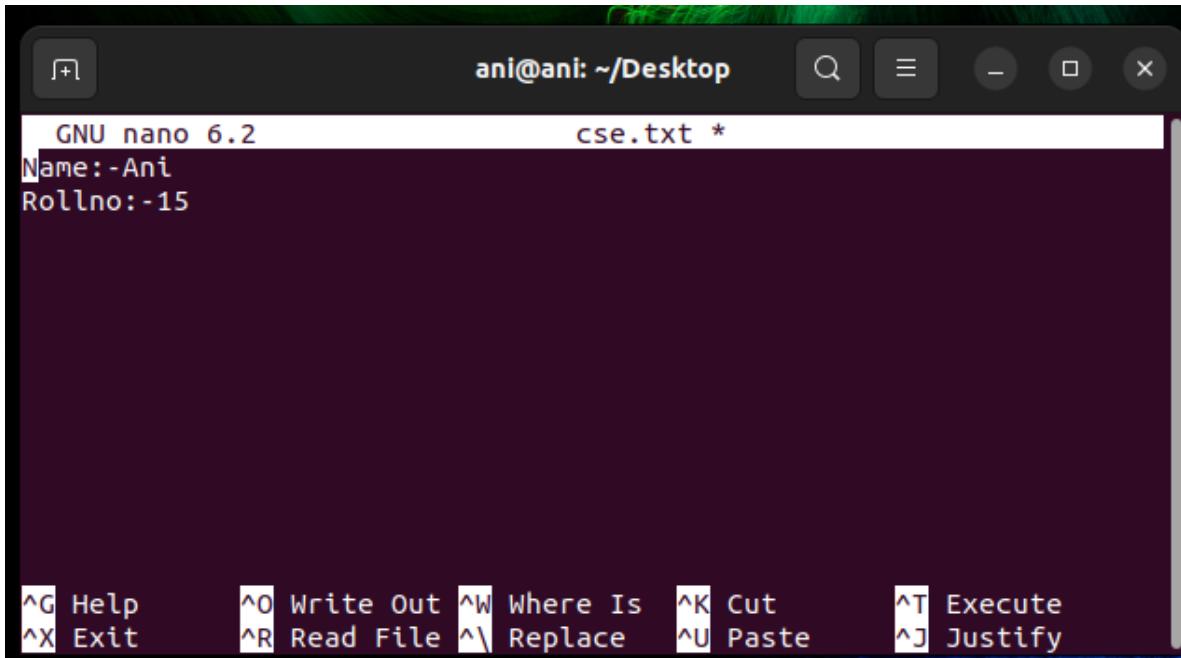


```
ani@ani:~/Desktop$ cat cse.txt
Name:-Ani
Rollno:-15

ani@ani:~/Desktop$
```

14. nano — nano is already installed text editors in the Linux command line.

The nano command is a good text editor that denotes keywords with color and can recognize most languages.



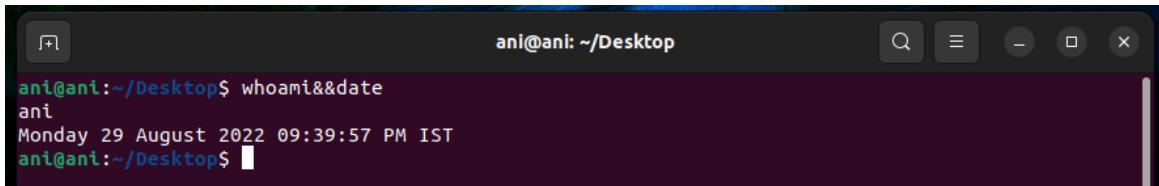
```
ani@ani: ~/Desktop
```

GNU nano 6.2 cse.txt *

Name:-Ani
Rollno:-15

^G Help ^O Write Out ^W Where Is ^K Cut ^T Execute
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify

15. and (&&): — and “and” command in linux which is used to combine two more commands

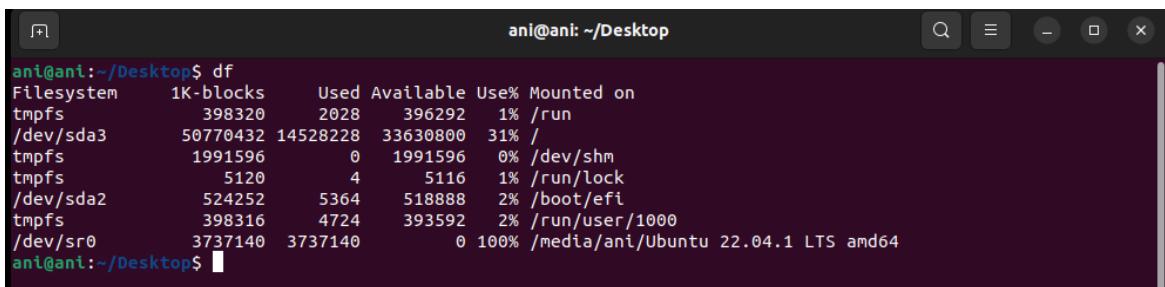


```
ani@ani: ~/Desktop$ whoami&&date
```

ani
Monday 29 August 2022 09:39:57 PM IST

```
ani@ani: ~/Desktop$
```

16. df — Use the df command to see the available disk space in each of the partitions in our system. We can just type in df in the command line and we can see each mounted partition and their used/available space in % and in KBs. If we want it shown in megabytes, we can use the command “df -m”.

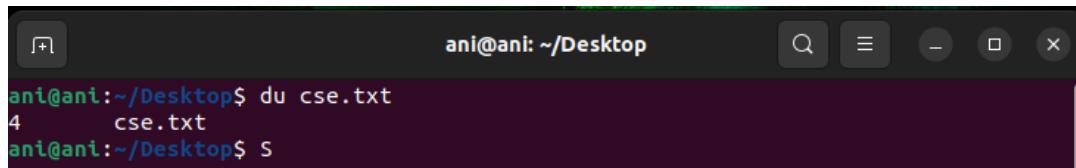


```
ani@ani: ~/Desktop$ df
```

Filesystem	1K-blocks	Used	Available	Use%	Mounted on
tmpfs	398320	2028	396292	1%	/run
/dev/sda3	50770432	14528228	33630800	31%	/
tmpfs	1991596	0	1991596	0%	/dev/shm
tmpfs	5120	4	5116	1%	/run/lock
/dev/sda2	524252	5364	518888	2%	/boot/efi
tmpfs	398316	4724	393592	2%	/run/user/1000
/dev/sr0	3737140	3737140	0	100%	/media/ani/Ubuntu 22.04.1 LTS amd64

```
ani@ani: ~/Desktop$
```

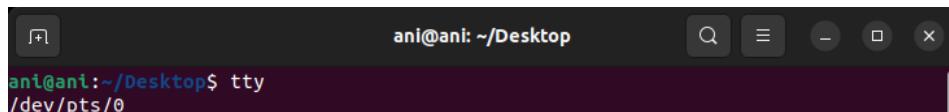
17. du — Use du to know the disk usage of a file in our system. If we want to know the disk usage for a particular folder or file in Linux, we can type in the command df and the name of the folder or file. For example, if we want to know the disk space used by the documents folder in Linux, we can use the command “du Documents”. We can also use the command “ls -lah” to view the file sizes of all the files in a folder.



A terminal window titled "ani@ani: ~/Desktop". The command "du cse.txt" is run, showing the output "4 cse.txt". The window has standard Linux-style title bar icons.

```
ani@ani:~/Desktop$ du cse.txt
4      cse.txt
ani@ani:~/Desktop$
```

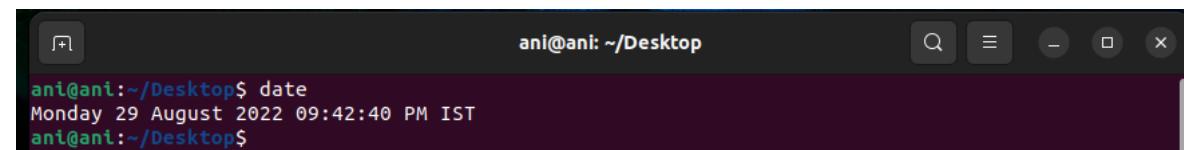
18. tty – Use tty to show the current display path name.



A terminal window titled "ani@ani: ~/Desktop". The command "tty" is run, showing the output "/dev/pts/0". The window has standard Linux-style title bar icons.

```
ani@ani:~/Desktop$ tty
/dev/pts/0
```

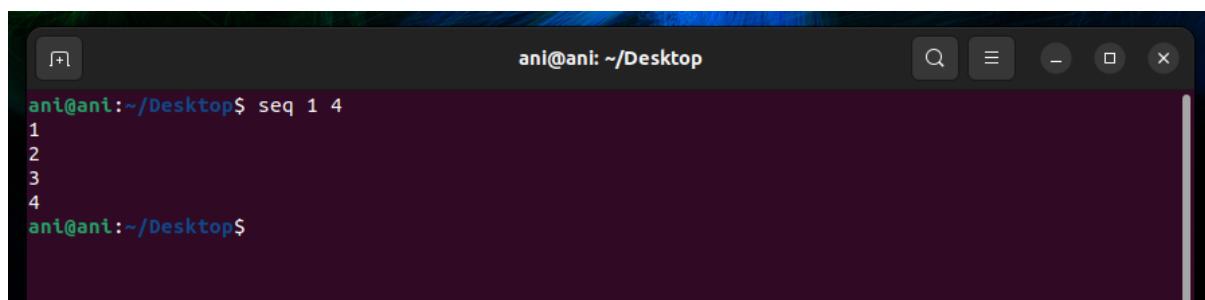
19.date: — Using date command we can see the system date and time in the given format.



A terminal window titled "ani@ani: ~/Desktop". The command "date" is run, showing the output "Monday 29 August 2022 09:42:40 PM IST". The window has standard Linux-style title bar icons.

```
ani@ani:~/Desktop$ date
Monday 29 August 2022 09:42:40 PM IST
ani@ani:~/Desktop$
```

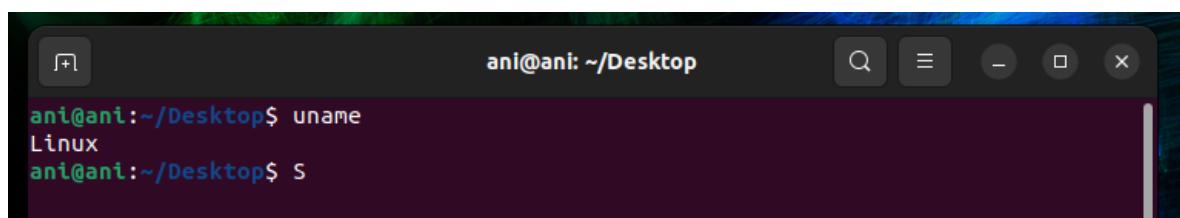
20.Seq – Using seq command in linux we can print a sequence in terminal using initial and final values given by us



A terminal window titled "ani@ani: ~/Desktop". The command "seq 1 4" is run, showing the output "1", "2", "3", and "4". The window has standard Linux-style title bar icons.

```
ani@ani:~/Desktop$ seq 1 4
1
2
3
4
ani@ani:~/Desktop$
```

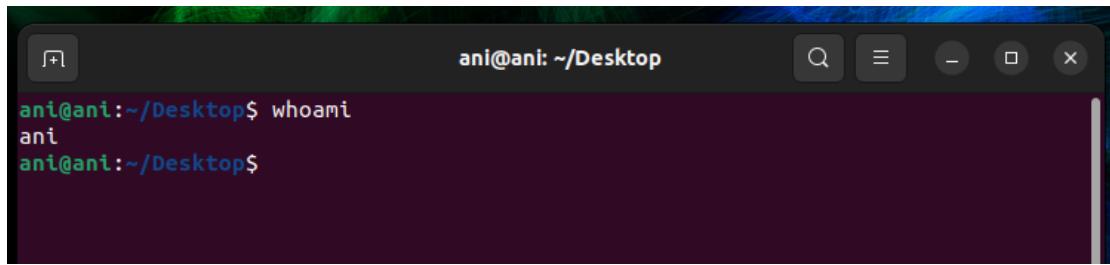
21. uname — Use uname to show the information about the system our Linux distro is running. Using the command “uname -a” prints most of the information about the system. This prints the kernel release date, version, processor type, etc.



A terminal window titled "ani@ani: ~/Desktop". The command "uname" is run, showing the output "Linux". The window has standard Linux-style title bar icons.

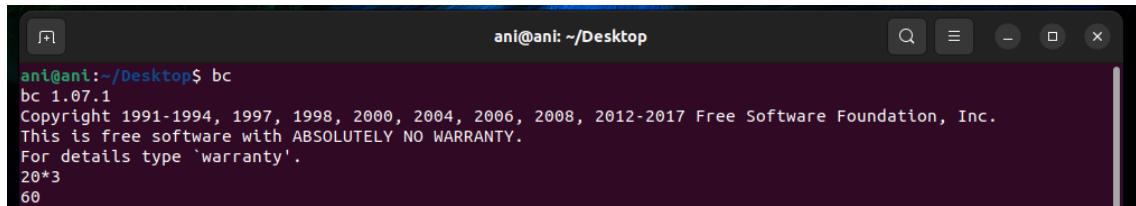
```
ani@ani:~/Desktop$ uname
Linux
ani@ani:~/Desktop$
```

22. Whoami: — using “whoami” command in linux we can find the name of current user.



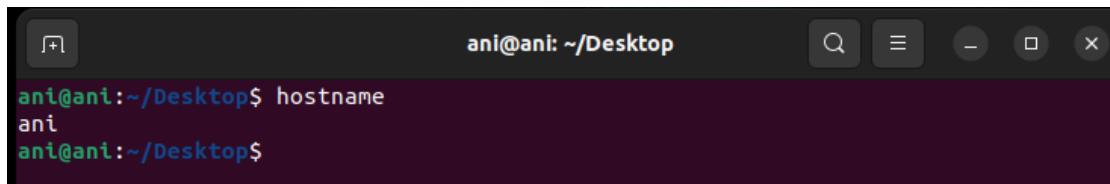
```
ani@ani:~/Desktop$ whoami
ani
ani@ani:~/Desktop$
```

23. Basic calculator- Using “bc” command in linux terminal we can access basic calculator



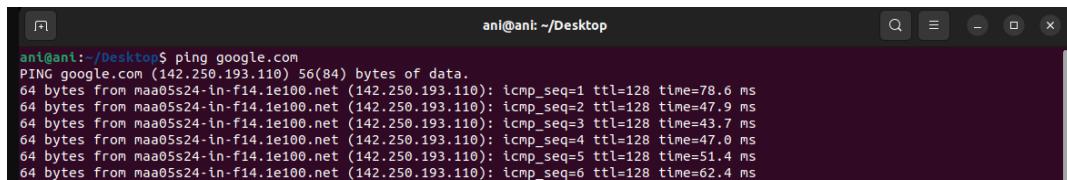
```
ani@ani:~/Desktop$ bc
bc 1.07.1
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
20*3
60
```

24. hostname — Using hostname gives us our name in our host or network. It displays our hostname and IP address. Typing “hostname” gives the output. Typing in “hostname -I” gives us our IP address in our network.



```
ani@ani:~/Desktop$ hostname
ani
ani@ani:~/Desktop$
```

25. ping — Use ping to check our connection to a server. when we type in, for example, “ping google.com”, it checks if it can connect to the server and come back. It measures this round-trip time and gives us the details about it. The use of this command for simple users like us is to check our internet connection. If it pings the Google server (in this case), we can confirm that our internet connection is active.



```
ani@ani:~/Desktop$ ping google.com
PING google.com (142.250.193.110) 56(84) bytes of data.
64 bytes from maa05s24-in-f14.1e100.net (142.250.193.110): icmp_seq=1 ttl=128 time=78.6 ms
64 bytes from maa05s24-in-f14.1e100.net (142.250.193.110): icmp_seq=2 ttl=128 time=47.9 ms
64 bytes from maa05s24-in-f14.1e100.net (142.250.193.110): icmp_seq=3 ttl=128 time=43.7 ms
64 bytes from maa05s24-in-f14.1e100.net (142.250.193.110): icmp_seq=4 ttl=128 time=47.0 ms
64 bytes from maa05s24-in-f14.1e100.net (142.250.193.110): icmp_seq=5 ttl=128 time=51.4 ms
64 bytes from maa05s24-in-f14.1e100.net (142.250.193.110): icmp_seq=6 ttl=128 time=62.4 ms
```

Result:

We have performed 25 unix shell commands in ubuntu terminal and recorded their output

Exp No: 2(a)

Name: PONNURI ANIRUDDHA

Date: 16/08/2022

Reg No: RA2112704010015

Process Creation using fork ()

and Usage of getpid (), getppid (), wait () functions

Aim: --

To create process using fork () and Usage of getpid (), getppid (), wait () functions.

