CHITKARA UNIVERSITY

Subject: Operating System

Subject Code: 22CS005

Session: 2022 – 23

Department: DCSE

Submitted By:

Kartavya

Group 24 B

2210990484

Submitted To:

Dr. Savita Walia

INDEX

CHITKARA UNIVERSITY

Experiment – 1

Aim: Installation and configuration of Linux OS.

Theory: Linux is an open source and free operating system to install which allows anyone with programming knowledge to modify and create its own operating system as per their requirements. Over many years, it has become more user-friendly and supports a lot of features such as

1. Reliable when used with servers
2. No need of antivirus
3. A Linux server can run nonstop with the boot for many years.

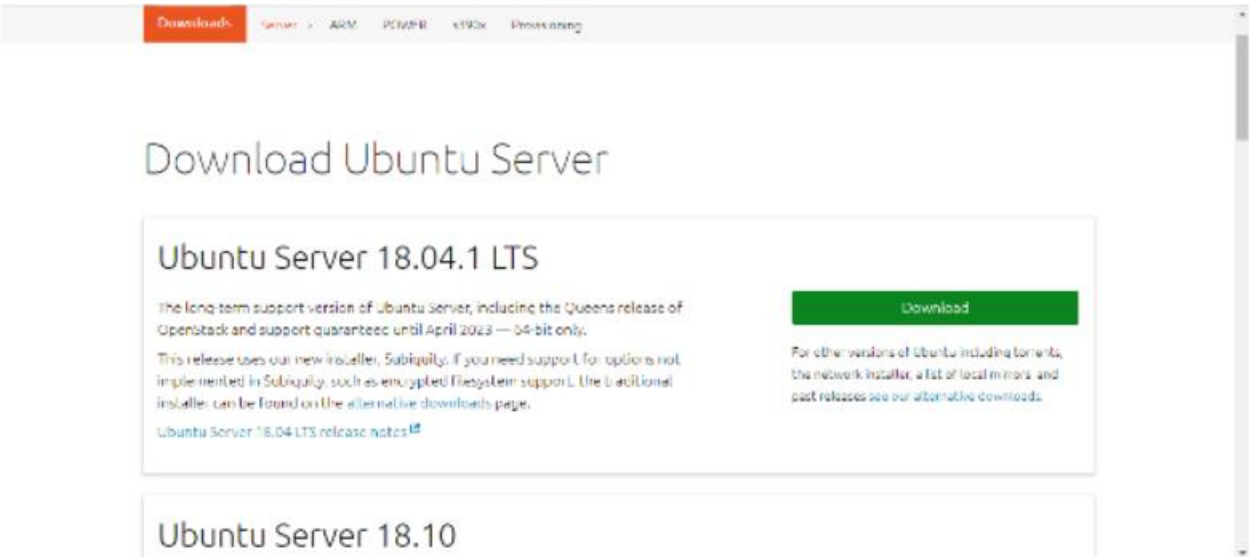
It has many distributions such as Ubuntu, Fedora, Redhat, Debian but all run on top of Linux server itself. Installation of every distribution is similar; thus, installation of Ubuntu is explained here.

Procedure:

Linux can be installed in the given 2 ways:

(A) Install Linux Using CD-ROM or USB Stick

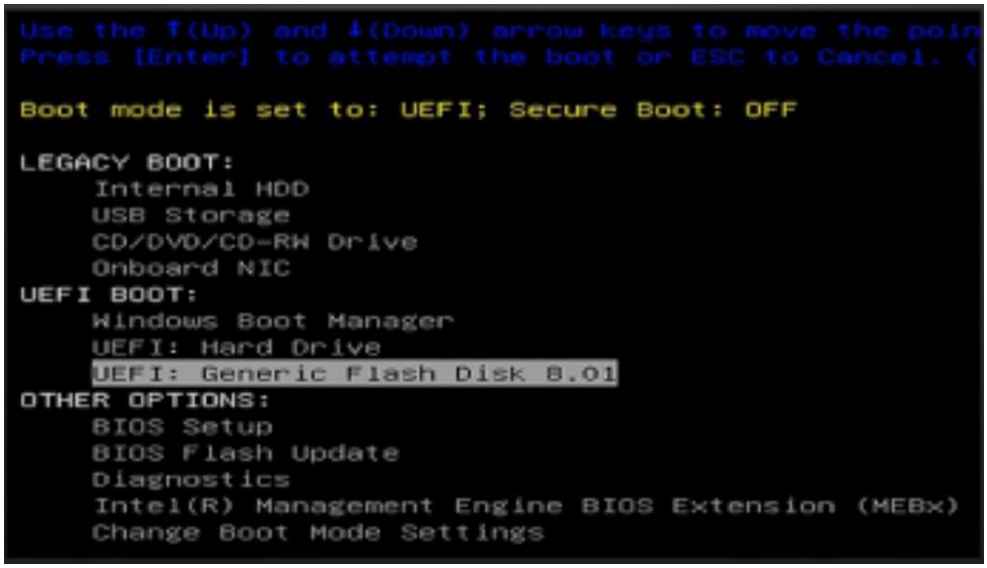
Download .iso or the ISO files on a computer from the internet and store it in the CD-ROM or USB stick after making it bootable using Pen Drive Linux and UNetBootin.



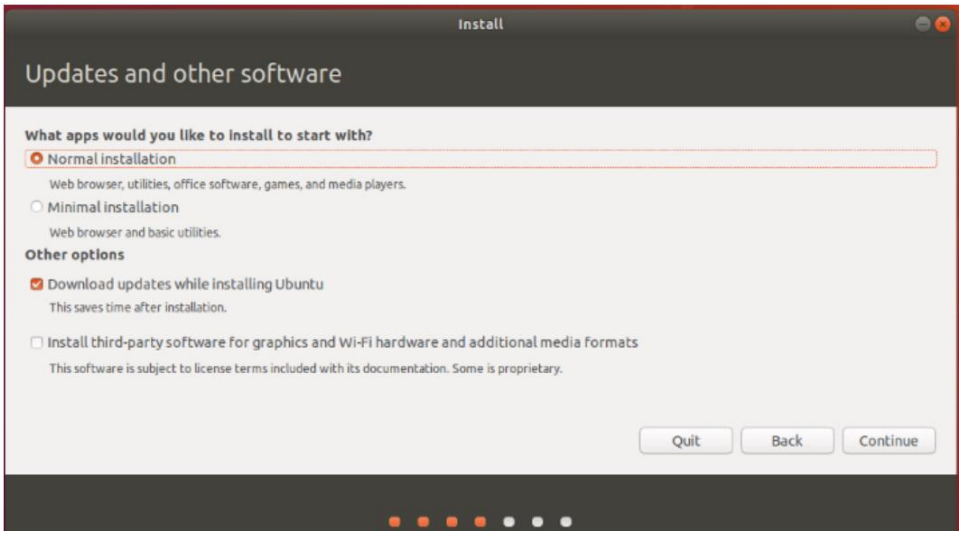
1. Boot into the USB Stick:

   You need to restart your computer after attaching CD –ROM or pen drive into the computer. Press enter at the time of boot, here select the CD-ROM or 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 you to select from various boot options before starting the system. All the options either it is USB or CD ROM or number of operating systems you will get a list from which you need to select one.

Note:-
You will see a new screen when your computer boots up called "GNU GRUB", a boot loader that handles installations for Linux. This screen will only appear in case there is more than one operating system.

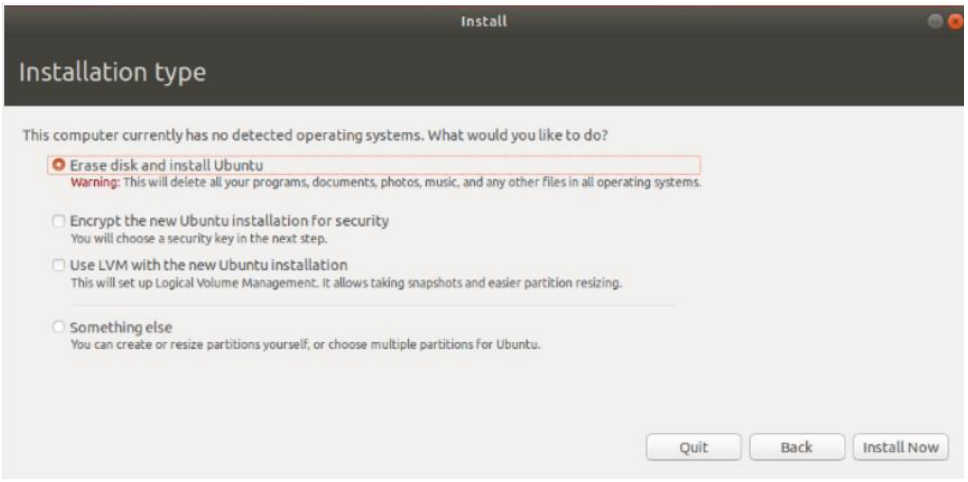


- Set the keyboard layout.
- Now you will be asked What apps would you like to install to start with Linux? The two options are 'Normal installation' and 'Minimal installation'.