Procedure: --

1)Fork (): -

fork () system call is used to create child processes in a C program. fork () is used where parallel processing is required in your application. The fork () system function is defined in the headers sys/types.h and unistd.h

2)getpid (): -

It returns the process ID (PID) of the calling process.

3)getppid (): -

It returns the process ID of the parent of the calling process.

4)Wait (): -

The wait () system call suspends execution of the calling thread until one of its children terminates.

Program: -

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

int main(int argc, char *argv[])
{
    printf("I am: %d\n", (int) getpid());
```

```

pid_t pid = fork();

printf("fork returned: %d\n", (int) pid);

if (pid < 0) {
    perror("Fork failed");
}

if (pid == 0) {
    printf("I am the child with pid %d\n", (int) getpid());
    printf("Child process is exiting\n");
    exit(0);
}

printf("I am the parent waiting for the child process to end\n");

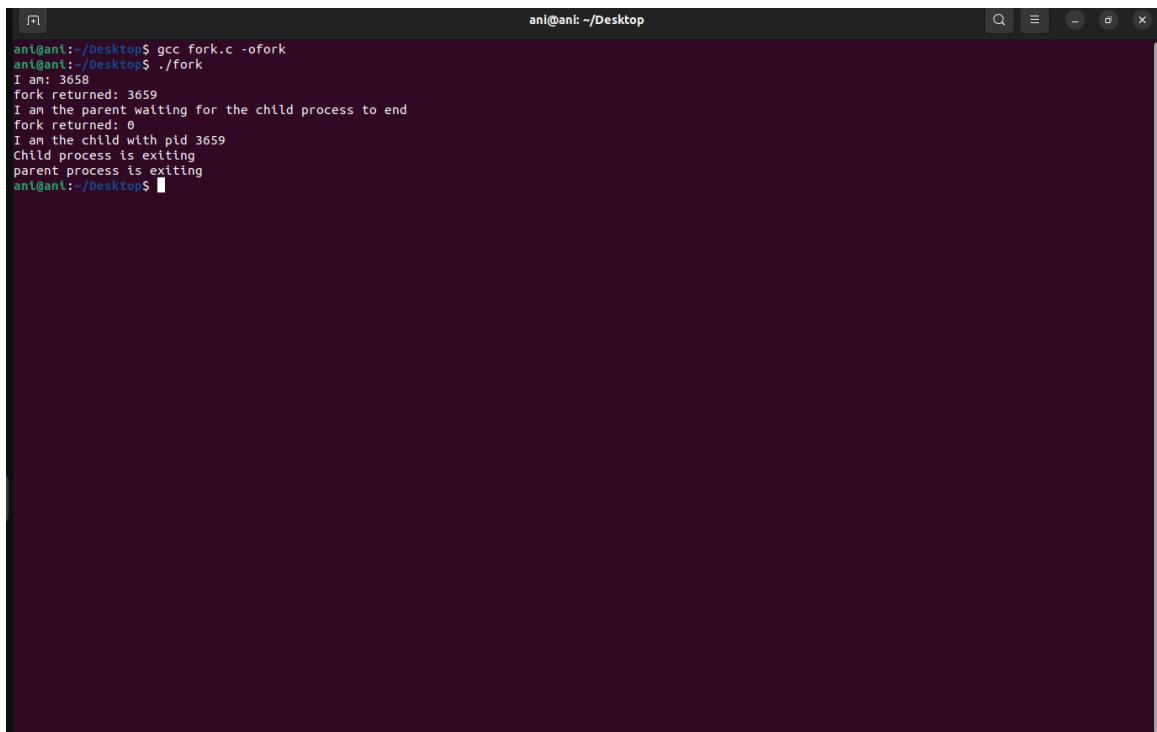
wait(NULL);

printf("parent process is exiting\n");

return(0);
}

```

OUTPUT: --



```

ani@ani:~/Desktop$ gcc fork.c -o fork
ani@ani:~/Desktop$ ./fork
I am: 3658
fork returned: 3659
I am the parent waiting for the child process to end
fork returned: 0
I am the child with pid 3659
Child process is exiting
parent process is exiting
ani@ani:~/Desktop$ 

```

Result: -

Thus, we have successfully created process using fork () and Usage of getpid (), getppid (), wait () functions.

Exp No: 2(b)

Name: PONNURI ANIRUDDHA

Date: 16/08/2022

Reg No: RA2112704010015

Zombie Process And Orphan Process

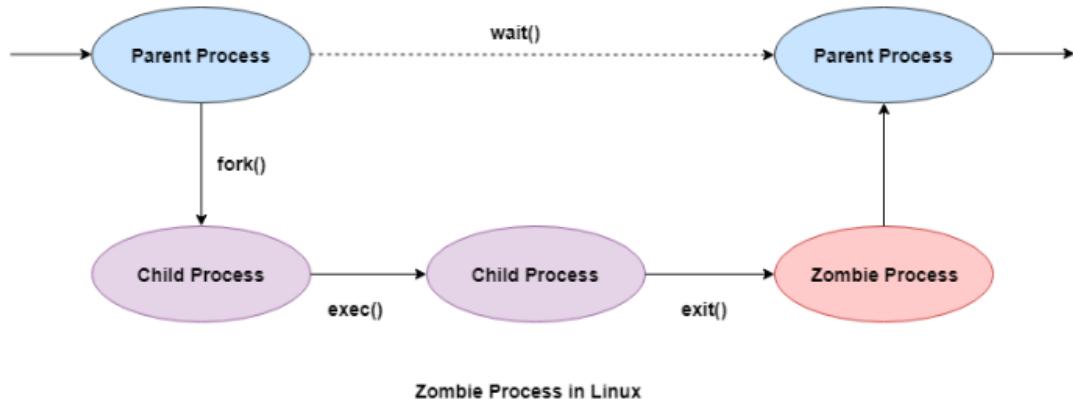
Aim: --

To create zombie and orphan process using c programming

Procedure: --

Zombie Process

A zombie process is a process whose execution is completed but it still has an entry in the process table. Zombie processes usually occur for child processes, as the parent process still needs to read its child's exit status. Once this is done using the wait system call, the zombie process is eliminated from the process table. This is known as reaping the zombie process



Orphan Process: --

A process whose parent process no more exists i.e., either finished or terminated without waiting for its child process to terminate is called an orphan process. Parent process finishes execution and exits while the child process is still executing and is called an orphan process.

Code (Zombie Process): --

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

int main() {
    int pid = fork();
    if (pid > 0) {
        sleep(20);
        printf("Parent process is exiting\n");
    }
    else {
        printf("Child process is exiting\n");
        printf("This is now considered as zombie process\n");
        exit(0);
    }
    return 0;
}
```

Output (Zombie Process): --

The terminal window shows the command `gcc zombie.c -o zombie` being run, followed by the output of the program itself. The output indicates that the child process sleeps for 20 seconds, then exits, marking it as a zombie process. The parent process then exits.

```
ani@ani:~/Desktop$ gcc zombie.c -o zombie
ani@ani:~/Desktop$ ./zombie
Child process is exiting
This is now considered as zombie process
Parent process is exiting
ani@ani:~/Desktop$
```

Code (Orphan Process): --

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

int main() {
    int pid;
    pid = fork();

    if(pid == 0)
    {
        printf("I am the child, my process ID is %d\n",getpid());
        printf("My parent's process ID is %d\n",getppid());
        sleep(2);
        printf("\nAfter sleep\nI am the child, my process ID is %d\n",getpid());
        printf("My parent's process ID is %d\n",getppid());
        exit(0);
    }
    else
    {
        sleep(3);
        printf("I am the parent, my process ID is %d\n",getpid());
        printf("Parent terminates\n");
    }
    return 0;
}
```

Output (Orphan Process): --



```
ani@ani:~/Desktop$ gcc chlld.c -ochild
ani@ani:~/Desktop$ ./child
I am the child, my process ID is 14347
my parent's process ID is 14346
I am the parent, my process ID is 14346
Parent terminates
ani@ani:~/Desktop$ After sleep
I am the child, my process ID is 14347
my parent's process ID is 1096
ani@ani:~/Desktop$ ani@ani:~/Desktop$
```

Result: -

Thus we have successfully created Zombie process and Orphan Process.

Exp No.3:

Date: 23/08/2022

Name: PONNURI ANIRUDDHA

Reg No: RA2112704010015

Multithreading and pthread in C

Aim: --

Creation of Multithreading using posix threads in C

Procedure: --

Thread:

A Thread is a single sequence stream within a process. In short thread is a unit of process.

Multithreading:

Multithreading is the ability of a program or an operating system to enable more than one user at a time without requiring multiple copies of the program running on the computer. Multithreading can also handle multiple requests from the same user.

POSIX threads:

The POSIX thread libraries are a C/C++ thread API based on standards. It enables the creation of a new concurrent process flow. It works well on multi-processor or multi-core systems, where the process flow may be scheduled to execute on another processor, increasing speed through parallel or distributed processing. Because the system does not create a new system, virtual memory space and environment for the process, threads need less overhead than “forking” or creating a new process. While multiprocessor systems are the most effective, benefits can also be obtained on uniprocessor systems that leverage delay in I/O and other system processes that may impede process execution

CODE: --

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

void *threadCreation(void *vargp)
{
    sleep(1);
    printf("A new thread is created \n");
}
```

```

    return NULL;
}

int main()
{
    pthread_t thread_id;
    printf("Before Thread\n");
    pthread_create(&thread_id, NULL, threadCreation, NULL);
    pthread_join(thread_id, NULL);
    printf("The Thread ID is of thread is %ld\n",thread_id);
    printf("After Thread\n");
    exit(0);
}

```

Output: --

```

ani@ani:~/Desktop$ gcc thread.c -o thread
ani@ani:~/Desktop$ ./thread
Before Thread
A new thread is created
The Thread ID is of thread is 140498575185472
After Thread
ani@ani:~/Desktop$ 

```

Result: --

A thread is created using Multithreading concept using posix threads.

Exp No 4:
Date:30/08/2022

Name: PONNURI ANIRUDDHA
Reg No: RA2112704010015

Mutual Exclusion using semaphore and monitor

Aim: –

Multithreading using Mutual exclusion using semaphore and monitor

Procedure: –

Thread synchronization

Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as a critical section. Processes' access to critical section is controlled by using synchronization techniques. When one thread starts executing the critical section (a serialized segment of the program) the other thread should wait until the first thread finishes. If proper synchronization techniques are not applied, it may cause a race condition where the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.

Mutex: --

1. A Mutex is a lock that we set before using a shared resource and release after using it.
2. When the lock is set, no other thread can access the locked region of code.
3. So, we see that even if thread 2 is scheduled while thread 1 was not done accessing the shared resource and the code is locked by thread 1 using mutexes then thread 2 cannot even access that region of code.
4. So, this ensures synchronized access of shared resources in the code.

Mutual Exclusion using semaphore

A semaphore is an integer variable that allows many processes in a parallel system to manage access to a common resource like a multitasking OS. It is an integer variable (S), and it is initialized with the number of resources in the system. The wait() and signal() methods are the only methods that may modify the semaphore (S) value. When one process modifies the semaphore value, other processes can't modify the semaphore value simultaneously.

Semaphores are an abstract entity provided by an operating system (not the hardware). Semaphores:

- are named by a unique semaphore id
- consist of a tuple (id, count, queue), where count is an integer and queue are a list of processes.

- a non-negative count always means that the queue is empty
 - a count of negative n indicates that the queue contains n waiting processes.
 - a count of positive n indicates that n resources are available and n requests can be granted without delay.
- sem = semcreate(val) -- creates a semaphore with the given initial value
- semdelete(sem) -- delete a semaphore
- wait(sem) -- decrement the semaphore count. if negative, suspend the process and place in queue. (Also referred to as P())
- signal(sem) -- increment the semaphore count, allow the first process in the queue to continue. (Also referred to as V())

Mutual Exclusion using Monitor

It is a synchronization technique that enables threads to mutual exclusion and the wait() for a given condition to become true. It is an abstract data type. It has a shared variable and a collection of procedures executing on the shared variable. A process may not directly access the shared data variables, and procedures are required to allow several processes to access the shared data variables simultaneously.

At any particular time, only one process may be active in a monitor. Other processes that require access to the shared variables must queue and are only granted access after the previous process releases the shared variables.

The monitor is made up of four primary parts:

- Initialization: The code for initialization is included in the package, and we just need it once when creating the monitors.
- Private Data: It is a feature of the monitor in an operating system to make the data private. It holds all of the monitor's secret data, which includes private functions that may only be utilized within the monitor. As a result, private fields and functions are not visible outside of the monitor.
- Monitor Procedure: Procedures or functions that can be invoked from outside of the monitor are known as monitor procedures.
- Monitor Entry Queue: Another important component of the monitor is the Monitor Entry Queue. It contains all of the threads, which are commonly referred to as procedures only.

Code (MUTEX): -

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

pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* threadFunction(void* arg)
{
    pthread_mutex_lock(&lock);

    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);

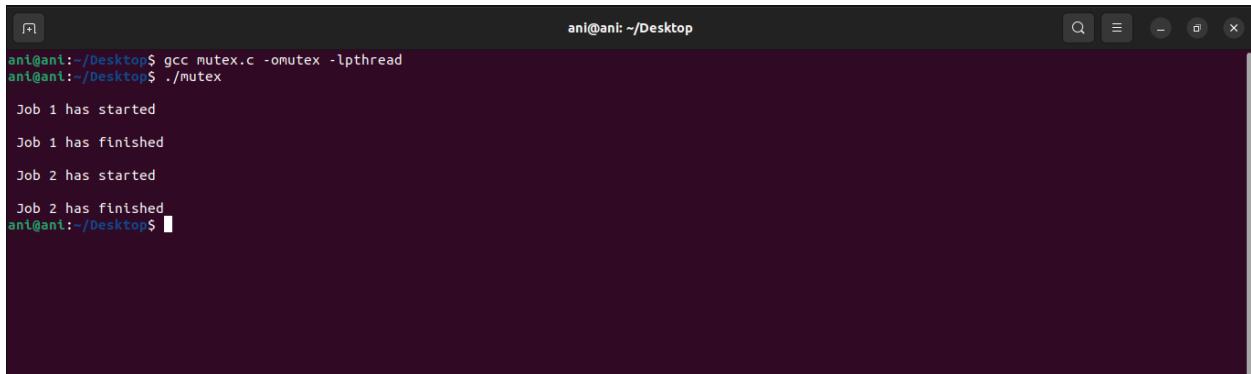
    pthread_mutex_unlock(&lock);

    return NULL;
}

int main(void){
    int i = 0;
    int err;
    while(i<2){
        err = pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);
        if(err != 0)
            printf("\ncan't create thread :[%os]", strerror(err));
        i++;
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
```

```
        return 0;  
    }
```

Output (MUTEX): --



The screenshot shows a terminal window titled "ani@ani: ~/Desktop". The terminal displays the following output:

```
ani@ani:~/Desktop$ gcc mutex.c -lpthread  
ani@ani:~/Desktop$ ./mutex  
Job 1 has started  
Job 1 has finished  
Job 2 has started  
Job 2 has finished  
ani@ani:~/Desktop$
```

Code (Semaphores): --

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

sem_t mutex;

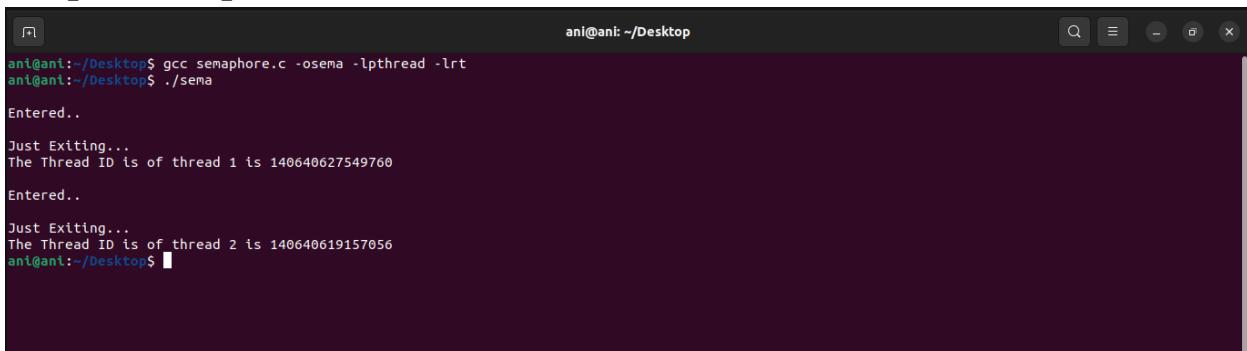
void* thread(void* arg)
{
    //wait
    sem_wait(&mutex);
    printf("\nEntered..\n");

    //critical section
    sleep(4);

    //signal
    printf("\nJust Exiting...\n");
    sem_post(&mutex);
}

int main()
{
    sem_init(&mutex, 0, 1);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    printf("The Thread ID is of thread 1 is %ld\n",t1);
    pthread_join(t2,NULL);
    printf("The Thread ID is of thread 2 is %ld\n",t2);
    sem_destroy(&mutex);
    return 0;
}
```

Output(Semaphores): --



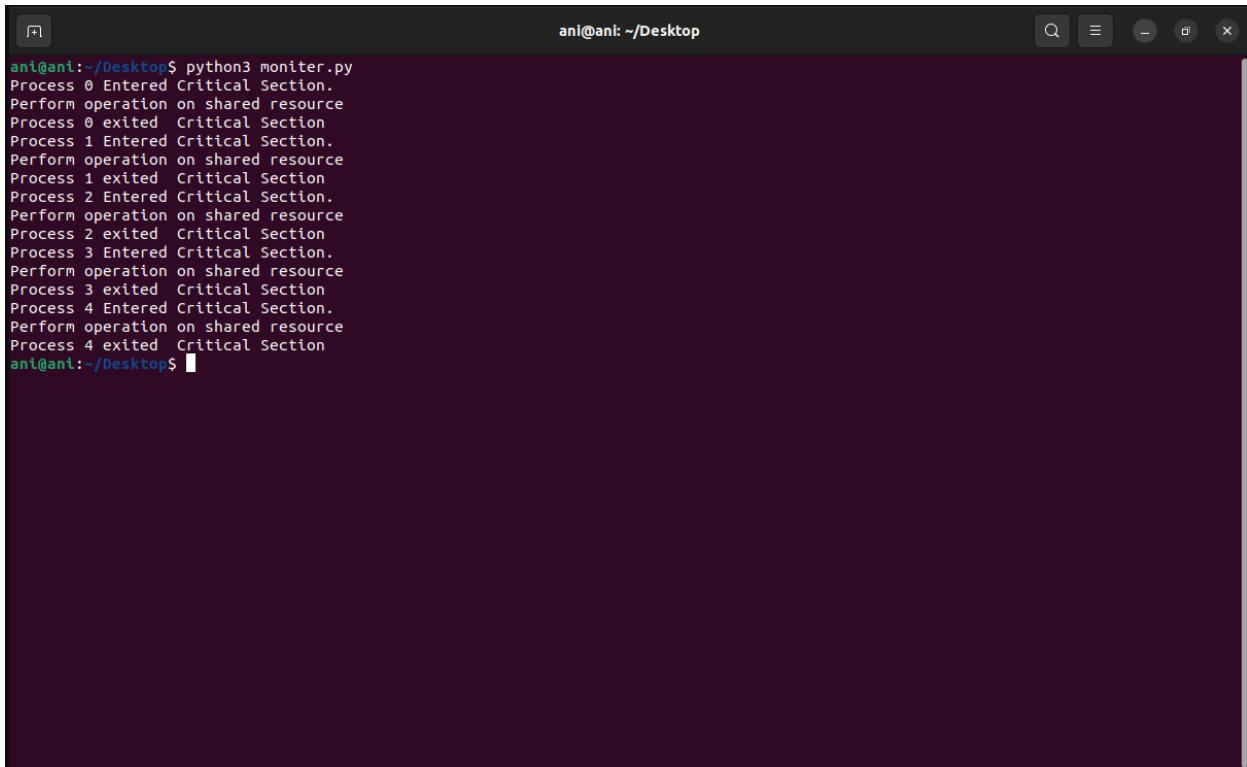
The screenshot shows a terminal window titled "ani@ani: ~/Desktop". The terminal displays the following output:

```
ani@ani:~/Desktop$ gcc semaphore.c -osema -lpthread -lrt
ani@ani:~/Desktop$ ./sema
Entered..
Just Exiting...
The Thread ID is of thread 1 is 140640627549760
Entered..
Just Exiting...
The Thread ID is of thread 2 is 140640619157056
ani@ani:~/Desktop$
```

Code (Monitors): --

```
import threading
import time
class testAndSet():
    #lock variable used in test and set is defined here
    Lock = 0
    def test(self,*args):
        if self.Lock == 0:
            # The critical section goes here...
            self.criticalsection(args[0])
        else:
            #while other process is executing current process is waiting
            while self.Lock == 1:
                print(f"Process {args[0]} waiting")
    def criticalsection(self,i):
        self.Lock = 1
        print(f"Process {i} Entered Critical Section.\nPerform operation on
              shared resource")
        #exit section
        self.Lock = 0
        print(f"Process {i} exited Critical Section")
    def main(self):
        t1 = threading.Thread(target = self.test, args = (0,))
        t1.start()
        t2 = threading.Thread(target = self.test, args = (1,))
        t2.start()
        t3 = threading.Thread(target = self.test, args = (2,))
        t3.start()
        t4 = threading.Thread(target = self.test, args = (3,))
        t4.start()
        t5 = threading.Thread(target = self.test, args = (4,))
        t5.start()
if __name__ == "__main__":
    t = testAndSet()
    t.main()
```

Output (Monitors): --



A terminal window titled "ani@ani: ~/Desktop" showing the execution of a Python script named "monitor.py". The script outputs log messages indicating the entry and exit of a critical section by five processes (0, 1, 2, 3, 4). The log shows each process entering the critical section, performing an operation on a shared resource, and then exiting the section.

```
ani@ani:~/Desktop$ python3 monitor.py
Process 0 Entered Critical Section.
Perform operation on shared resource
Process 0 exited  Critical Section
Process 1 Entered Critical Section.
Perform operation on shared resource
Process 1 exited  Critical Section
Process 2 Entered Critical Section.
Perform operation on shared resource
Process 2 exited  Critical Section
Process 3 Entered Critical Section.
Perform operation on shared resource
Process 3 exited  Critical Section
Process 4 Entered Critical Section.
Perform operation on shared resource
Process 4 exited  Critical Section
ani@ani:~/Desktop$
```

Result: -

Successfully executed Mutual Exclusion using semaphore and monitor

Exp No. 5

Date: 06/09/2022

Name: PONNURI ANIRUDDHA

Reg No: RA2112704010015

Reader-Writer problem

Aim:

Implementation of Reader-Writer problem

Procedure:

The readers-writers problem relates to an object such as a file that is shared between multiple processes. Some of these processes are readers i.e., they only want to read the data from the object and some of the processes are writers i.e., they want to write into the object. The readers-writers problem is used to manage synchronization so that there are no problems with the object data. For example - If two readers access the object at the same time there is no problem. However, if two writers or a reader and writer access the object at the same time, there may be problems.

To solve this situation, a writer should get exclusive access to an object i.e. when a writer is accessing the object, no reader or writer may access it. However, multiple readers can access the object at the same time.

Code: --

```
#include<semaphore.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
sem_t x,y;
pthread_t tid;
pthread_t writerthreads[100],readerthreads[100];
int readercount = 0;

void *reader(void* param)
{
    sem_wait(&x);
    readercount++;
}
```

```

if(readercount==1)
    sem_wait(&y);
    sem_post(&x);
    printf("%d reader is inside\n",readercount);
    usleep(3);
    sem_wait(&x);
    readercount--;
    if(readercount==0)
    {
        sem_post(&y);
    }
    sem_post(&x);
    printf("%d Reader is leaving\n",readercount+1);
    return NULL;
}

void *writer(void* param)
{
    printf("Writer is trying to enter\n");
    sem_wait(&y);
    printf("Writer has entered\n");
    sem_post(&y);
    printf("Writer is leaving\n");
    return NULL;
}

int main()
{
    int n2,i;

```

```

printf("Enter the number of readers:");
scanf("%d",&n2);
printf("\n");
int n1[n2];
sem_init(&x,0,1);
sem_init(&y,0,1);
for(i=0;i<n2;i++)
{
    pthread_create(&writerthreads[i],NULL,reader,NULL);
    pthread_create(&readerthreads[i],NULL,writer,NULL);
}
for(i=0;i<n2;i++)
{
    pthread_join(writerthreads[i],NULL);
    pthread_join(readerthreads[i],NULL);
}
}

```

Output: --

```

ani@ani:~/Desktop$ gcc rw.c
ani@ani:~/Desktop$ ./a.out
Enter the number of readers:2
1 reader is inside
Writer is trying to enter
2 reader is inside
2 Reader is leaving
Writer is trying to enter
Writer has entered
Writer has entered
Writer is leaving
Writer is leaving
1 Reader is leaving
1 Reader is leaving
ani@ani:~/Desktop$ 

```

Result: --

Thus we come to know about Implementation of Reader-Writer problem using c.

Exp No. 6

Date: 13/09/2022

Name: PONNURI ANIRUDDHA

Reg No: RA2112704010015

Dining Philosophers Problem

Aim: --

To implement Dinning philosopher's problem

Procedure: --

N philosophers, spend their time thinking and eating spaghetti. They eat at a round table with N individual seats. For eating each philosopher needs two forks (the resources). There are N forks on the table, one left and one right of each seat. When a philosopher cannot grab both forks it sits and waits. Eating takes random time, then the philosopher puts the forks down and leaves the dining room. After spending some random time thinking he again becomes hungry, and the circle repeats itself.

CODE: --

```
import threading  
import random  
import time  
  
#inheriting threading class in Thread module  
class Philosopher(threading.Thread):  
    running = True #used to check if everyone is finished eating  
    def __init__(self, index, forkOnLeft, forkOnRight):  
        threading.Thread.__init__(self)  
        self.index = index  
        self.forkOnLeft = forkOnLeft  
        self.forkOnRight = forkOnRight  
  
    def run(self):  
        while(self.running):  
            # Philosopher is thinking (but really is sleeping).  
            time.sleep(30)
```

```

print ('Philosopher %s is hungry.' % self.index)
self.dine()

def dine(self):
    # if both forks are free, then philosopher will eat
    fork1, fork2 = self.forkOnLeft, self.forkOnRight
    while self.running:
        fork1.acquire() # wait operation on left fork
        locked = fork2.acquire(False)
        if locked: break #if right fork is not available leave left fork
        fork1.release()
        print ('Philosopher %s swaps forks.' % self.index)
        fork1, fork2 = fork2, fork1
    else:
        return
    self.dining()
    #release both the fork after dining
    fork2.release()
    fork1.release()

def dining(self):
    print ('Philosopher %s starts eating. "% self.index)
    time.sleep(30)
    print ('Philosopher %s finishes eating and leaves to think.' % self.index)

def main():
    forks = [threading.Semaphore() for n in range(5)] #initialising array of semaphore
    i.e forks

```

```

#here (i+1)%5 is used to get right and left forks circularly between 1-5
philosophers= [Philosopher(i, forks[i%5], forks[(i+1)%5])
               for i in range(5)]

Philosopher.running = True
for p in philosophers: p.start()
time.sleep(100)
Philosopher.running = False
print ("Now we're finishing.")

if __name__ == "__main__":
    main()

```

Output: --

```

ant@ani:~/Desktop$ python3 dine.py
Philosopher 0 is hungry.
Philosopher 0 starts eating.
Philosopher 1 is hungry.
Philosopher 2 is hungry.
Philosopher 2 starts eating.
Philosopher 3 is hungry.
Philosopher 4 is hungry.
Philosopher 4 swaps forks.
Philosopher 2 finishes eating and leaves to think.
Philosopher 3 starts eating.
Philosopher 0 finishes eating and leaves to think.
Philosopher 4 swaps forks.
Philosopher 1 starts eating.
Philosopher 3 finishes eating and leaves to think.
Philosopher 2 is hungry.
Philosopher 4 starts eating.
Philosopher 0 is hungry.
Philosopher 1 finishes eating and leaves to think.
Philosopher 2 starts eating.
Now we're finishing.