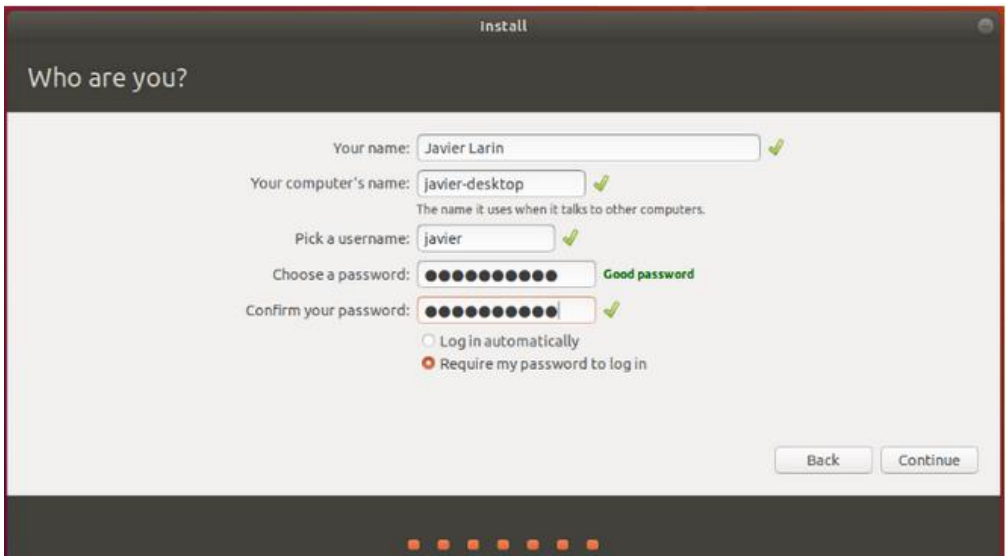
2. Derive Selection:

Select the drive for installation of OS to be completed. Select "Erase Disk and install Ubuntu" in case you want to replace the existing OS otherwise select "Something else" option and click INSTALL NOW.



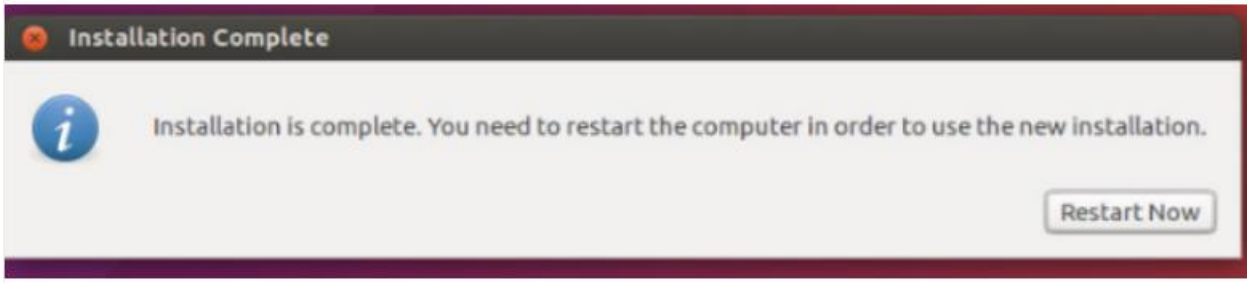
3. Start Installation:

- A small panel will ask for confirmation. Click Continue in case you don't want to change any information provided. Select your location on the map and install Linux.
- Provide the login details.



4. Complete the Installation Process:

After the installation is complete you will see a prompt to restart the computer.



You can also download drivers of your choice through the System Settings menu. Just follow these steps:

Additional Drivers > select the graphics driver from the list.

Many useful drivers will be available in the list, such as Wi-Fi drivers.

There are many other options also available to use and install Linux

(B) Install Linux Using Virtual Box VMWARE:

In this way, nothing will affect your Windows operating system.

What Are Requirements?

- Good internet connection
- At least 4GB RAM
- At least 12GB of free space

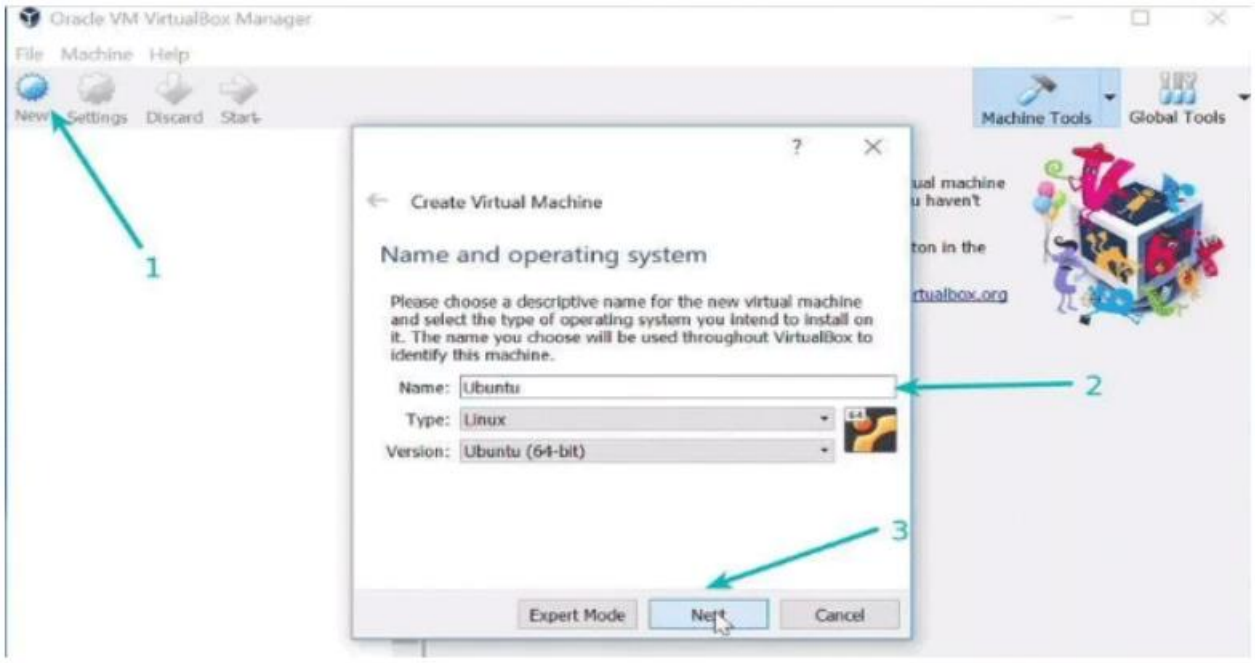
Steps:

1. Download the VIRTUAL BOX from original ORACLE VIRTUAL BOX site. You can refer below link

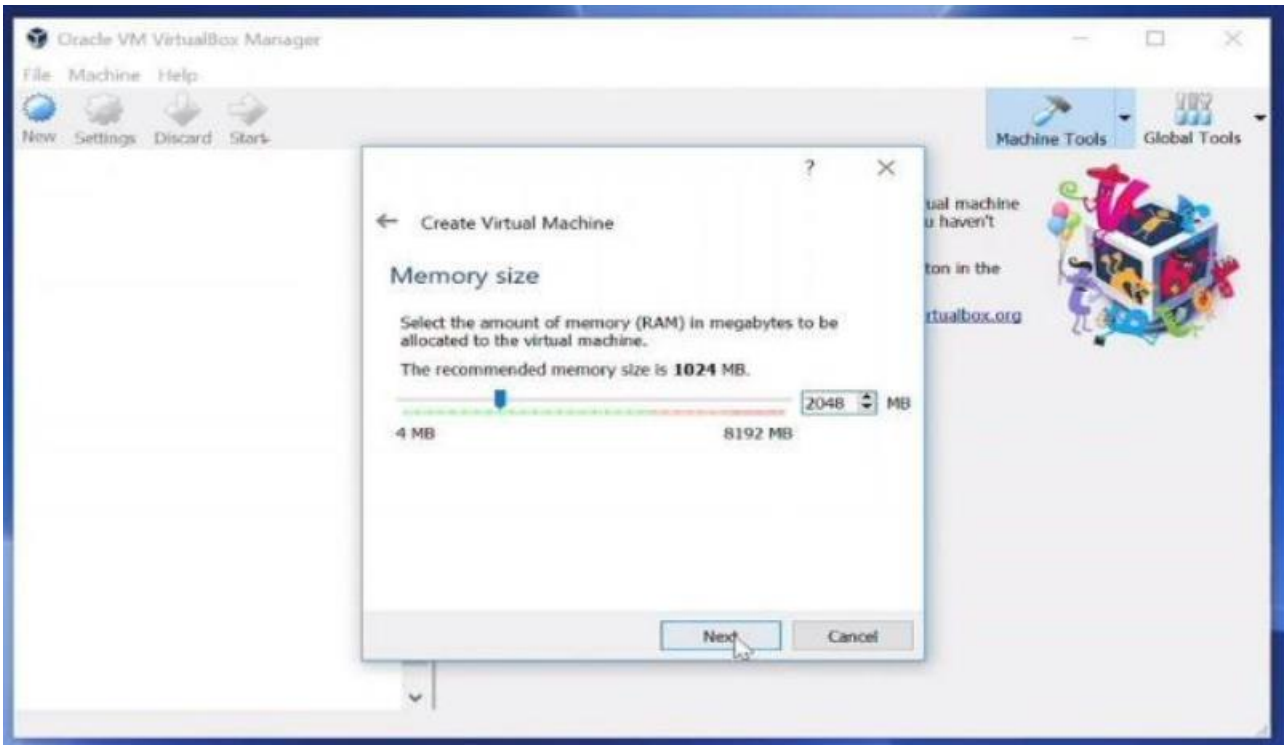
https://www.virtualbox.org/

2. Install Linux Using Virtual Box

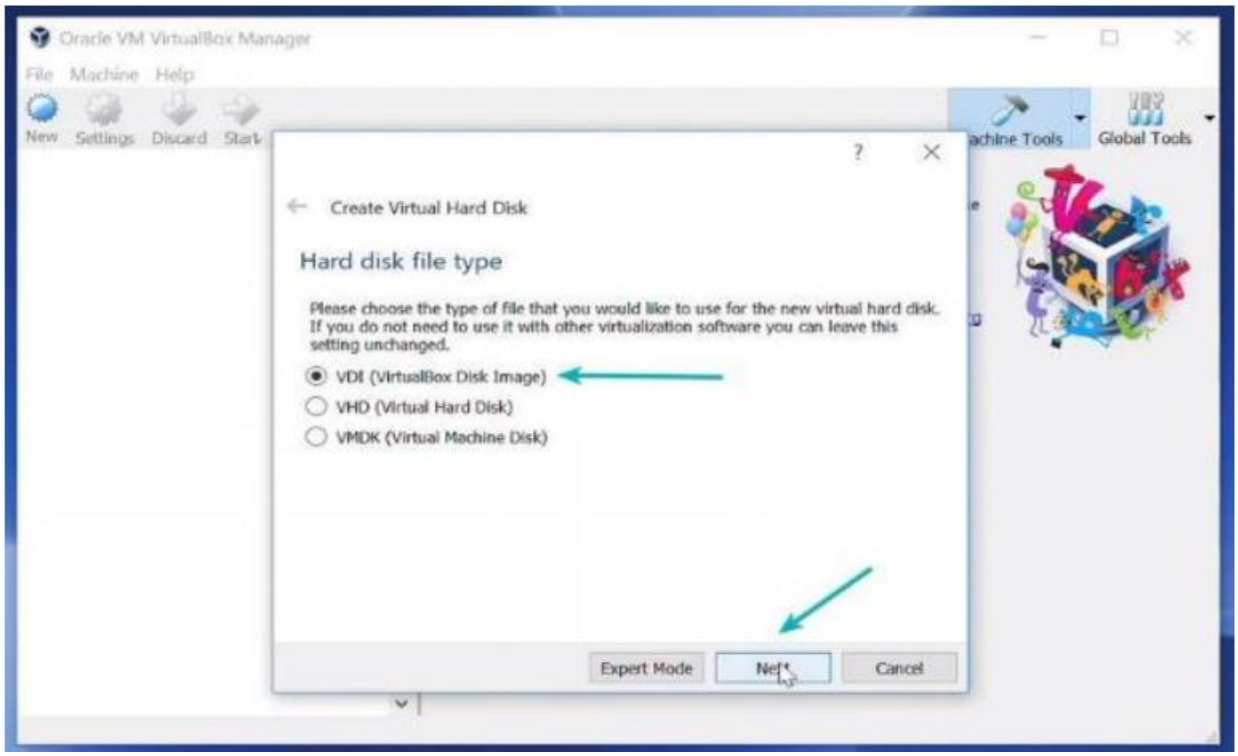
Use the .iso file or ISO file that can be downloaded from the internet and start the virtual box.



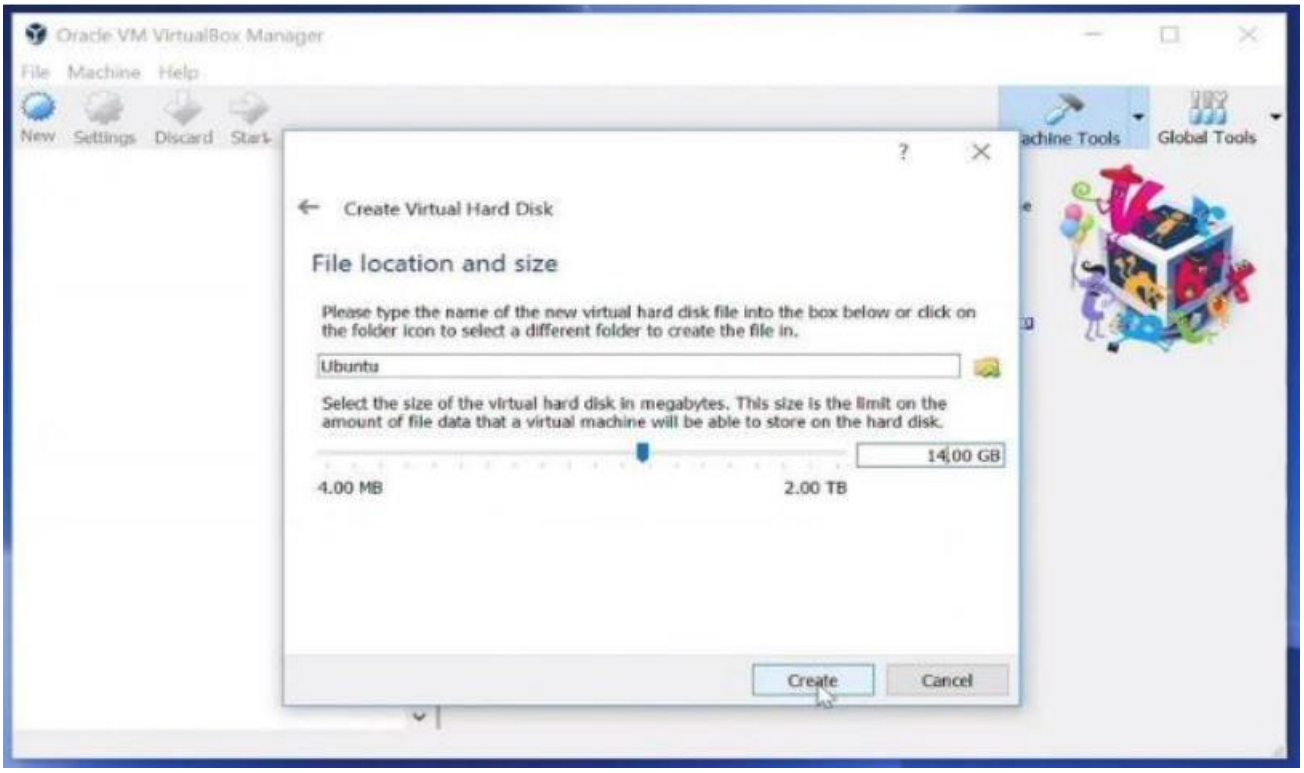
Here we need to allocate RAM to virtual OS. It should be 2 GB as per minimum requirement.



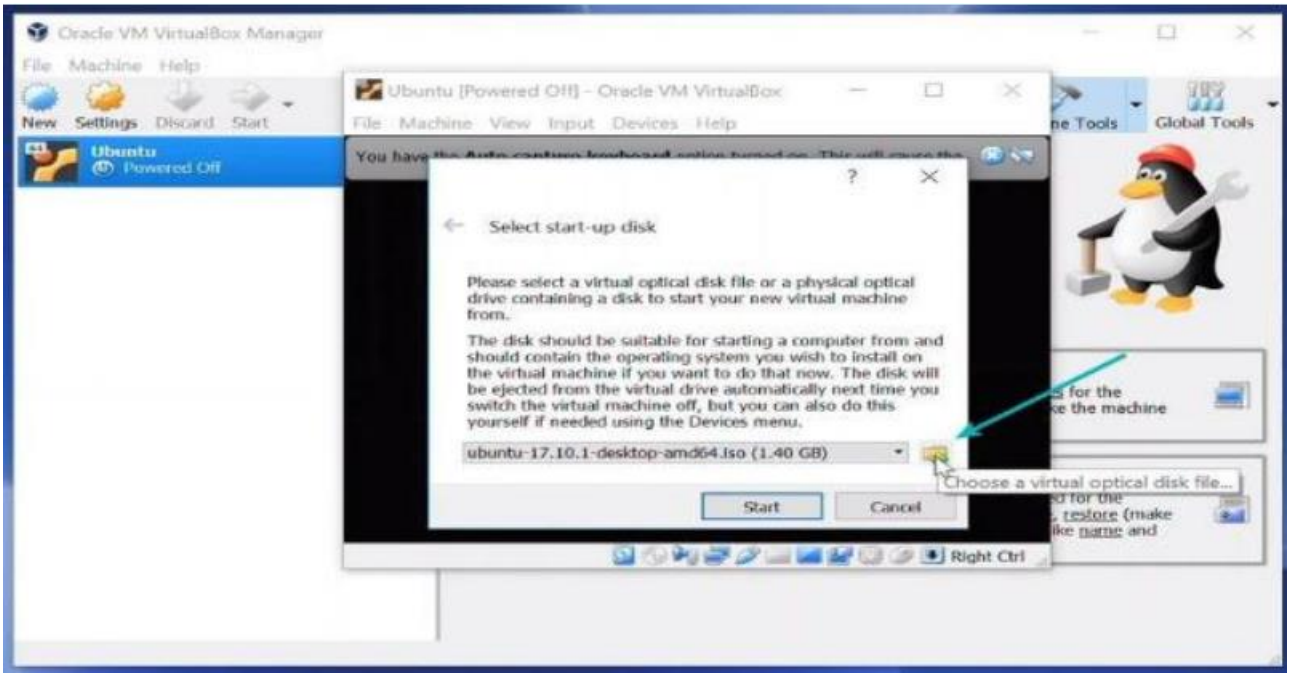
Choose an option under Create a virtual disk.