```

Result: --

We have successfully implemented dining philosopher's problem in python

Exp No. 7

Date: 19/09/2022

Name: PONNURI ANIRUDDHA

Reg No: RA2112704010015

Bankers Algorithm

Aim: --

To implement Banker Algorithm

Procedure: --

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an “s-state” check to test for possible activities, before deciding whether allocation should be allowed to continue.

Banker's algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan, then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders come to withdraw their money, then the bank can easily do it. In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

CODE: --

```
# P0, P1, P2, P3, P4 are the Process names here
n = 5 # Number of processes
m = 3 # Number of resources
# Allocation Matrix
alloc = [[0, 1, 0 ],[ 2, 0, 0 ],[3, 0, 2 ],[2, 1, 1],[ 0, 0, 2 ]]
# MAX Matrix
max = [[7, 5, 3 ],[3, 2, 2 ],[ 9, 0, 2 ],[2, 2, 2],[4, 3, 3 ]]
avail = [3, 3, 2] # Available Resources
```

$$f = [0]^*n$$

$$ans = [0]^*n$$

$$ind = 0$$

for k in range(n):

f[k] = 0

need = [[0 for i in range(m)] for i in range(n)]

for i in range(n):

for j in range(m):

need[i][j] = max[i][j] - alloc[i][j]

y = 0

for k in range(5):

for i in range(n):

if (f[i] == 0):

flag = 0

for j in range(m):

if (need[i][j] > avail[j]):

flag = 1

break

if (flag == 0):

ans[ind] = i

ind += 1

for y in range(m):

avail[y] += alloc[i][y]

f[i] = 1

print("Following is the SAFE Sequence")

for i in range(n - 1):

print(" P", ans[i], " ->", sep="", end="")

print(" P", ans[n - 1], sep="")

OUTPUT: --



```
ani@ani:~/Desktop$ python3 banker.py
Following is the SAFE Sequence
P1 -> P3 -> P4 -> P0 -> P2
ani@ani:~/Desktop$
```

the system will skip process 0 and move to process 1. Because each process holds some resource and if the resource is not enough, the process can't be completed. Since the system doesn't have enough resources for process 0, it cannot complete that process. Hence the deadlock condition occurs. Once a process is completed, the system will regain the allocated resources. AS a result, there will be enough resources to allocate for upcoming processes and deadlock conditions can be avoided. Therefore, the system allocates resources in the order P1, P3, P4, P0, P2.

RESULT: --

We have successfully implemented Banker's algorithm.

Exp No. 8

Name: PONNURI ANIRUDDHA

Date: 26/09/2022

Reg No: RA2112704010015

FCFS and SJF Scheduling

Aim: --

To implement FCFS (First Come First Serve) and SJF (shortest job first) in CPU scheduling.

Procedure: --

FCFS stands for First Come First Serve. In the FCFS scheduling algorithm, the job that arrived first in the ready queue is allocated to the CPU and then the job that came second, and so on. We can say that the ready queue acts as a FIFO (First In First Out) queue thus the arriving jobs/processes are placed at the end of the queue. FCFS is a non-preemptive scheduling algorithm as a process holds the CPU until it either terminates or performs I/O. Thus, if a longer job has been assigned to the CPU then many shorter jobs after it will have to wait. This algorithm is used in most batch operating systems.

The waiting time is the time which the second process is kept in the waiting state so that the first process is moved to the running state and executed. The burst time is the time for execution on the CPU. The turnaround time is the time in which the request for any request has been fulfilled or a job is completed. The completion time is the total time in which the process completes execution from its ready state to the executed state. The algorithms that the FCFS works are as follows:

$$\text{Turn Around Time} = \text{Completion Time} - \text{Arrival Time}$$

$$\text{Waiting Time} = \text{Turnaround time} - \text{Burst Time}$$

Algorithm:

- The first process arrives at the CPU.
- The completion time is calculated so that the waiting time for the next processes are calculated.
- The next processes enter/arrive at the CPU.
- While the process 1 is completing its execution the rest of the processes are kept for wait.
- The turnaround time is calculated by subtracting completion time of the process with the arrival time at the CPU.
- The total waiting time is calculated by subtracting the turnaround time and the burst time.

Shortest Job First (SJF):

Shortest job first (SJF) is a scheduling process that selects the waiting process with the smallest execution time to execute next. This scheduling method may or may not be preemptive. Significantly reduces the average waiting time for other processes waiting to be executed. The full form of SJF is Shortest Job First.

Algorithm:

- Sort all the processes according to the arrival time.
- Then select that process that has minimum arrival time and minimum Burst time.
- After completion of the process make a pool of processes that arrives afterward till the completion of the previous process and select that process among the pool which is having minimum Burst time.

Code (FCFS):

```
def findWaitingTime(processes, n, bt, wt, at):  
    service_time = [0] * n  
    service_time[0] = 0  
    wt[0] = 0  
  
    # calculating waiting time  
    for i in range(1, n):  
  
        # Add burst time of previous processes  
        service_time[i] = (service_time[i - 1] +  
                           bt[i - 1])  
  
        # Find waiting time for current  
        # process = sum - at[i]  
        wt[i] = service_time[i] - at[i]  
  
        # If waiting time for a process is in  
        # negative that means it is already  
        # in the ready queue before CPU becomes
```

```

# idle so its waiting time is 0
if (wt[i] < 0):
    wt[i] = 0

# Function to calculate turn around time
def findTurnAroundTime(processes, n, bt, wt, tat):

    # Calculating turnaround time by
    # adding bt[i] + wt[i]
    for i in range(n):
        tat[i] = bt[i] + wt[i]

# Function to calculate average waiting
# and turn-around times.
def findavgTime(processes, n, bt, at):
    wt = [0] * n
    tat = [0] * n

    # Function to find waiting time
    # of all processes
    findWaitingTime(processes, n, bt, wt, at)

    # Function to find turn around time for
    # all processes
    findTurnAroundTime(processes, n, bt, wt, tat)

    # Display processes along with all details
    print("Processes  Burst Time  Arrival Time  Waiting",
          "Time  Turn-Around Time  Completion Time \n")
    total_wt = 0

```

```

total_tat = 0

for i in range(n):

    total_wt = total_wt + wt[i]

    total_tat = total_tat + tat[i]

    compl_time = tat[i] + at[i]

    print(" ", i + 1, "\t\t", bt[i], "\t\t", at[i],
          "\t\t", wt[i], "\t\t", tat[i], "\t\t", compl_time)

print("Average waiting time = %.5f %(total_wt /n)")

print("\nAverage turn around time = ", total_tat / n)

# Driver code

if __name__ == "__main__":
    # Process id's
    processes = [1, 2, 3]
    n = 3

    # Burst time of all processes
    burst_time = [5, 9, 6]

    # Arrival time of all processes
    arrival_time = [0, 3, 6]

    findavgTime(processes, n, burst_time, arrival_time)

```

OUTPUT (FCFS): --

```
ani@ani:~/Desktop$ python3 fcfs.py
Processes    Burst Time    Arrival Time    Waiting Time    Turn-Around Time    Completion Time
 1            5                  0                0                  5                  5
 2            9                  3                2                 11                 14
 3            6                  6                8                 14                 20
Average waiting time = 3.33333
Average turn around time = 10.0
ani@ani:~/Desktop$ S
```

Code (SJF): --

```
def findWaitingTime(processes, n, wt):
    rt = [0] * n

    # Copy the burst time into rt[]
    for i in range(n):
        rt[i] = processes[i][1]

    complete = 0
    t = 0
    minm = 999999999
    short = 0
    check = False

    # Process until all processes gets
    # completed
    while (complete != n):

        # Find process with minimum remaining
        # time among the processes that
        # arrives till the current time
        for j in range(n):
            if ((processes[j][2] <= t) and
                (rt[j] < minm) and rt[j] > 0):
                minm = rt[j]
                short = j
                check = True

        if (check == False):
            t += 1
            continue

        rt[short] -= 1
        if (rt[short] == 0):
            complete += 1
            print("Process ", short, " completed at time ", t)
            print("Waiting time of ", short, " is ", t - processes[short][1])
            print("Remaining burst time of ", short, " is ", rt[short])
```

```

# Reduce remaining time by one
rt[short] -= 1

# Update minimum
minm = rt[short]
if (minm == 0):
    minm = 999999999

# If a process gets completely
# executed
if (rt[short] == 0):

# Increment complete
complete += 1
check = False

# Find finish time of current
# process
fint = t + 1

# Calculate waiting time
wt[short] = (fint - proc[short][1] -
            proc[short][2])

if (wt[short] < 0):
    wt[short] = 0

# Increment time

```

```

t += 1

# Function to calculate turn around time
def findTurnAroundTime(processes, n, wt, tat):

    # Calculating turnaround time
    for i in range(n):
        tat[i] = processes[i][1] + wt[i]

# Function to calculate average waiting
# and turn-around times.
def findavgTime(processes, n):
    wt = [0] * n
    tat = [0] * n

    # Function to find waiting time
    # of all processes
    findWaitingTime(processes, n, wt)

    # Function to find turn around time
    # for all processes
    findTurnAroundTime(processes, n, wt, tat)

# Display processes along with all details
print("Processes   Burst Time   Waiting",
      "Time   Turn-Around Time")

total_wt = 0
total_tat = 0
for i in range(n):

```

```

total_wt = total_wt + wt[i]
total_tat = total_tat + tat[i]
print(" ", processes[i][0], " ",
      processes[i][1], " ",
      wt[i], " ", tat[i])

print("Average waiting time = %.5f %%(total_wt /n) )
print("Average turn around time = ", total_tat /n)

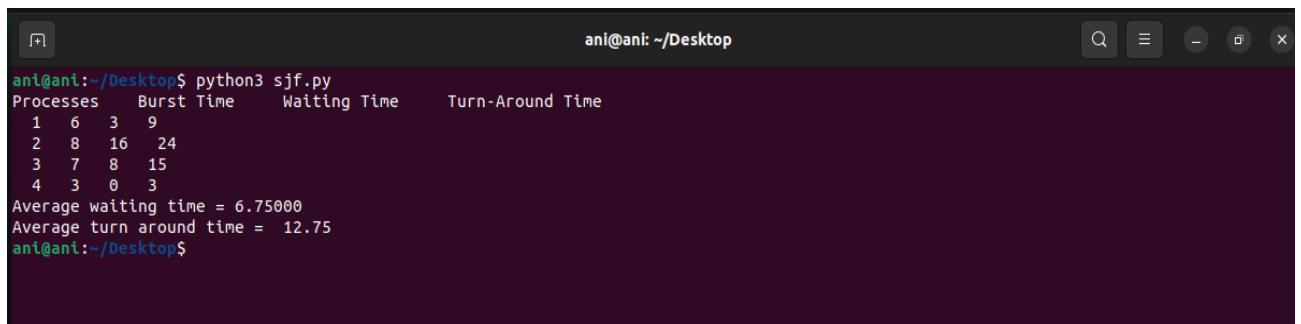
```

```

# Driver code
if __name__ == "__main__":
    # Process id's
    proc = [[1, 6, 1], [2, 8, 1],
            [3, 7, 2], [4, 3, 3]]
    n = 4
    findavgTime(proc, n)

```

OUTPUT (SJF): --



```

ant@ani:~/Desktop$ python3 sjf.py
Processes      Burst Time      Waiting Time     Turn-Around Time
 1      6      3      9
 2      8     16     24
 3      7      8     15
 4      3      0      3
Average waiting time = 6.75000
Average turn around time = 12.75
ant@ani:~/Desktop$
```

Result:

The code has been implemented for FCFS (First come first serve) type of CPU Scheduling and SJF (Shortest Job First) type of CPU Scheduling.

Exp No. 9

Name: PONNURI ANIRUDDHA

Date:03/10/2022

Reg No: RA2112704010015

Priority and Round robin scheduling

Aim: --

To implement Priority Type and Round Robin type of CPU Scheduling

Procedure: --

Priority CPU Scheduling

Priority scheduling is one of the most common scheduling algorithms in batch systems. Each process is assigned a priority. Process with the highest priority is to be executed first and so on. Processes with the same priority are executed on first come first served basis. Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Round Robin CPU Scheduling: --

Round Robin is a CPU scheduling algorithm where each process is cyclically assigned a fixed time slot. It is the preemptive version of First come First Serve CPU Scheduling algorithm. Round Robin CPU Algorithm generally focuses on Time Sharing technique. The period of time for which a process or job is allowed to run in a pre-emptive method is called time quantum. Each process or job present in the ready queue is assigned the CPU for that time quantum, if the execution of the process is completed during that time then the process will end else the process will go back to the waiting table and wait for its next turn to complete the execution.

Code (Priority Scheduling): --

```
def findWaitingTime(processes, n, wt):  
    wt[0] = 0  
  
    # calculating waiting time  
  
    for i in range(1, n):  
  
        wt[i] = processes[i - 1][1] + wt[i - 1]  
  
  
    # Function to calculate turn around time  
  
    def findTurnAroundTime(processes, n, wt, tat):  
  
  
        # Calculating turnaround time by  
  
        # adding bt[i] + wt[i]  
  
        for i in range(n):  
  
            tat[i] = processes[i][1] + wt[i]  
  
  
    # Function to calculate average waiting  
  
    # and turn-around times.  
  
    def findavgTime(processes, n):  
  
        wt = [0] * n  
  
        tat = [0] * n  
  
  
        # Function to find waiting time  
  
        # of all processes  
  
        findWaitingTime(processes, n, wt)
```

```

# Function to find turn around time

# for all processes

findTurnAroundTime(processes, n, wt, tat)

# Display processes along with all details

print("\nProcesses Burst Time Waiting","Time Turn-Around Time")

total_wt = 0

total_tat = 0

for i in range(n):

    total_wt = total_wt + wt[i]

    total_tat = total_tat + tat[i]

    print(" ", processes[i][0], "\t\t",

          processes[i][1], "\t\t",

          wt[i], "\t\t", tat[i])

print("\nAverage waiting time = %.5f %(total_wt / n))

print("Average turn around time = ", total_tat / n)

def priorityScheduling(proc, n):

    # Sort processes by priority

```

```

proc = sorted(proc, key = lambda proc:proc[2], reverse = True);

print("Order in which processes gets executed")

for i in proc:

    print(i[0], end = " ")

findavgTime(proc, n)

# Driver code

if __name__ == "__main__":
    pass

# Process id's

proc = [[1, 10, 1],
        [2, 5, 0],
        [3, 8, 1]]

n = 3

priorityScheduling(proc, n)

```

OUTPUT (Priority Scheduling): --

```

ani@ani:~/Desktop$ python3 pr.py
Order in which processes gets executed
1 3 2
Processes Burst Time Waiting Time Turn-Around Time
 1      10       0      10
 3       8      10      18
 2       5      18      23

Average waiting time = 9.33333
Average turn around time = 17.0
ani@ani:~/Desktop$ 