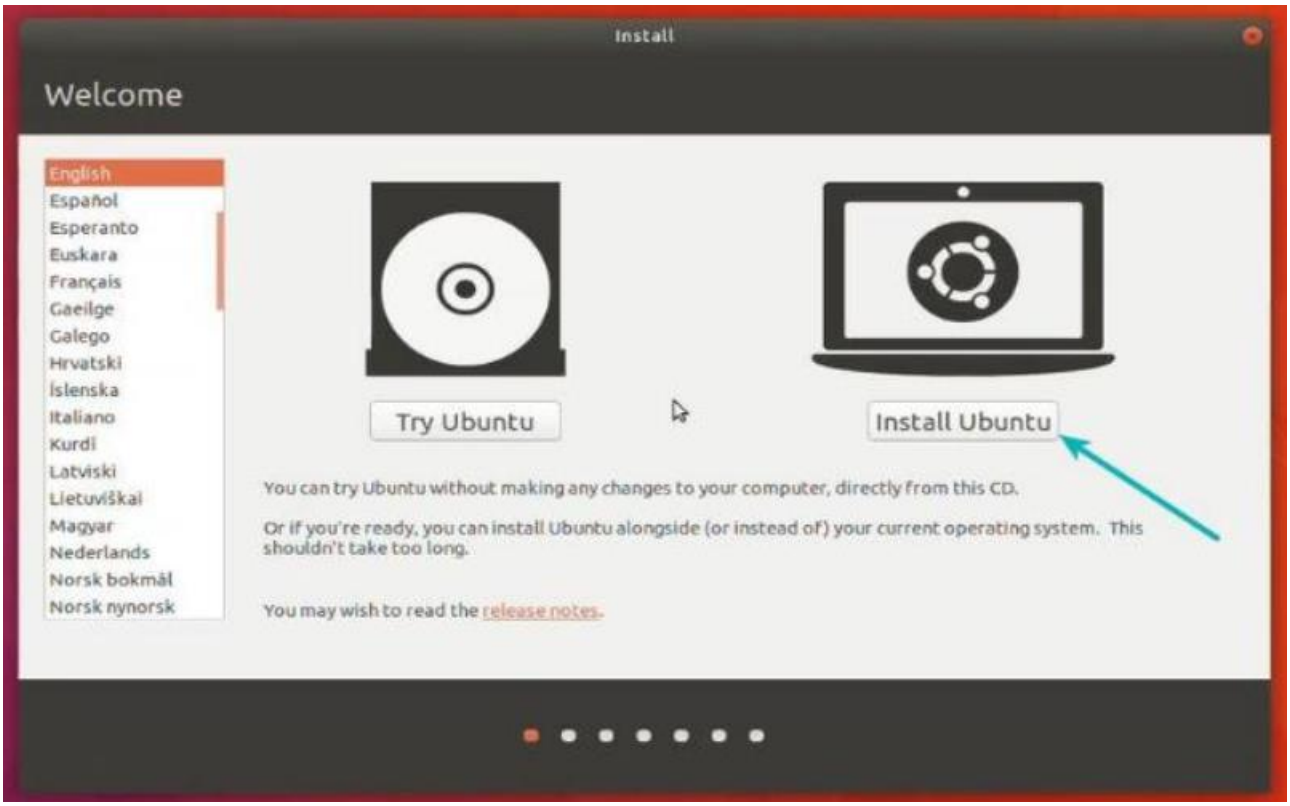
Choose a type of storage on physical hard disk. And choose the disk size(min 12 GB as per requirement)

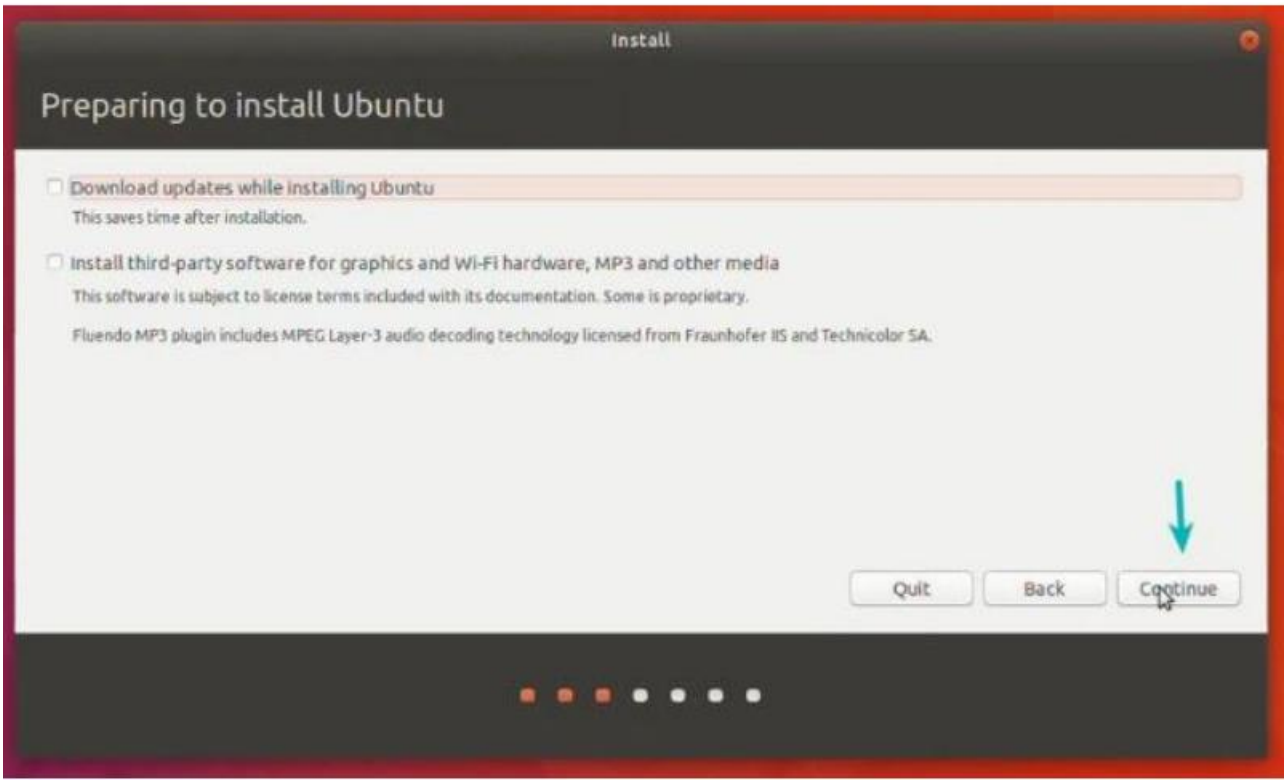


Click on create option and then click on the START button to start the virtual box and browse to the location of the .iso file of the OS.