```

CODE (Round Robin CPU Scheduling): --

```
def findWaitingTime(processes, n, bt, wt, quantum):  
    rem_bt = [0] * n  
  
    # Copy the burst time into rt[]  
  
    for i in range(n):  
        rem_bt[i] = bt[i]  
  
    t = 0 # Current time  
  
    # Keep traversing processes in round  
    # robin manner until all of them are  
    # not done.  
  
    while(1):  
        done = True  
  
        # Traverse all processes one by  
        # one repeatedly  
  
        for i in range(n):  
  
            # If burst time of a process is greater  
            # than 0 then only need to process further  
  
            if (rem_bt[i] > 0) :  
  
                done = False # There is a pending process  
  
                # Decrease burst time by quantum  
                rem_bt[i] -= quantum  
  
                # If burst time left is more than  
                # quantum then set it to quantum  
                # otherwise set it to burst time left  
                if (rem_bt[i] < quantum):  
                    quantum = rem_bt[i]  
  
                # Increase waiting time of current  
                # process by one  
                wt[i] += 1
```

```

if (rem_bt[i] > quantum) :

    # Increase the value of t i.e. shows
    # how much time a process has been processed

    t += quantum

    # Decrease the burst_time of current
    # process by quantum

    rem_bt[i] -= quantum

# If burst time is smaller than or equal
# to quantum. Last cycle for this process

else:

    # Increase the value of t i.e. shows
    # how much time a process has been processed

    t = t + rem_bt[i]

    # Waiting time is current time minus
    # time used by this process

    wt[i] = t - bt[i]

# As the process gets fully executed
# make its remaining burst time = 0

```

```

rem_bt[i] = 0

# If all processes are done

if (done == True):

    break

# Function to calculate turn around time

def findTurnAroundTime(processes, n, bt, wt, tat):

# Calculating turnaround time

for i in range(n):

    tat[i] = bt[i] + wt[i]

# Function to calculate average waiting

# and turn-around times.

def findavgTime(processes, n, bt, quantum):

    wt = [0] * n

    tat = [0] * n

# Function to find waiting time

# of all processes

findWaitingTime(processes, n, bt, wt, quantum)

```

```

# Function to find turn around time

#for all processes

findTurnAroundTime(processes, n, bt, wt, tat)

# Display processes along with all details

print("Processes    Burst Time    Waiting", "Time    Turn-Around Time")

total_wt = 0

total_tat = 0

for i in range(n):

        total_wt = total_wt + wt[i]

        total_tat = total_tat + tat[i]

        print(" ", i + 1, "\t\t", bt[i],

                   "\t\t", wt[i], "\t\t", tat[i])

print("\nAverage waiting time = %.5f %(total_wt /n) )

print("Average turn around time = %.5f %(total_tat / n))

# Driver code

if __name__ == "__main__":

# Process id's

proc = [1, 2, 3]

n = 3

```

```

# Burst time of all processes
burst_time = [10, 5, 8]

# Time quantum
quantum = 2;
findavgTime(proc, n, burst_time, quantum)

```

OUTPUT (Round Robin CPU Scheduling): --

```

ani@ant:~/Desktop$ python3 rr.py
Processes      Burst Time      Waiting Time      Turn-Around Time
 1              10              13                23
 2              5               10                15
 3              8               13                21

Average waiting time = 12.00000
Average turn around time = 19.66667
ani@ant:~/Desktop$ S

```

RESULT: --

The priority based and round robin based types of CPU scheduling algorithms are implemented and tested.

Exp No. 10

Name: PONNURI ANIRUDDHA

Date: 12/10/2022

Reg No: RA2112704010015

FIFO Page Replacement Algorithm

Aim: --

To implement FIFO page replacement algorithm.

Procedure: --

Page Replacement algorithm:

In operating systems that use paging for memory management, page replacement algorithms are needed to decide which page needs to be replaced when a new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and the Operating System replaces one of the existing pages with a newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

CODE (FCFS): --

```
from queue import Queue

def pageFaults(pages, n, capacity):
    # To represent set of current pages.

    # We use an unordered_set so that we quickly check if a page is present in set
    # or not

    s = set()

    # To store the pages in FIFO manner

    indexes = Queue()

    # Start from initial page

    page_faults = 0

    for i in range(n):
        # Check if the set can hold more pages

        if (len(s) < capacity):

            # Insert it into set if not present already which represents page
            # fault

            if (pages[i] not in s):
```

```

    s.add(pages[i])
    # increment page fault
    page_faults += 1

    # Push the current page into the queue
    indexes.put(pages[i])

# If the set is full then need to perform FIFO
# i.e. remove the first page of the queue from
# set and queue both and insert the current page
else:

    # Check if current page is not
    # already present in the set
    if (pages[i] not in s):

        # Pop the first page from the queue
        val = indexes.queue[0]

        indexes.get()

# Remove the indexes page
s.remove(val)

# insert the current page
s.add(pages[i])

# push the current page into
# the queue

```

```

        indexes.put(pages[i])

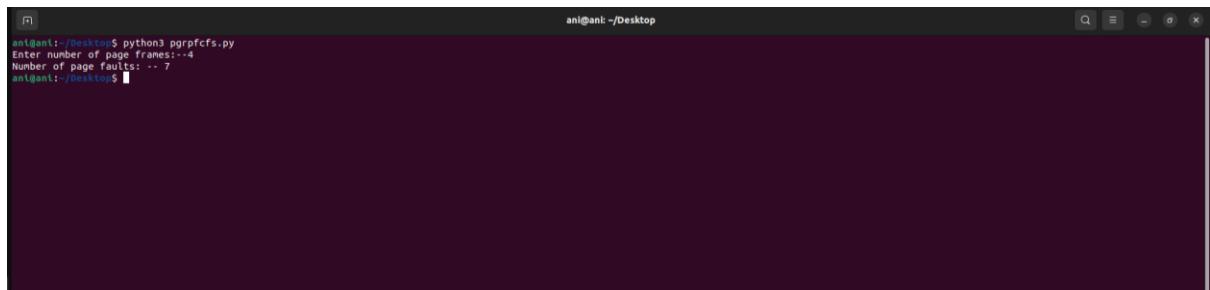
# Increment page faults
page_faults += 1

return page_faults

# Driver code
if __name__ == '__main__':
    pages = [7, 0, 1, 2, 0, 3, 0,
              4, 2, 3, 0, 3, 2]
    n = len(pages)
    capacity = int(input("Enter number of page frames:--"))
    print("Number of page faults: --",pageFaults(pages, n, capacity))

```

OUTPUT (FCFS): --



```

ani@ani:~/Desktop$ python3 pgpfefs.py
Enter number of page frames:--4
Number of page faults: -- 7
ani@ani:~/Desktop$ 

```

RESULT: --

The FIFO Page Replacement Algorithm is successfully implemented using python program.

Exp No. 11

Name: PONNURI ANIRUDDHA

Date: 18/10/2022

Reg No: RA2112704010015

LRU and LFU Page Replacement Algorithm

Aim: --

To implement LRU and LFU Page Replacement Algorithm

Procedure: --

LRU (Least Recently Used) Page Replacement:

The LRU stands for the Least Recently Used. It keeps track of page usage in the memory over a short period of time. It works on the concept that pages that have been highly used in the past are likely to be significantly used again in the future. It removes the page that has not been utilized in the memory for the longest time. LRU is the most widely used algorithm because it provides fewer page faults than the other methods.

LFU (Least Frequently Used) Page Replacement:

In this, it is using the concept of paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when the new page comes in. Whenever a new page is referred to and is not present in memory, the page fault occurs and the Operating System replaces one of the existing pages with a newly needed page. LFU is one such page replacement policy in which the least frequently used pages are replaced. If the frequency of pages is the same, then the page that has arrived first is replaced first.

Code (LRU): --

```
capacity = int(input('Enter number of page frames :-- '))

processList = [ 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2]

# List of current pages in Main Memory

s = []

pageFaults = 0

for i in processList:

    # If i is not present in currentPages list

    if i not in s:

        # Check if the list can hold equal pages

        if(len(s) == capacity):

            s.remove(s[0])

            s.append(i)

        else:

            s.append(i)

    # Increment Page faults

    pageFaults += 1

    # If page is already there in

    # currentPages i.e in Main

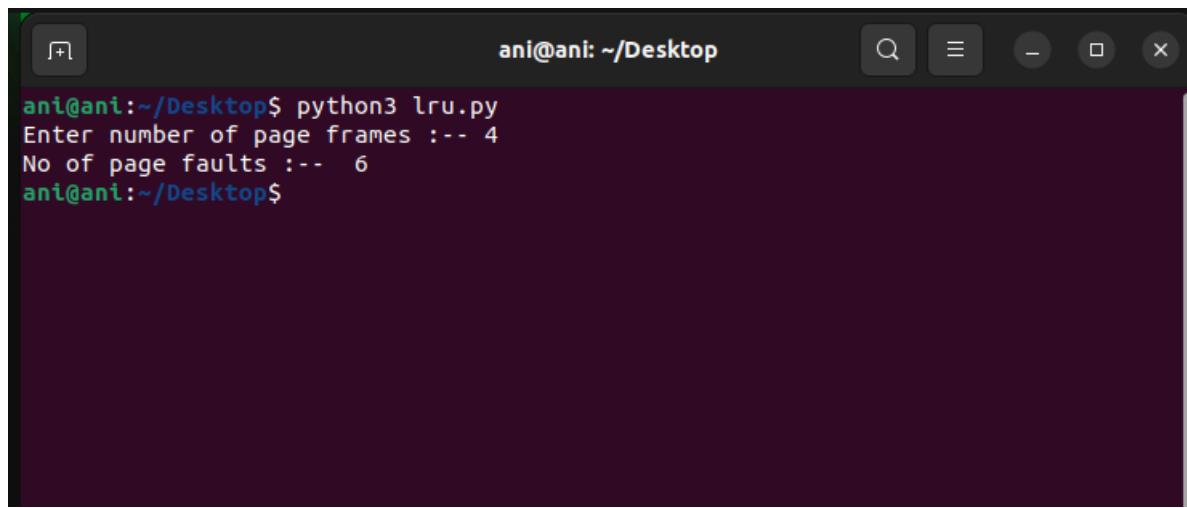
    else:

        # Remove previous index of current page

        s.remove(i)
```

```
# Now append it, at last index  
s.append(i)  
  
print("No of page faults :-- ",pageFaults)
```

OUTPUT (LRU): --



A screenshot of a terminal window titled "ani@ani: ~/Desktop". The window shows the command "python3 lru.py" being run, followed by user input for the number of page frames (4) and the resulting number of page faults (6). The terminal has a dark background with light-colored text and standard window controls.

```
ani@ani:~/Desktop$ python3 lru.py
Enter number of page frames :-- 4
No of page faults :-- 6
ani@ani:~/Desktop$
```

Code (LFU): --

```
#include <bits/stdc++.h>
using namespace std;
int pageFaults(int n, int c, int pages[])
{
    // Initialise count to 0
    int count = 0;
    vector<int> v;
    // To store frequency of pages
    unordered_map<int, int> mp;

    int i;
    for (i = 0; i <= n - 1; i++) {

        // Find if element is present in memory or not
        auto it = find(v.begin(), v.end(), pages[i]);

        // If element is not present
        if (it == v.end()) {

            // If memory is full
            if (v.size() == c) {

                // Decrease the frequency
                mp[v[0]]--;
                // Remove the first element as
                // It is least frequently used
                v.erase(v.begin());
            }
            v.push_back(pages[i]);
            count++;
        }
    }
    return count;
}
```

```

    }

    // Add the element at the end of memory
    v.push_back(pages[i]);
    // Increase its frequency
    mp[pages[i]]++;

    // Increment the count
    count++;
}

else {

    // If element is present
    // Remove the element
    // And add it at the end
    // Increase its frequency
    mp[pages[i]]++;
    v.erase(it);
    v.push_back(pages[i]);
}

// Compare frequency with other pages
// starting from the 2nd last page
int k = v.size() - 2;

// Sort the pages based on their frequency
// And time at which they arrive
// if frequency is same
// then, the page arriving first must be placed first

```

```

        while (mp[v[k]] > mp[v[k + 1]] && k > -1) {
            swap(v[k + 1], v[k]);
            k--;
        }
        return count;
    }
}

```

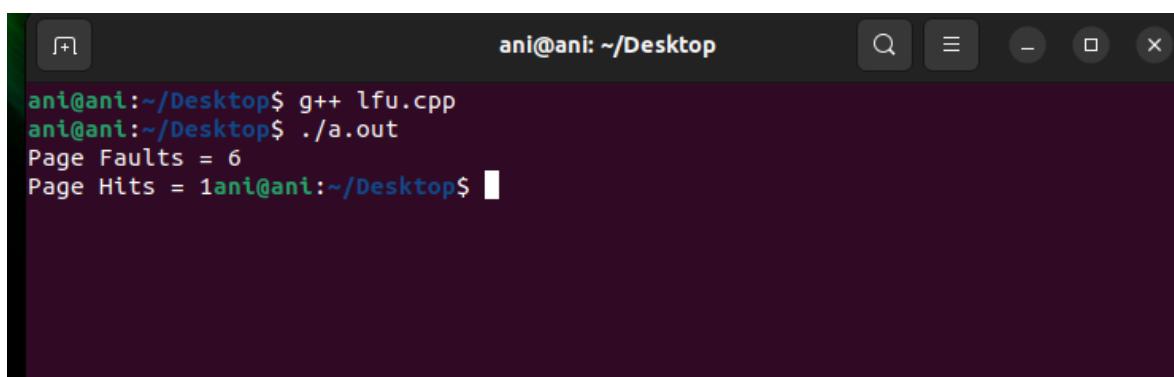
```

int main()
{
    int pages[] = { 1, 2, 3, 4, 2, 1, 5 };
    int n = 7, c = 3;

    cout << "Page Faults = " << pageFaults(n, c, pages) << endl;
    cout << "Page Hits = " << n - pageFaults(n, c, pages);
    return 0;
}

```

OUTPUT (LFU): --



```

ani@ani:~/Desktop$ g++ lfu.cpp
ani@ani:~/Desktop$ ./a.out
Page Faults = 6
Page Hits = 1

```

RESULT: --

Successfully implemented LRU and LFU page replacement algorithm in python

Exp No. 12

Name: PONNURI ANIRUDDHA

Date: 18/10/2022

Reg No: RA2112704010015

Best fit and Worst fit memory management policies

Aim: --

To implement Best fit and Worst fit memory management policies

Procedure: --

BEST FIT MEMORY MANAGEMENT: --

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

Advantages of Best-Fit Allocation:

Memory Efficient. The operating system allocates the job minimum possible space in the memory, making memory management very efficient. To save memory from getting wasted, it is the best method.

Best-Fit Allocation Benefits:

It is a Slow Process. Checking the whole memory for each job makes the working of the operating system very slow. It takes a lot of time to complete the work.

WORST FIT MEMORY MANAGEMENT: --

In this allocation method, the process scans the entire memory, looking for the biggest hole or partition, and then it is allocated to that hole or partition. In order to find the biggest hole, the method must go over the entire memory, which takes time.

Worst-Fit Allocation Benefits

There will be significant internal fragmentation because this process selects the greatest hole or fragment. This internal fragmentation will now be sufficiently large to allow for the placement of additional smaller processes in the unused partition.

Worst-Fit Allocation Drawbacks:

It is a slow procedure since it first navigates across every memory partition before choosing the largest one, which takes a long time.