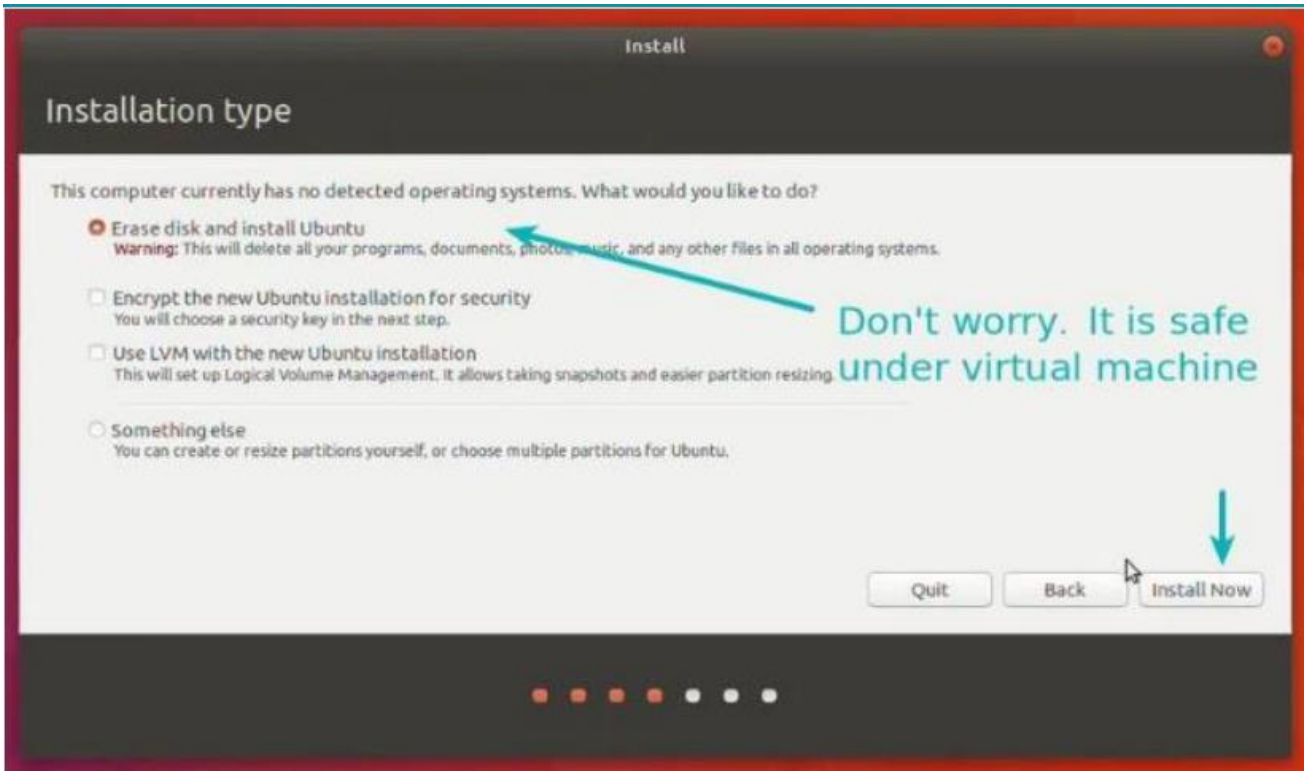


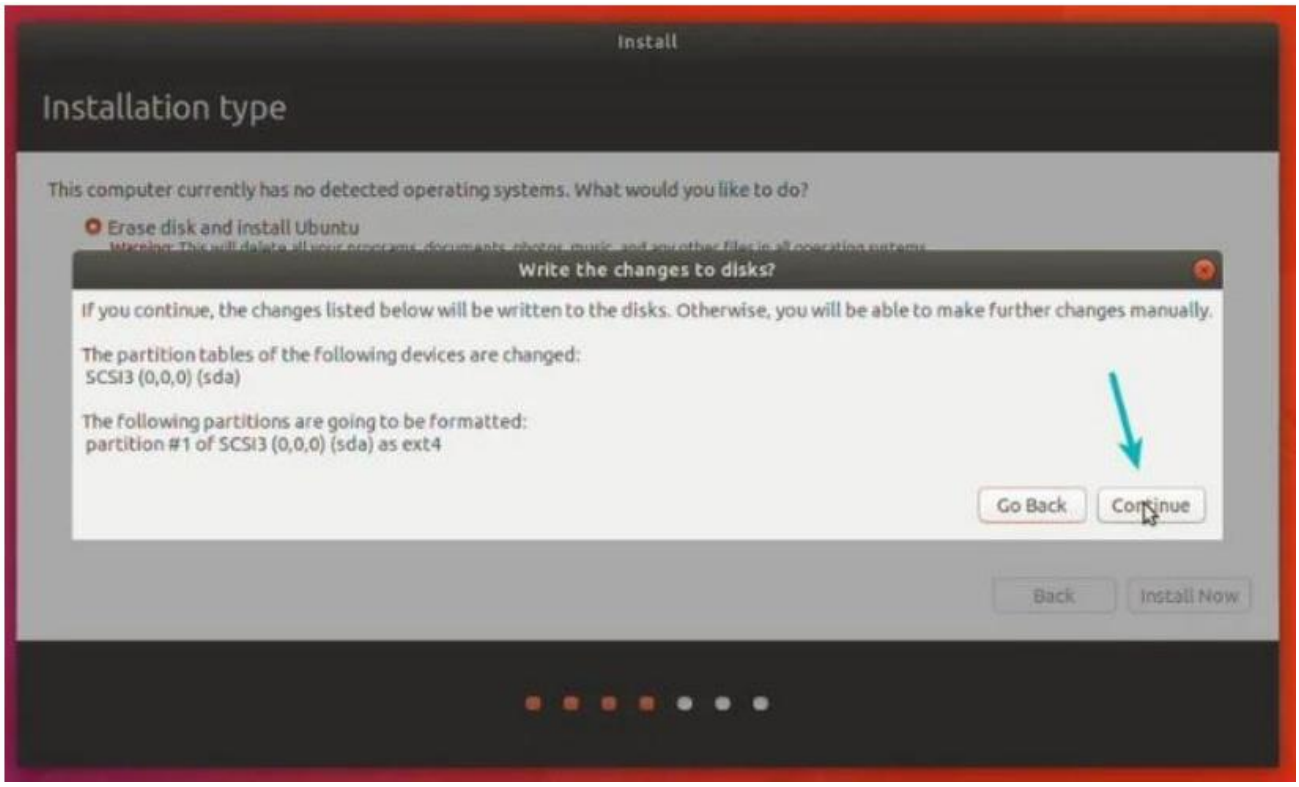
Now Linux OS will start, Click on install option.

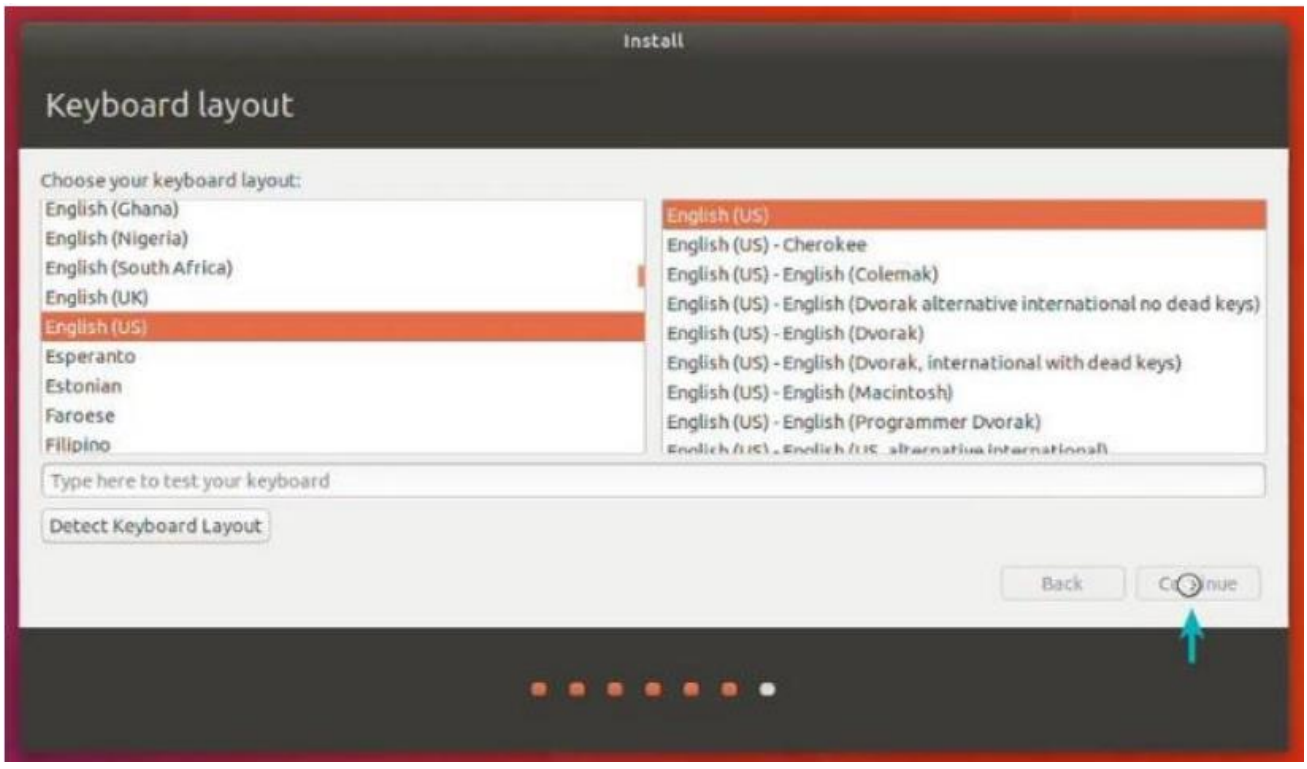




Select the drive for completing the OS installation. Select "Erase Disk and install Ubuntu" in case you want to replace the existing OS otherwise select "Something else" option and click INSTALL NOW.