CODE (BEST FIT): --

```
def bestFit(blockSize, m, processSize, n):  
    # Stores block id of the block allocated to a process  
    allocation = [-1] * n  
  
    for i in range(n):  
  
        # Find the best fit block for current process  
        bestIdx = -1  
  
        for j in range(m):  
            if blockSize[j] >= processSize[i]:  
                if bestIdx == -1:  
                    bestIdx = j  
  
                elif blockSize[bestIdx] > blockSize[j]:  
                    bestIdx = j  
  
        # If we could find a block for  
        # current process  
        if bestIdx != -1:  
  
            # allocate block j to p[i] process  
            allocation[i] = bestIdx  
  
            # Reduce available memory in this block.  
            blockSize[bestIdx] -= processSize[i]  
  
    print("Process No. Process Size    Block no.")  
    for i in range(n):  
        print(i + 1, "           ", processSize[i], end = "          ")  
        if allocation[i] != -1:
```

```

        print(allocation[i] + 1)

else:
    print("Not Allocated")

# Driver code

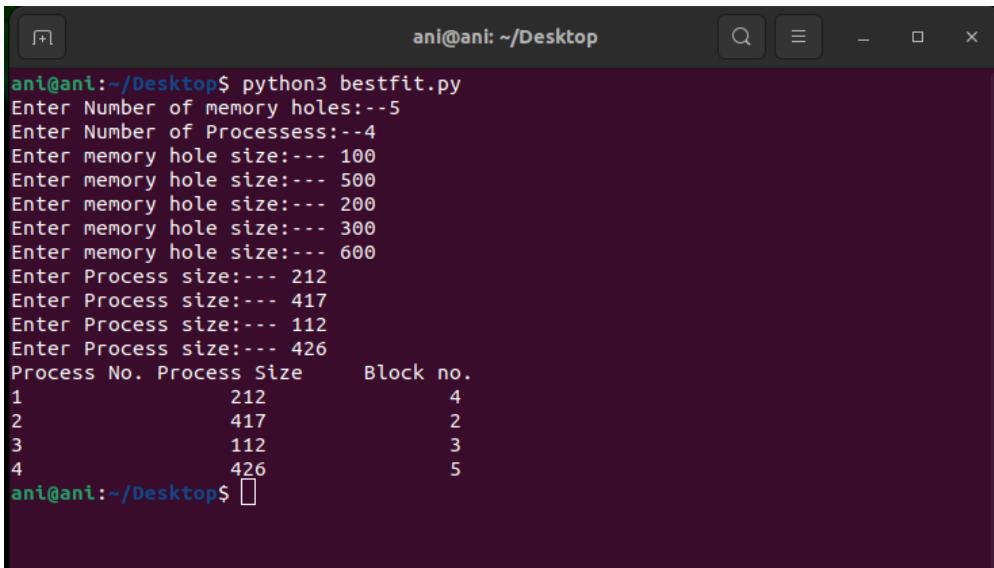
if __name__ == '__main__':
    blockSize = []
    processSize = []
    m = int(input("Enter Number of memory holes:--"))
    n = int(input("Enter Number of Processes:--"))
    for i in range(m):
        x=int(input("Enter memory hole size:--- "))
        blockSize.append(x)

for i in range(n):
    x=int(input("Enter Process size:--- "))
    processSize.append(x)

bestFit(blockSize, m, processSize, n)

```

OUTPUT (BEST FIT): --



```

ani@ani:~/Desktop$ python3 bestfit.py
Enter Number of memory holes:--5
Enter Number of Processes:--4
Enter memory hole size:--- 100
Enter memory hole size:--- 500
Enter memory hole size:--- 200
Enter memory hole size:--- 300
Enter memory hole size:--- 600
Enter Process size:--- 212
Enter Process size:--- 417
Enter Process size:--- 112
Enter Process size:--- 426
Process No. Process Size      Block no.
1              212             4
2              417             2
3              112             3
4              426             5
ani@ani:~/Desktop$ 

```

CODE (WORST FIT): --

```
def worstFit(blockSize, m, processSize, n):  
    # Stores block id of the block allocated to a process  
    # Initially no block is assigned  
    # to any process  
    allocation = [-1] * n  
  
    # pick each process and find suitable blocks  
    # according to its size ad assign to it  
    for i in range(n):  
  
        # Find the best fit block for  
        # current process  
        wstIdx = -1  
        for j in range(m):  
            if blockSize[j] >= processSize[i]:  
                if wstIdx == -1:  
                    wstIdx = j  
                elif blockSize[wstIdx] < blockSize[j]:  
                    wstIdx = j  
  
        # If we could find a block for  
        # current process  
        if wstIdx != -1:  
  
            # allocate block j to p[i] process  
            allocation[i] = wstIdx  
  
            # Reduce available memory in this block.
```

```

blockSize[wstIdx] -= processSize[i]

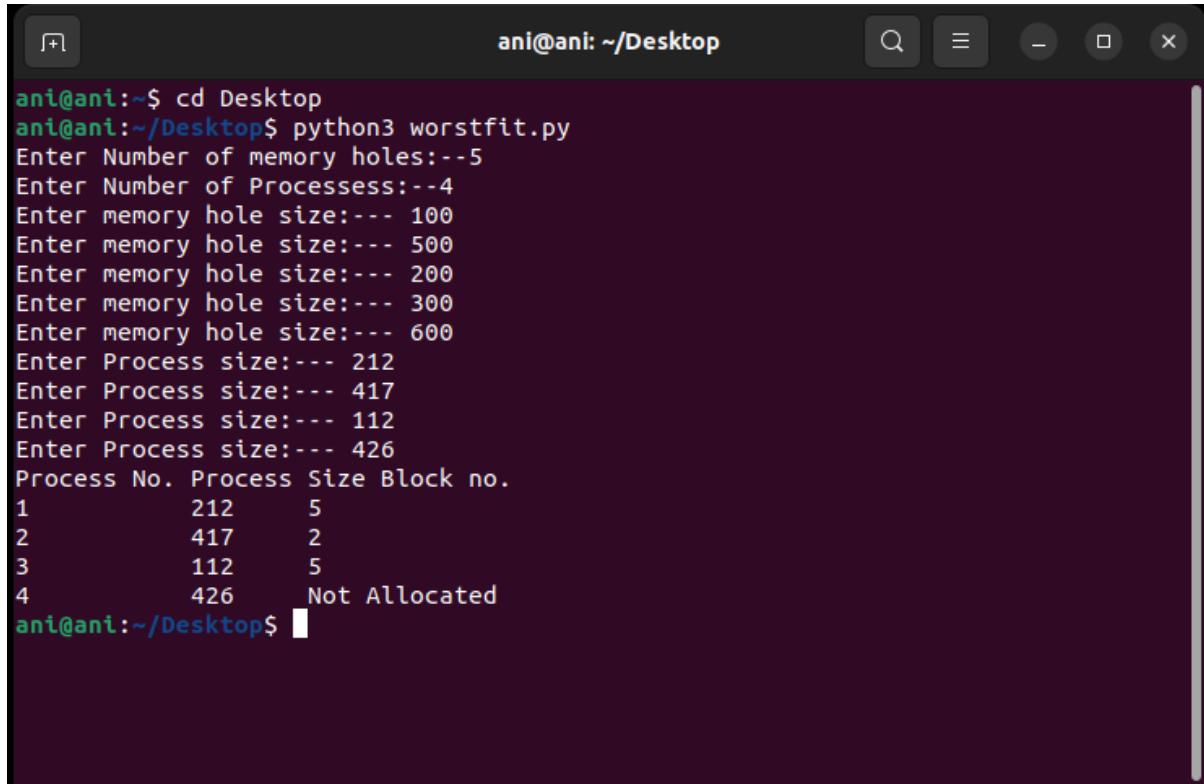
print("Process No. Process Size Block no.")
for i in range(n):
    print(i + 1, "      ",
          processSize[i], end="      ")
    if allocation[i] != -1:
        print(allocation[i] + 1)
    else:
        print("Not Allocated")

# Driver code
if __name__ == '__main__':
    blockSize = []
    processSize = []
    m = int(input("Enter Number of memory holes:--"))
    n = int(input("Enter Number of Processes:--"))
    for i in range(m):
        x = int(input("Enter memory hole size:--- "))
        blockSize.append(x)
    for i in range(n):
        x = int(input("Enter Process size:--- "))
        processSize.append(x)

worstFit(blockSize, m, processSize, n)

```

OUTPUT (WORST FIT): --



The screenshot shows a terminal window titled "ani@ani: ~/Desktop". The user has run a Python script named "worstfit.py". The script prompts for the number of memory holes and processes, and then lists the memory hole sizes and process sizes. It then displays the allocation results:

Process No.	Process Size	Block no.
1	212	5
2	417	2
3	112	5
4	426	Not Allocated

RESULT: --

The best fit and worst fit memory management policies are implemented using python programming.

Exp No. 13

Date: 26/10/2022

Name: PONNURI ANIRUDDHA

Reg No: RA2112704010015

Disk Scheduling algorithm

Aim: --

To implement Disk Scheduling algorithm

Procedure: --

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

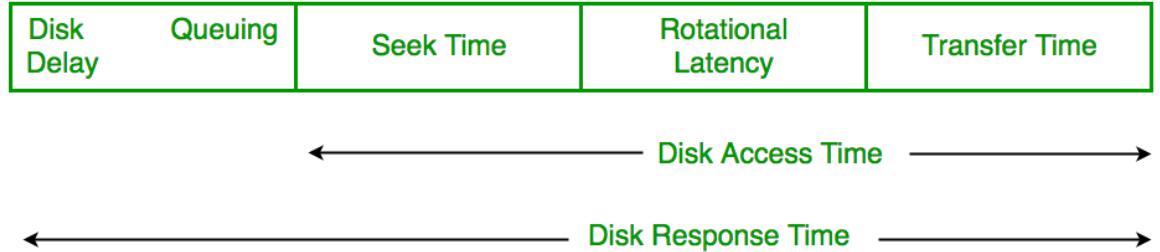
Disk scheduling is important because:

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

There are many Disk Scheduling Algorithms but before discussing them let's have a quick look at some of the important terms:

- Seek Time: Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.
- Rotational Latency: Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.
- Transfer Time: Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.
- Disk Access Time: Disk Access Time is:

$$\text{Disk Access Time} = \text{Seek Time} + \text{Rotational Latency} + \text{Transfer Time}$$



- Disk Response Time: Response Time is the average of time spent by a request waiting to perform its I/O operation. *Average Response time* is the response time of all requests. *Variance Response Time* is a measure of how individual requests are serviced with respect to average response time. So the disk scheduling algorithm that gives minimum variance response time is better.

Disk Scheduling Algorithms

FCFS:

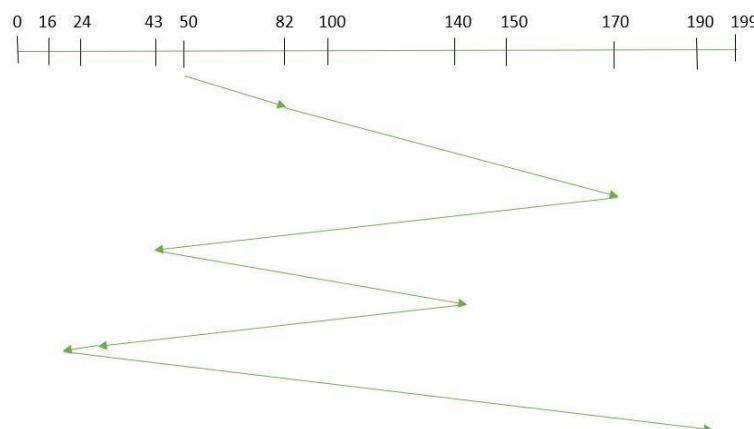
FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue. Let us understand this with the help of an example.

Algorithm:

- Let Request array represents an array storing indexes of tracks that have been requested in ascending order of their time of arrival. ‘head’ is the position of disk head.
- Let us one by one take the tracks in default order and calculate the absolute distance of the track from the head.
- Increment the total seek count with this distance.
- Currently serviced track position now becomes the new head position.
- Go to step 2 until all tracks in request array have not been serviced.

Example:

Suppose the order of request is (82, 170, 43, 140, 24, 16, 190)
And current position of Read/Write head is: 50



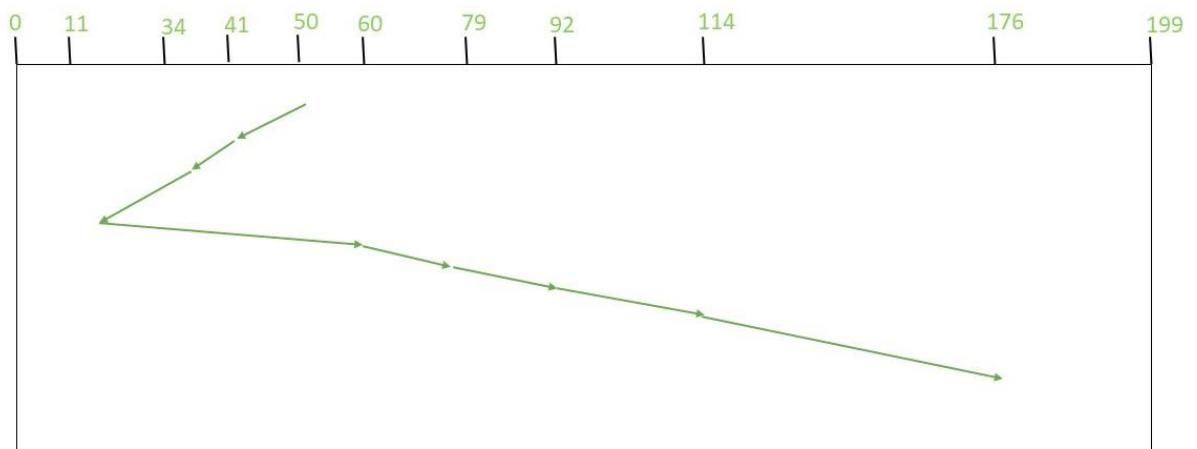
Shortest Seek Time First:

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

Example:

Suppose the order of request is- (176, 79, 34, 60, 92, 11, 41, 114)

And current position of Read/Write head is : 50



So, total seek time:

$$\begin{aligned} &= (50-41)+(41-34)+(34-11)+(60-11)+(79-60)+(92-79)+(114-92) \\ &\quad +(176-114) \\ &= 204 \end{aligned}$$

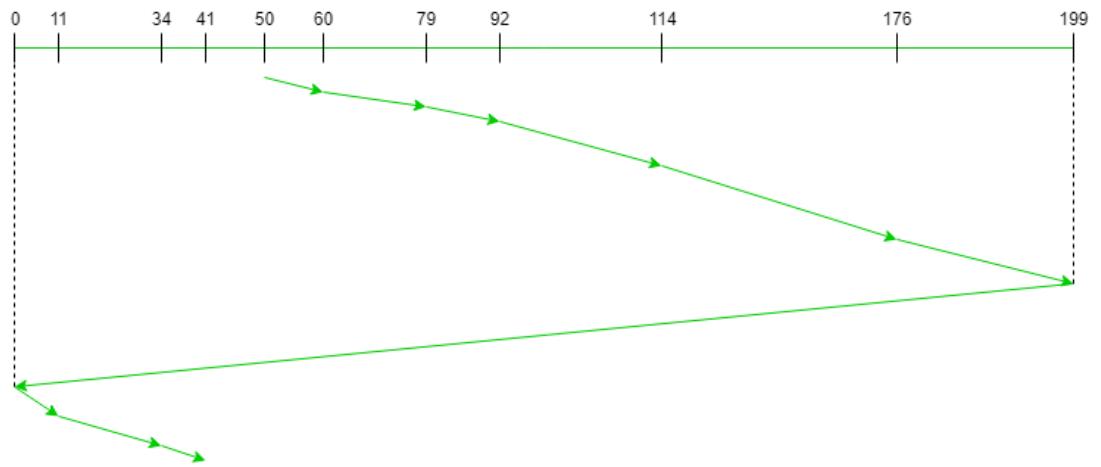
CSCAN:

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

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

Example:

Suppose the requests to be addressed are- 176, 79, 34, 60, 92, 11, 41, 114. And the Read/Write arm is at 50, and it is also given that the disk arm should move “towards the larger value”.