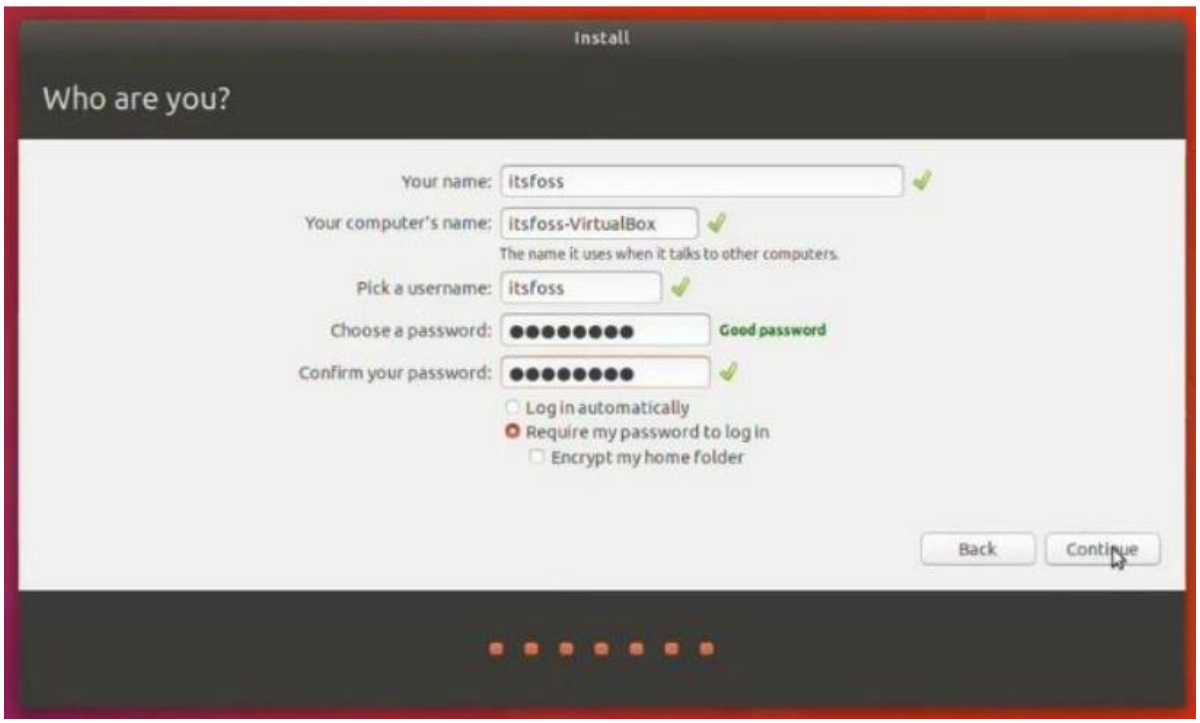




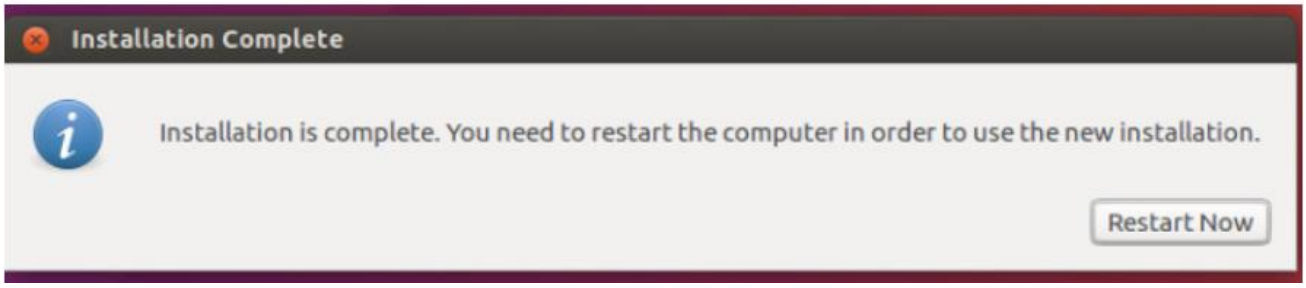


Click on Continue.

Choose a username and password.



You are almost done. It should take 10-15 minutes to complete the installation. Once the installation finishes, restart the system.



NOTE: In case of any issue close and again start the virtual box.

The Linux operating systems now offer millions of programs/applications to choose from, most of them free to install! Linux is also the OS of choice for Server environments due to its stability and reliability (Mega-companies like Amazon, Facebook, and Google use Linux for their Servers). It proves to be a good choice for everyone.

Experiment – 2

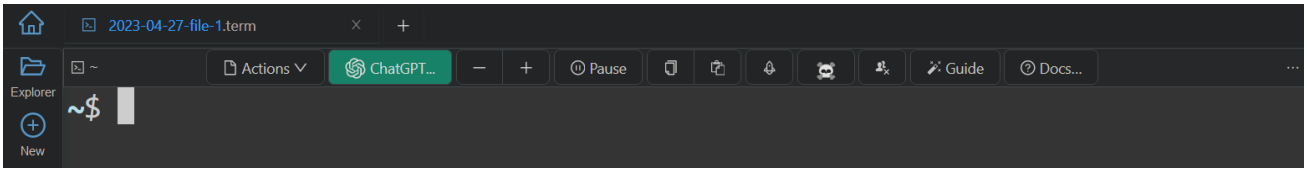
Aim: To illustrate basic of GCC compiler i.e. compilation and execution of a program on Linux terminal.

Theory: The term compiler refers to a piece of software that converts our source code from a high-level programming language to a low-level programming language (machine-level code) to build an executable program file and in Linux Operating Systems and compile C program in Linux, we'll need to install the GCC Compiler. In Ubuntu repositories, GCC Compiler is a part of the build-essential package, and this package is exactly what we will be installing in our Linux Operating System. If you're interested in learning more about the build-essential meta-package, You can refer here.

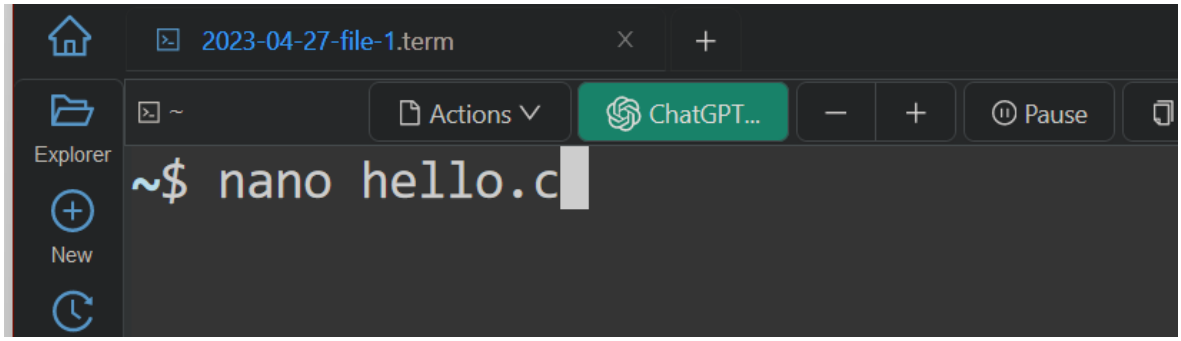
GCC Compiler (GNU Compiler Collection) is a collection of compilers and libraries for the programs written in C, C++, Ada, GO, D, Fortran, and Objective-C programming languages and is distributed under the GNU General Public License.

Procedure:

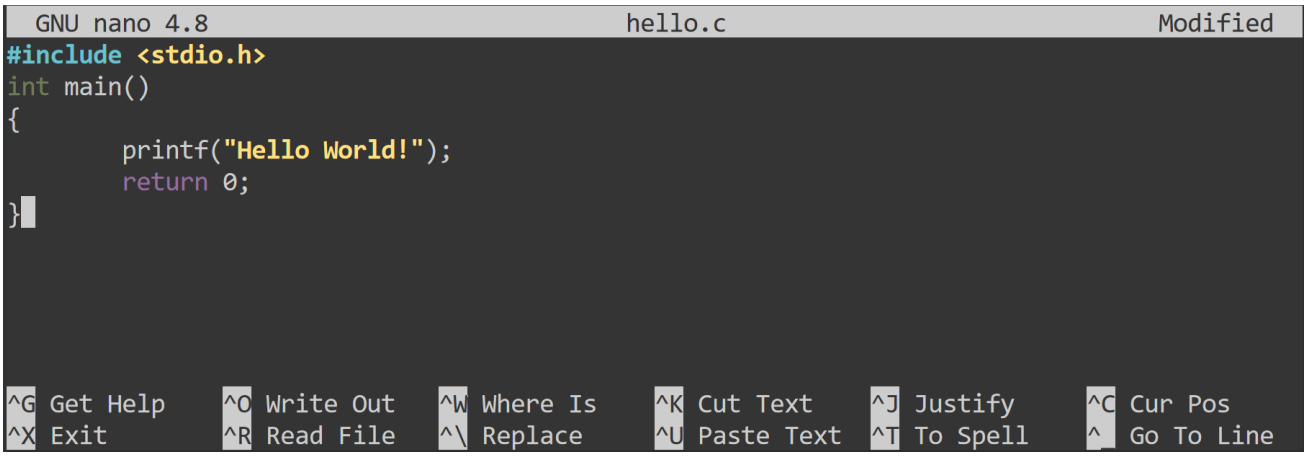
1. Open Linux Terminal.