Therefore, the total seek count is calculated as:

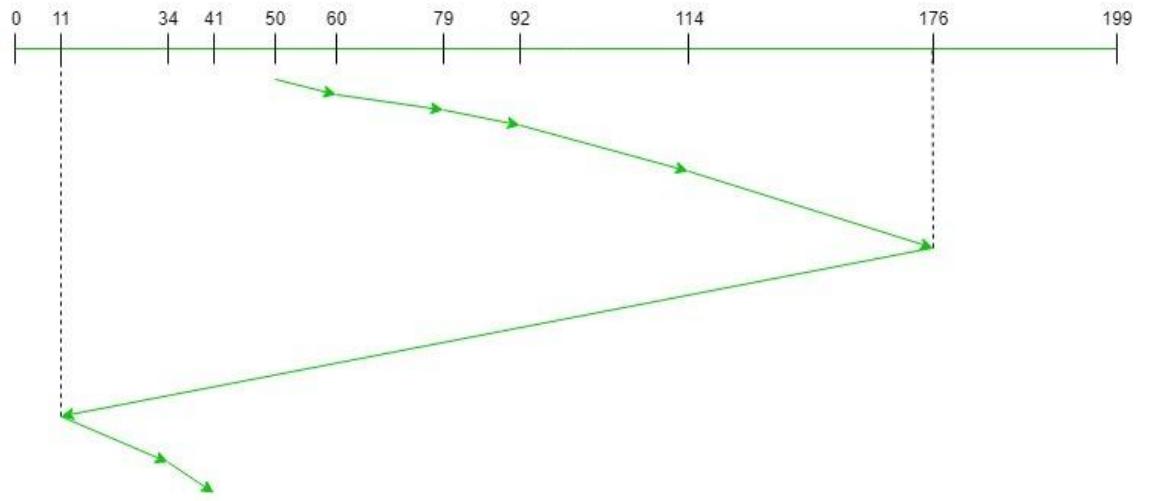
$$\begin{aligned} &= (60-50)+(79-60)+(92-79) + (114-92)+(176-114)+(199-176)+(199-0) \\ &\quad +(11-0)+(34-11)+(41-34) \\ &= 389 \end{aligned}$$

CLOOK

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

Example:

Suppose the requests to be addressed are- 176, 79, 34, 60, 92, 11, 41, 114. And the Read/Write arm is at 50 and it is also given that the disk arm should move “towards the larger value”



Therefore, the total seek count

$$\begin{aligned} &= (60 - 50) + (79 - 60) + (92 - 79) + (114 - 92) + (176 - 114) \\ &\quad + (176 - 11) + (34 - 11) + (41 - 34) \\ &= 321 \end{aligned}$$

CODE (FCFS): --

```
def FCFS(arr, head):  
  
    seek_count = 0;  
    distance, cur_track = 0, 0;  
  
    for i in range(size):  
        cur_track = arr[i]  
  
        # calculate absolute distance  
        distance = abs(cur_track - head)  
  
        # increase the total count  
        seek_count += distance  
  
        # accessed track is now new head  
        head = cur_track  
  
    print("Total number of seek operations = ",seek_count)  
  
    # Seek sequence would be the same  
    # as request array sequence  
    print("Seek Sequence is");  
  
    for i in range(size):  
        print(arr[i]);  
  
    # Driver code  
if __name__ == '__main__':
```

```

# request array

arr = []

size = int(input("Enter size of request array:-- "))

for i in range(size):

    x = int(input('Enter positon {i+1}:--'))

    arr.append(x)

head = int(input("Enter Current position of head :-- "))

```

FCFS(arr, head);

OUTPUT (FCFS): --

```

ani@ani:~/Desktop$ python3 dfcfs.py
Enter size of request array:-- 7
Enter position 1:-- 82
Enter position 2:-- 170
Enter position 3:-- 43
Enter position 4:-- 140
Enter position 5:-- 24
Enter position 6:-- 16
Enter position 7:-- 198
Enter current position of head :-- 50
Total number of seek operations = 642
Seek Sequence is
82
170
43
140
24
16
198
ani@ani:~/Desktop$ 

```

CODE (SSTF): --

```
def calculateDifference(queue, head, diff):  
    for i in range(len(diff)):  
        diff[i][0] = abs(queue[i] - head)
```

```
# find unaccessed track which is  
# at minimum distance from head
```

```
def findMin(diff):
```

```
    index = -1
```

```
    minimum = 999999999
```

```
    for i in range(len(diff)):
```

```
        if (not diff[i][1] and
```

```
            minimum > diff[i][0]):
```

```
                minimum = diff[i][0]
```

```
                index = i
```

```
    return index
```

```
def shortestSeekTimeFirst(request, head):
```

```
    if (len(request) == 0):
```

```
        return
```

```
    l = len(request)
```

```
    diff = [0] * l
```

```
    # initialize array
```

```
    for i in range(l):
```

```
        diff[i] = [0, 0]
```

```

# count total number of seek operation
seek_count = 0

# stores sequence in which disk
# access is done
seek_sequence = [0] * (l + 1)

for i in range(l):
    seek_sequence[i] = head
    calculateDifference(request, head, diff)
    index = findMin(diff)

    diff[index][1] = True

# increase the total count
seek_count += diff[index][0]

# accessed track is now new head
head = request[index]

# for last accessed track
seek_sequence[len(seek_sequence) - 1] = head

print("Total number of seek operations =", seek_count)

print("Seek Sequence is")

```

```

# print the sequence
for i in range(l + 1):
    print(seek_sequence[i])

# Driver code

if __name__ == "__main__":
    # request array
    arr = []
    size = int(input("Enter size of request array:-- "))
    for i in range(size):
        x = int(input("Enter positon {i+1}:--'"))
        arr.append(x)
    head = int(input("Enter Current position of head :-- "))
    shortestSeekTimeFirst(arr,head)

```

OUTPUT (SSTF): --

```

root@root:~/Desktop$ python3 sstf.py
Enter size of request array:-- 8
Enter positon 1:-176
Enter positon 2:-79
Enter positon 3:-34
Enter positon 4:-60
Enter positon 5:-92
Enter positon 6:-11
Enter positon 7:-41
Enter positon 8:-114
Enter Current positon of head :-- 50
Total number of seek operations = 204
Seek Sequence is
50
41
34
11
60
79
92
114
176
root@root:~/Desktop$

```

CODE(CSCAN): --

```
disk_size = 200

def CSCAN(arr, head):

    seek_count = 0
    distance = 0
    cur_track = 0
    left = []
    right = []
    seek_sequence = []

    left.append(0)
    right.append(disk_size - 1)

    for i in range(size):
        if (arr[i] < head):
            left.append(arr[i])
        if (arr[i] > head):
            right.append(arr[i])

    # Sorting left and right vectors
    left.sort()
    right.sort()

    # First service the requests
    # on the right side of the
    # head.
    for i in range(len(right)):
        cur_track = right[i]
```

```

# Appending current track
# to seek sequence
seek_sequence.append(cur_track)

# Calculate absolute distance
distance = abs(cur_track - head)

# Increase the total count
seek_count += distance

# Accessed track is now new head
head = cur_track

# Once reached the right end
# jump to the beginning.
head = 0

# adding seek count for head returning from 199 to 0
seek_count += (disk_size - 1)

# Now service the requests again
# which are left.
for i in range(len(left)):
    cur_track = left[i]

# Appending current track
# to seek sequence
seek_sequence.append(cur_track)

```

```

# Calculate absolute distance
distance = abs(cur_track - head)

# Increase the total count
seek_count += distance

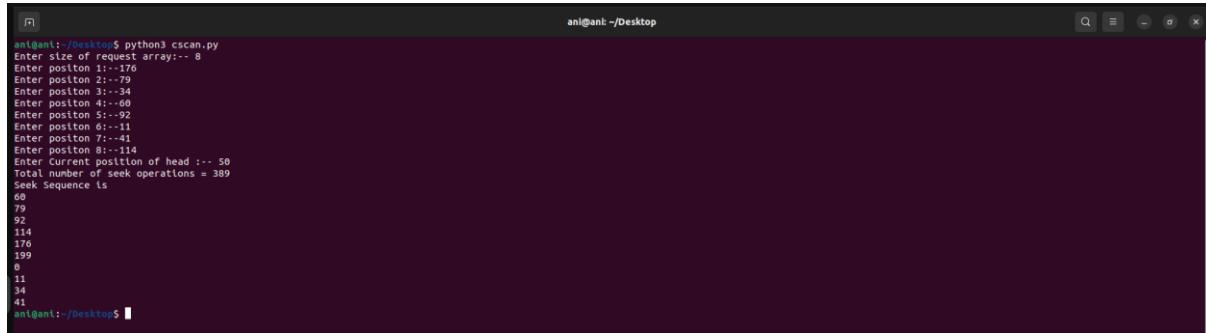
# Accessed track is now the new head
head = cur_track

print("Total number of seek operations =", seek_count)
print("Seek Sequence is")
print(*seek_sequence, sep="\n")

# Driver code
if __name__ == "__main__":
    # request array
    arr = []
    size = int(input("Enter size of request array:-- "))
    for i in range(size):
        x = int(input(f'Enter positon {i+1}:-- '))
        arr.append(x)
    head = int(input("Enter Current position of head :-- "))
    CSCAN(arr, head)

```

OUTPUT (CSCAN): --



A terminal window titled "ani@ani: ~/Desktop" showing the execution of a Python script named cscan.py. The script prompts for 8 positions and a head position, then calculates seek operations. The output shows the seek sequence.

```
ani@ani:~/Desktop$ python3 cscan.py
Enter size of request array:-- 8
Enter positon 1:--176
Enter positon 2:--114
Enter positon 3:--34
Enter positon 4:--60
Enter positon 5:--92
Enter positon 6:--11
Enter positon 7:--41
Enter positon 8:--114
Enter current position of head :-- 50
Total number of seek operations = 389
Seek Sequence ls
66
79
92
114
176
199
0
11
34
41
ani@ani:~/Desktop$
```

CODE (CLOOK): --

disk_size = 200

def CLOOK(arr, head):

seek_count = 0

distance = 0

cur_track = 0

left = []

right = []

seek_sequence = []

for i in range(size):

if (arr[i] < head):

left.append(arr[i])

if (arr[i] > head):

right.append(arr[i])

Sorting left and right vectors

left.sort()

right.sort()

First service the requests

on the right side of the

head

for i in range(len(right)):

cur_track = right[i]

```
# Appending current track  
# seek sequence  
seek_sequence.append(cur_track)
```

```
# Calculate absolute distance  
distance = abs(cur_track - head)
```

```
# Increase the total count  
seek_count += distance
```

```
# Accessed track is now new head  
head = cur_track
```

```
# Once reached the right end  
# jump to the last track that  
# is needed to be serviced in  
# left direction  
seek_count += abs(head - left[0])  
head = left[0]
```

```
# Now service the requests again  
# which are left  
for i in range(len(left)):  
    cur_track = left[i]
```

```
# Appending current track to  
# seek sequence  
seek_sequence.append(cur_track)
```

```

# Calculate absolute distance
distance = abs(cur_track - head)

# Increase the total count
seek_count += distance

# Accessed track is now the new head
head = cur_track

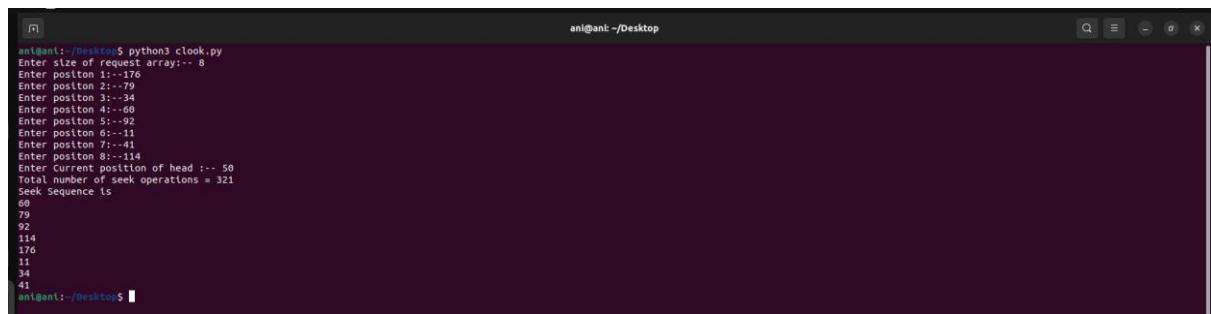
print("Total number of seek operations =", seek_count)
print("Seek Sequence is")

for i in range(len(seek_sequence)):
    print(seek_sequence[i])

# Driver code
if __name__ == "__main__":
    # request array
    arr = []
    size = int(input("Enter size of request array:-- "))
    for i in range(size):
        x = int(input(f'Enter positon {i+1}:-- '))
        arr.append(x)
    head = int(input("Enter Current position of head :-- "))
    CLOOK(arr,head)

```

OUTPUT (CLOOK): --



```
anil@ankit:~/Desktop$ python3 clook.py
Enter size of request array:-- 8
Enter position 1:--176
Enter position 2:--79
Enter position 3:--34
Enter position 4:--60
Enter position 5:--92
Enter position 6:--11
Enter position 7:--41
Enter position 8:--114
Enter Current position of head :-- 50
Total number of seek operations = 321
Seek Sequence is
60
79
92
114
116
11
11
34
41
anil@ankit:~/Desktop$
```

RESULT: --

Successfully implemented disk scheduling algorithms in python.

Exp No. 14

Name: PONNURI ANIRUDDHA

Date: 08/11/2022

Reg No: RA2112704010015

Sequential and Indexed file Allocation

Aim: --

To implement Sequential and Indexed file Allocation

Procedure: --

Sequential file Allocation:

Files are normally stored on the disks. So the main problem is how to allocate space to those files. So that disk space is utilized effectively and files can be accessed quickly. Three major strategies of allocating disc space are in wide use. Sequential, indexed and linked.

In this allocation strategy, each file occupies a set of contiguously blocks on the disk. This strategy is best suited. For sequential files, the file allocation table consists of a single entry for each file. It shows the filenames, starting block of the file and size of the file. The main problem with this strategy is, it is difficult to find the contiguous free blocks in the disk and some free blocks could happen between two files

Indexed Allocation

In this scheme, a special block known as the Index block contains the pointers to all the blocks occupied by a file. Each file has its own index block. The i th entry in the index block contains the disk address of the i th file block. The directory entry contains the address of the index block.

Advantages:

- This supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks.
- It overcomes the problem of external fragmentation.

Disadvantages:

- The pointer overhead for indexed allocation is greater than linked allocation.
- For very small files, say files that expand only 2-3 blocks, the indexed allocation would keep one entire block (index block) for the pointers which is inefficient in terms of memory utilization.

Algorithm (Sequential File Allocation):

- 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.

CODE (Sequential file Allocation): --

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

#define TOTAL_DISK_BLOCKS 32
#define TOTAL_DISK_INODES 8

#ifndef MAX
#define MAX 15
#endif

int blockStatus[TOTAL_DISK_BLOCKS]; //free = 0
int blockStart;
```

```

struct file_table {
    char fileName[20];
    int startBlock;
    int fileSize;
    int allotStatus;
};

struct file_table fileTable[TOTAL_DISK_BLOCKS - TOTAL_DISK_INODES];

int AllocateBlocks(int Size) {
    int i = 0, count = 0, inList = 0, nextBlock = 0;
    int allocStartBlock = TOTAL_DISK_INODES;
    int allocEndBlock = TOTAL_DISK_BLOCKS - 1;

    // check whether sufficient free blocks are available
    for (i = 0; i < (TOTAL_DISK_BLOCKS - TOTAL_DISK_INODES); i++)
        if (blockStatus[i] == 0)
            count++;
        if (count < Size)
            return 1; // not enough free blocks

    count = 0;
    while (count < Size) {
        nextBlock = (rand() % (allocEndBlock - allocStartBlock + 1)) + allocStartBlock;

        for (i = nextBlock; i < (nextBlock + Size); i++) {
            if (blockStatus[i] == 0)
                count = count + 1;
    }
}

```

```

        else {
            count = 0;
            break;
        }
    }

    blockStart = nextBlock;
    for (int i = 0; i < Size; i++) {
        blockStatus[blockStart + i] = 1;
    }
    if (count == Size)
        return nextBlock; // success
    else
        return 1; // not successful
}

void main() {
    int i = 0, j = 0, numFiles = 0, nextBlock = 0, ret = 1, totalFileSize = 0;
    char s[20];
    //-- -
    char *header[] = {"FILE_fileName", "FILE_SIZE", "BLOCKS_OCCUPIED"};
    printf("File allocation method: SEQUENTIAL\n");
    printf("Total blocks: %d\n", TOTAL_DISK_BLOCKS);
    printf("File allocation start at block: %d\n", TOTAL_DISK_INODES);
    printf("File allocation end at block: %d\n", TOTAL_DISK_BLOCKS - 1);
    printf("Size (kB) of each block: 1\n\n");
    printf("Enter no of files: ");
    scanf("%d", &numFiles);
}

```

```

//numFiles = 3;

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

//-- -
printf("\nEnter the name of file # %d: ", i+1);
scanf("%s", fileTable[i].fileName);

printf("Enter the size (kB) of file # %d: ", i+1);
scanf("%d", &fileTable[i].fileSize);

//strcpy(fileTable[i].fileName, "testfile");

srand(1234);

ret = AllocateBlocks(fileTable[i].fileSize);

//-- -
if (ret == 1) {
    exit(0);
} else {
    fileTable[i].startBlock = ret;
}
}

printf("\n%*s %*s %*s\n", -MAX, header[0], -MAX, header[1], MAX, header[2]);
//Seed the pseudo-random number generator used by rand() with the value seed
srand(1234);

//-- -
for (j = 0; j < numFiles; j++) {

printf("\n%*s %*d ", -MAX, fileTable[j].fileName, -MAX, fileTable[j].fileSize);

for(int k=0;k<fileTable[j].fileSize;k++) {

printf("%d%os", fileTable[j].startBlock+k, (k == fileTable[j].fileSize-1) ? "" : "-");
}
}
}

```

```

    }
}

printf("\nFile allocation completed. Exiting.\n");
}

```

OUTPUT (Sequential file Allocation): --

```

ani@ani: $ cd Desktop
ani@ani:~/Desktop$ gcc seq.c
ani@ani:~/Desktop$ ./a.out
File allocation method: SEQUENTIAL
Total blocks: 32
File allocation start at block: 8
File allocation end at block: 31
Size (kB) of each block: 1

Enter no of files: 4

Enter the name of file #1: test1
Enter the size (kB) of file #1: 4

Enter the name of file #2: test2
Enter the size (kB) of file #2: 5

Enter the name of file #3: test3
Enter the size (kB) of file #3: 4

Enter the name of file #4: test4
Enter the size (kB) of file #4: 6

FILE_fileName      FILE_SIZE        BLOCKS_OCCUPIED
test1              4                14-15-16-17
test2              5                19-20-21-22-23
test3              4                27-28-29-30
test4              6                8-9-10-11-12-13
File allocation completed. Exiting.
ani@ani:~/Desktop$ 

```

CODE (**Indexed Allocation**): --

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
int f[50], index[50], i, n, st, len, j, c, k, ind, count=0;
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);
}

```

OUTPUT (Indexed Allocation): --

```

ani@ani:~/Desktop$ gcc seq.c
ani@ani:~/Desktop$ ./a.out
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
ani@ani:~/Desktop$ 

```

RESULT: --

Successfully implemented Sequential and Indexed file Allocation in c program

Exp No. 15

Name: PONNURI ANIRUDDHA

Date: 14/11/2022

Reg No: RA2112704010015

File organization schemes for single level and two-level directory

Aim: --

To implement File organization schemes for single level and two-level directory

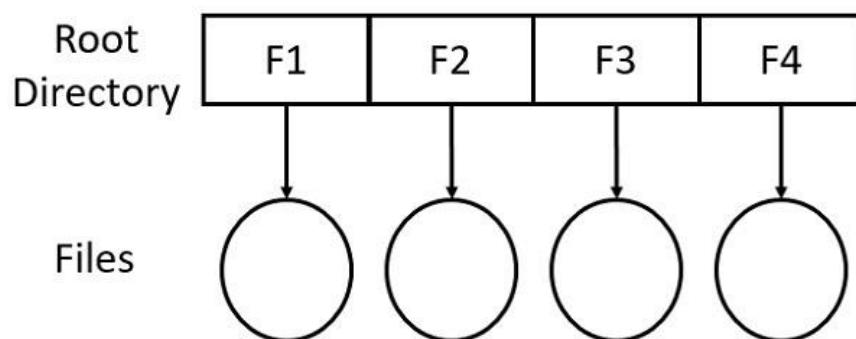
Procedure: --

A File system contains thousands and millions of files, owned by several users. The directory structure organizes these files by keeping entries of all the related files. The file entries have information like file name, type, location, the mode in which the file can be accessed by other users in the system. Directory structure provides both the above-discussed features. A directory always has information about the group of related files. Whenever a user or a process request for a file, the file system search for the file's entry in the directory and when the match is found, it obtains the file's location from there.

Types of Directory Structures

1. Single-level directory structure

Single level directory structure has only one directory which is called the root directory. The users are not allowed to create subdirectories under the root directory. All the files created by the several users are present in the root directory only.

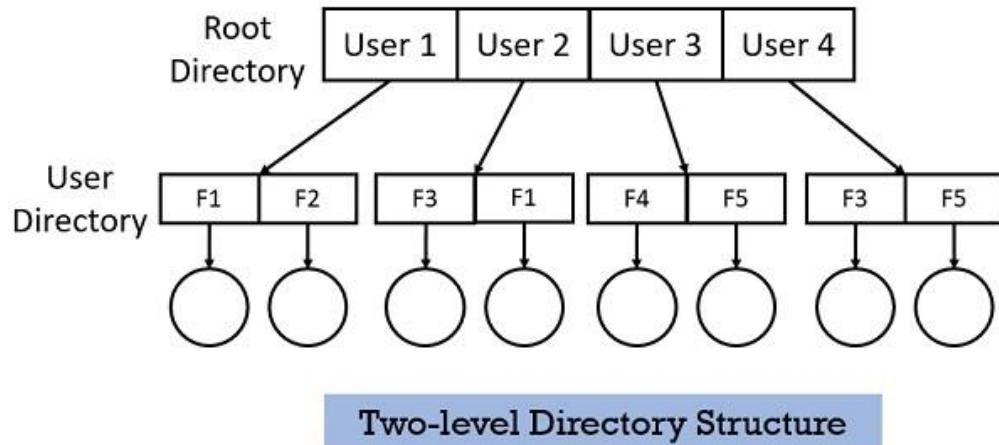


Single-level Directory Structure

There is one drawback of Single-level directory structure, a user cannot use the same file name used by another user in the system. Even if the file with the same name is created the old file will get destroyed first and replaced by the new file having the same name.

2. Two-level directory structure

In Two-level directory structure, the users create directory directly inside the root directory. But once a user creates such directory, further he cannot create any subdirectory inside that directory. Observe the figure below, 4 users have created their separate directory inside the root directory. But further, no subdirectory is created by the users.



This two-level structure allows each user to keep their files separately inside their own directory. This structure allows to use the same name for the files but under different user directories.

CODE (Single-level directory structure): --

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
void main()
{
    int nf=0,i=0,j=0,ch;
    char mdname[10],fname[10][10],name[10];
    printf("Enter the directory name:");
    scanf("%s",mdname);
    printf("Enter the number of files:");
    scanf("%d",&nf);
    do
```

```

{
printf("Enter file name to be created:");
scanf("%s",name);
for(i=0;i<nf;i++)
{
if(!strcmp(name,fname[i]))
break;
}
if(i==nf)
{
strcpy(fname[j++],name);
nf++;
}
else
printf("There is already %s\n",name);
printf("Do you want to enter another file(yes - 1 or no - 0):");
scanf("%d",&ch);
}
while(ch==1);

printf("Directory name is:%s\n",mdname);
printf("Files names are:");
for(i=0;i<j;i++)
printf("\n%s",fname[i]);
}

```

OUTPUT (Single-level directory structure): --



The screenshot shows a terminal window with the following session:

```

ani@ani:~/Desktop$ gcc adir.c
ani@ani:~/Desktop$ ./a.out
Enter the directory name:Temp
Enter the number of files:3
Enter file name to be created:aaa
Do you want to enter another file(yes - 1 or no - 0):1
Enter file name to be created:bbb
Do you want to enter another file(yes - 1 or no - 0):1
Enter file name to be created:ccc
Do you want to enter another file(yes - 1 or no - 0):0
Directory name is:Temp
Files names are:
aaa
bbb
ccc
ani@ani:~/Desktop$ 

```

CODE: --

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

struct st
{
    char dname[10];
    char sname[10][10];
    char fname[10][10][10];
    int ds,sds[10];
}dir[10];

void main()
{
    int i,j,k,n;
    printf("enter number of directories:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("enter directory %d names:",i+1);
        scanf("%s",dir[i].dname);
        printf("enter size of directories:");
        scanf("%d",&dir[i].ds);
        for(j=0;j<dir[i].ds;j++)
        {
            printf("enter subdirectory name and size:");
            scanf("%s",dir[i].sname[j]);
            scanf("%d",&dir[i].sds[j]);
            for(k=0;k<dir[i].sds[j];k++)
            {

```

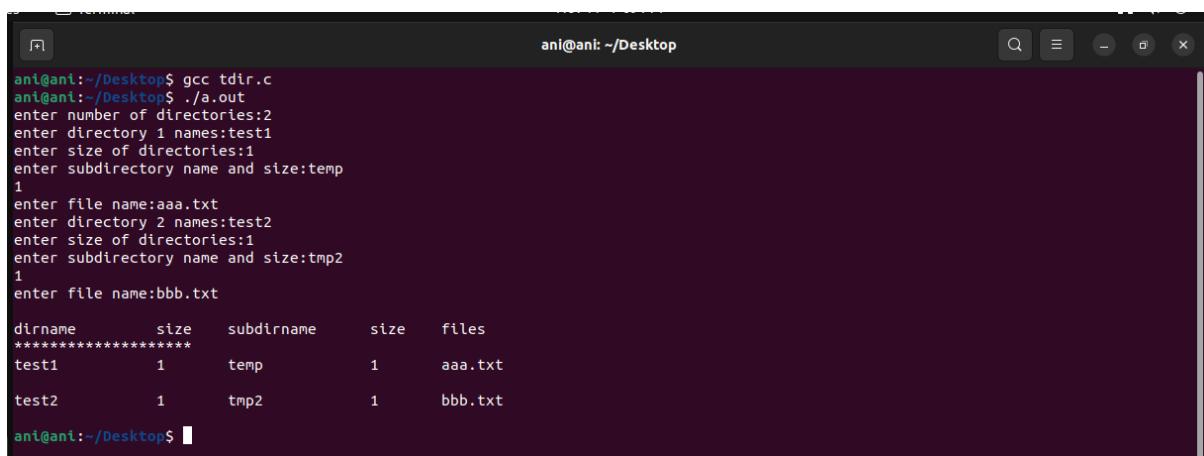
```

printf("enter file name:");
scanf("%s",dir[i].fname[j][k]);
}
}
}

printf("\ndirname\t\tsize\tsubdirname\tsize\tfiles");
printf("\n*****\n");
for(i=0;i<n;i++)
{
printf("%s\t%d",dir[i].dname,dir[i].ds);
for(j=0;j<dir[i].ds;j++)
{
printf("\t%s\t%d\t",dir[i].sdname[j],dir[i].sds[j]);
for(k=0;k<dir[i].sds[j];k++)
printf("%s\t",dir[i].fname[j][k]);
printf("\n\t");
}
printf("\n");
}
}

```

OUTPUT (Two-level directory structure): --



The screenshot shows a terminal window titled "ani@ani: ~/Desktop". The user has run the command "gcc tdir.c" followed by "./a.out". They are prompted to enter the number of directories, which they type "2". Then, they enter "test1" as the name of the first directory and "temp" as its size. For the second directory, "test2", they enter "tmp2" as its size. Finally, they add a file named "aaa.txt" to "test1" and "bbb.txt" to "test2". The output displays the directory structure with columns for dirname, size, subdirname, size, and files. The final output is:

```

ani@ani:~/Desktop$ gcc tdir.c
ani@ani:~/Desktop$ ./a.out
enter number of directories:2
enter directory 1 names:test1
enter size of directories:1
enter subdirectory name and size:temp
1
enter file name:aaa.txt
enter directory 2 names:test2
enter size of directories:1
enter subdirectory name and size:tmp2
1
enter file name:bbb.txt

dirname      size      subdirname      size      files
*****
test1        1         temp            1         aaa.txt
test2        1         tmp2            1         bbb.txt

```

RESULT: --

Successfully implemented File organization schemes for single level and two level directory using c language.