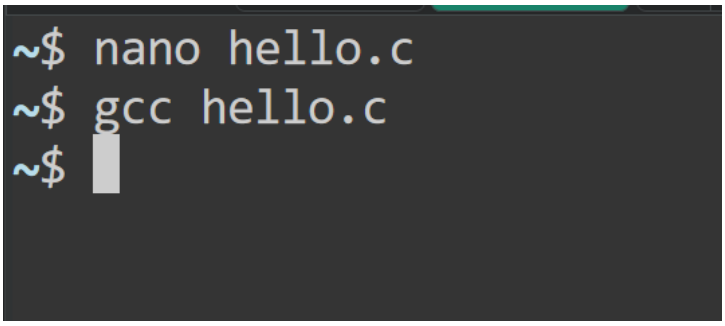
2. Enter the command “nano hello.c” to open a text file editor where we will enter a C program which we have named as “hello.c”.



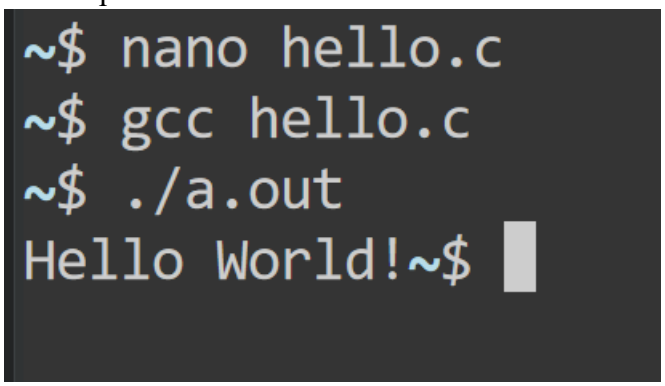
3. You’ll enter the text file editor. Type a simple program to print “Hello World!”.



4. To exit the editor, press Ctrl+X, then Ctrl+Y.
5. Enter the command “gcc hello.c” to compile the C program we wrote earlier.



6. To generate the output of the compiled C program, type “./a.out” in the terminal. “Hello World!” will be printed in the terminal.



Experiment – 3

Aim: Implement the following system call using C language: getrusage, uname, sysinfo and gettimeofday.

Theory:

The **getrusage()** returns current resource usages, for a *who* of either RUSAGE_SELF or RUSAGE_CHILDREN. The former asks for resources used by the current process, the latter for resources used by those of its children that have terminated and have been waited for.

The **uname()** function retrieves information identifying the operating system you are running on. The argument name points to a memory area where a structure describing the operating system the process is running on can be stored.

The **gettimeofday()** function retrieves the current Coordinated Universal Time (UTC) and places it in the timeval structure pointed to by *tp*. If *tzp* is not NULL, the time zone information is returned in the time zone structure pointed to by *tzp*.

The **sysinfo()** function copies information relating to the operating system on which the process is executing into the buffer pointed to by *buf*. It can also set certain information where appropriate *command*s are available. The *count* parameter indicates the size of the buffer.

Procedure:

The following C programs are written in textfile editor and their output in the Linux Terminal shows the functioning of the given system calls:

(A) getrusage( ):

```c
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
#include <stdio.h>

int main()
 {
  struct rusage usage;
  struct timeval start, end;
  int i, j, k = 0;

  getrusage(RUSAGE_SELF, &usage);
  start = usage.ru_stime;
  for (i = 0; i < 10000; i++) {
    /* Double loop for more interesting results. */
    for (j = 0; j < 10000; j++) {
      k += 20;
    }
  }
  getrusage(RUSAGE_SELF, &usage);
  end = usage.ru_stime;

  printf("Started at: %ld.%lds\n", start.tv_sec, start.tv_usec);
  printf("Ended at: %ld.%lds\n", end.tv_sec, end.tv_usec);
  return 0;
}
```

```
~$ nano hey.c
~$ gcc hey.c
~$ ./a.out
Started at: 0.0s
Ended at: 0.0s
~$
```

(B) uname( ):

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/utsname.h>

int main( void )
  {
    struct utsname sysinfo;

    if( uname( &sysinfo ) == -1 ) {
       perror( "uname" );
       return EXIT_FAILURE;
    }
    printf( "system name  : %s\n", sysinfo.sysname );
    printf( "node name    : %s\n", sysinfo.nodename );
    printf( "release name : %s\n", sysinfo.release );
    printf( "version name : %s\n", sysinfo.version );
    return EXIT_SUCCESS;
  }
```

```
~$ nano new.c
~$ gcc new.c
~$ ./a.out
system name  : Linux
node name    : project-267a0ad2-085f-44e1-9c46-bacc69d5a879
release name : 5.13.0-1033-gcp
version name : #40~20.04.1-Ubuntu SMP Tue Jun 14 00:44:12 UTC 2022
~$
```

(C) sysinfo( )

```
 GNU nano 4.8
#include <linux/kernel.h>
#include <stdio.h>
#include <sys/sysinfo.h>

int main ()
{
 /* Conversion constants. */
 const long minute = 60;
 const long hour = minute * 60;
 const long day = hour * 24;
 const double megabyte = 1024 * 1024;
 /* Obtain system statistics. */
 struct sysinfo si;
 sysinfo (&si);
 /* Summarize interesting values. */
 printf ("system uptime : %ld days, %ld:%02ld:%02ld\n",
     si.uptime / day, (si.uptime % day) / hour,
     (si.uptime % hour) / minute, si.uptime % minute);
 printf ("total RAM    : %5.1f MB\n", si.totalram / megabyte);
 printf ("free RAM     : %5.1f MB\n", si.freeram / megabyte);
 printf ("process count : %d\n", si.procs);
 return 0;
}
```

```
~$ nano new.c
~$ gcc new.c
~$ ./a.out
system uptime : 0 days, 4:43:37
total RAM    : 32104.3 MB
free RAM    : 15298.8 MB
process count : 2469
~$
```

(D) gettimeofday( )

```c
#include <stdio.h>
#include <sys/time.h>
#include <time.h>

int main()
{
    struct timeval tv;
    struct timezone tz;
    struct tm *today;
    int zone;

    gettimeofday(&tv,&tz);

    /* get time details */
    today = localtime(&tv.tv_sec);
    printf("It's %d:%0d:%0d.%d\n",
            today->tm_hour,
            today->tm_min,
            today->tm_sec,
            tv.tv_usec
          );
    /* set time zone value to hours, not minutes */
    zone = tz.tz_minuteswest/60;
    /* calculate for zones east of GMT */
    if( zone > 12 )
        zone -= 24;
    printf("Time zone: GMT %+d\n",zone);
    printf("Daylight Saving Time adjustment: %d\n",tz.tz_dsttime);

    return(0);
}
```

```
~$ nano hey.c
~$ gcc hey.c
hey.c: In function ‘main’:
hey.c:16:30: warning: format ‘%d’ expects argument of type ‘int’, but argument 5 has type ‘__suseconds_t’ {aka ‘long int’} [-Wformat=]
   16 |     printf("It's %d:%0d:%0d.%d\n",
      |                             ~^
      |                              |
      |                              int
      |                             %ld
......
   20 |           tv.tv_usec
      |           ~~~~~~~~~~
      |             |
      |             __suseconds_t {aka long int}
~$ ./a.out
It's 17:22:34.721330
Time zone: GMT +0
Daylight Saving Time adjustment: 0
~$
```

Experiment – 4

Aim: Write a program to print process id and parent id.

Theory: In Linux, each running task is known as "Process". Kernal assigns each process running in memory a unique ID called Process ID or PID. PID distinguishes one process from another running process.

The getpid() used to get the process id of the current process. Every process has a parent process. The process ID of the parent process can find using getppid(). The getpid() and getppid() are declared under <unistd.h>. If we don't include <unistd.h> then a warning occurs. To fix this warning <unistd.h> must be included.

Procedure:

```c
#include <sys/time.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
        int p_id, p_pid;
        p_id = getpid();
        p_pid = getppid();
        printf("ProcessID: %d \n", p_id);
        printf("Parent process ID: %d \n", p_pid);
        return 0;
}
```

```
~$ nano new.c
~$ gcc new.c
new.c: In function ‘main’:
new.c:8:16: warning: implicit declaration of function ‘getpid’ [-Wimplicit-function-declaration]
    8 |         p_id = getpid();
      |                ^~~~~~
new.c:9:17: warning: implicit declaration of function ‘getppid’ [-Wimplicit-function-declaration]
    9 |         p_pid = getppid();
      |                 ^~~~~~~
~$ ./a.out
ProcessID: 1169
Parent process ID: 1086
~$
```

Experiment – 5

Aim: Write a program to create 5 child processes of a parent processes and print their id using fork( ) system calls.

Theory: Fork system call is used for creating a new process, which is called ***child process***, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork( ) system call. A child process uses the same pc(program counter), same CPU registers, same open files which use in the parent process. It takes no parameters and returns an integer value. Below are different values returned by fork( ). ***Negative Value***: creation of a child process was unsuccessful. ***Zero***: Returned to the newly created child process. ***Positive value***: Returned to parent or caller. The value contains process ID of newly created child process.

Procedure:

```
 GNU nano 4.8                                                    nec.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t pid;
    int i;

    for (i = 0; i < 5; i++) {
        pid = fork();  // create a child process

        if (pid < 0) {  // error occurred
            printf("Fork failed!\n");
            exit(-1);
        }
        else if (pid == 0) {  // child process
            printf("Child process %d with ID %d\n", i+1, getpid());
            exit(0);
        }
        else {  // parent process
            printf("Parent process with ID %d created child process %d with ID %d\n", getpid(), i+1, pid);
        }
    }

    return 0;
}
```

```
~$ nano nec.c
~$ gcc nec.c
~$ ./a.out
Parent process with ID 1313 created child process 1 with ID 1314
Child process 1 with ID 1314
Parent process with ID 1313 created child process 2 with ID 1315
Child process 2 with ID 1315
Parent process with ID 1313 created child process 3 with ID 1316
Parent process with ID 1313 created child process 4 with ID 1317
Child process 4 with ID 1317
Parent process with ID 1313 created child process 5 with ID 1318
Child process 5 with ID 1318
Child process 3 with ID 1316
~$
```

Experiment – 6

Aim: Implement FCFS, SJF, priority scheduling and round robin scheduling in C Language.

Theory:

First Come, First Served (FCFS) also known as First In, First Out (FIFO) is the CPU scheduling algorithm in which the CPU is allocated to the processes in the order they are queued in the ready queue. FCFS follows non-preemptive scheduling which mean once the CPU is allocated to a process it does not leave the CPU until the process will not get terminated or may get halted due to some I/O interrupt.

Shortest Job First (SJF) is an algorithm in which the process having the smallest execution time is chosen for the next execution. This scheduling method can be preemptive or non-preemptive. It significantly reduces the average waiting time for other processes awaiting execution. The full form of SJF is Shortest Job First.

Priority Scheduling is a method of scheduling processes that is based on priority. In this algorithm, the scheduler selects the tasks to work as per the priority. The processes with higher priority should be carried out first, whereas jobs with equal priorities are carried out on a round-robin or FCFS basis. Priority depends upon memory requirements, time requirements, etc.

Round Robin Scheduling comes from the round-robin principle, where each person gets an equal share of something in turns. It is the oldest, simplest scheduling algorithm, which is mostly used for multitasking. In Round-robin scheduling, each ready task runs turn by turn only in a cyclic queue for a limited time slice. This algorithm also offers starvation free execution of processes.

Procedure:

(A) FCFS:

```c
#include <stdio.h>

void waiting_time(int processes[], int n, int bt[], int wt[]) {
    wt[0] = 0;
    for (int i = 1; i < n ; i++ )
        wt[i] = bt[i-1] + wt[i-1] ;
}

void turnaround_time(int processes[], int n, int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n ; i++)
        tat[i] = bt[i] + wt[i];
}

void avg_time(int processes[], int n, int bt[]) {
    int wt[n], tat[n], total_wt = 0, total_tat = 0;
    waiting_time(processes, n, bt, wt);
    turnaround_time(processes, n, bt, wt, tat);
    printf("Processes Burst time Waiting time Turn around time\n");
    for (int i=0; i<n; i++) {
        total_wt += wt[i];
        total_tat += tat[i];
        printf(" %d\t\t%d\t\t%d\t\t%d\n", i+1, bt[i], wt[i], tat[i]);
    }
    printf("Average waiting time = %d\n", total_wt / n);
    printf("Average turn around time = %d\n", total_tat / n);
}

int main() {
    int processes[] = { 1, 2, 3 };
    int n = sizeof processes / sizeof processes[0];
    int burst_time[] = {10, 5, 8};
    avg_time(processes, n, burst_time);
    return 0;
}
```

```
~$ nano new.c
~$ gcc new.c
~$ ./a.out
Processes Burst time Waiting time Turn around time
 1              10              0               10
 2              5               10              15
 3              8               15              23
Average waiting time = 8
Average turn around time = 16
~$
```

(B) SJF:

```
 GNU nano 4.8
#include <stdio.h>

int main() {
    int A[100][4], n, i, j, total = 0, index, temp;
    float avg_wt, avg_tat;
    printf("Enter number of process: ");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        printf("P%d: ", i + 1);
        scanf("%d", &A[i][1]);
        A[i][0] = i + 1;
    }
    for (i = 0; i < n; i++) {
        index = i;
        for (j = i + 1; j < n; j++)
            if (A[j][1] < A[index][1])
                index = j;
        temp = A[i][1], A[i][1] = A[index][1], A[index][1] = temp;
        temp = A[i][0], A[i][0] = A[index][0], A[index][0] = temp;
    }
    A[0][2] = 0;
    for (i = 1; i < n; i++) {
        A[i][2] = 0;
        for (j = 0; j < i; j++)
            A[i][2] += A[j][1];
        total += A[i][2];
    }
    avg_wt = (float)total / n, total = 0;
    printf("P\tBT\tWT\tTAT\n");
    for (i = 0; i < n; i++) {
        A[i][3] = A[i][1] + A[i][2];
        total += A[i][3];
        printf("P%d\t%d\t%d\t%d\n", A[i][0], A[i][1], A[i][2], A[i][3]);
    }
    avg_tat = (float)total / n;
    printf("Average Waiting Time= %f\nAverage Turnaround Time= %f", avg_wt, avg_tat);
    return 0;
}
```

```
~$ nano new.c
~$ gcc new.c
~$ ./a.out
Enter number of process: 3
P1: 3
P2: 2
P3: 1
P       BT      WT      TAT
P3      1       0       1
P2      2       1       3
P1      3       3       6
Average Waiting Time= 1.333333
Average Turnaround Time= 3.333333~$
```

(C) Priority Scheduling:

```c
#include <stdio.h>

// structure representing a process
struct process {
    int process_id;      // id of the process
    int burst_time;      // time required for execution
    int waiting_time;    // waiting time of the process
    int turn_around_time;// total time of execution
    int priority;        // priority of the process
};

// function to sort processes by priority
void sort_processes_by_priority(struct process processes[], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (processes[i].priority > processes[j].priority) {
                // swap processes
                struct process temp = processes[i];
                processes[i] = processes[j];
                processes[j] = temp;
            }
        }
    }
}

int main() {
    int n;  // total number of processes
    int total_waiting_time = 0;  // total waiting time of all processes
```

```c
    int total_turn_around_time = 0;  // total turn around time of all processes
    float avg_waiting_time;  // average waiting time of all processes
    float avg_turn_around_time;  // average turn around time of all processes

    printf("Enter the total number of processes: ");
    scanf("%d", &n);

    // create an array of processes
    struct process processes[n];

    // get process details
    for (int i = 0; i < n; i++) {
        printf("\nProcess %d\n", i + 1);
        printf("Enter the burst time: ");
        scanf("%d", &processes[i].burst_time);
        printf("Enter the priority: ");
        scanf("%d", &processes[i].priority);
        processes[i].process_id = i + 1;
    }

    // sort processes by priority
    sort_processes_by_priority(processes, n);

    // calculate waiting time and turn around time of each process
    for (int i = 0; i < n; i++) {
        processes[i].waiting_time = 0;
        for (int j = 0; j < i; j++) {
            processes[i].waiting_time += processes[j].burst_time;
```

```c
        processes[i].waiting_time = 0;
        for (int j = 0; j < i; j++) {
            processes[i].waiting_time += processes[j].burst_time;
        }
        total_waiting_time += processes[i].waiting_time;
        processes[i].turn_around_time = processes[i].burst_time + processes[i].waiting_time;
        total_turn_around_time += processes[i].turn_around_time;
    }

    // calculate average waiting time and average turn around time
    avg_waiting_time = (float) total_waiting_time / n;
    avg_turn_around_time = (float) total_turn_around_time / n;

    // print process details
    printf("\n\nProcess_id \t Burst Time \t Priority \t Waiting Time \t  Turnaround Time\n");
    printf("-------------------------------------------------------------------------\n");
    for (int i = 0; i < n; i++) {
        printf("%10d \t%10d \t%10d \t%10d \t%10d\n", processes[i].process_id, processes[i].burst_time,
               processes[i].priority, processes[i].waiting_time, processes[i].turn_around_time);
    }

    // print average waiting time and average turn around time
    printf("\n\nAverage waiting time: %.2f\n", avg_waiting_time);
    printf("Average turn around time: %.2f\n", avg_turn_around_time);

    return 0;
}
```

```
~$ nano new.c
~$ gcc new.c
~$ ./a.out
Enter the total number of processes: 3

Process 1
Enter the burst time: 3
Enter the priority: 2

Process 2
Enter the burst time: 2
Enter the priority: 1

Process 3
Enter the burst time: 4
Enter the priority: 3


Process_id       Burst Time      Priority        Waiting Time     Turnaround Time
-------------------------------------------------------------------------
         2               2               1               0               2
         1               3               2               2               5
         3               4               3               5               9


Average waiting time: 2.33
Average turn around time: 5.33
~$
```

(D) Round Robin Scheduling:

```c
#include<stdio.h>

int main() {
  int n, quantum;
  printf("Enter the total number of processes: ");
  scanf("%d", &n);
  printf("Enter the time quantum: ");
  scanf("%d", &quantum);

  int arrival_time[n], burst_time[n], remaining_time[n], completion_time[n], turnaround_time[n], waiting_time[n], total_waiting_time = 0, total_turnaround_time = 0;
  printf("\nEnter the arrival time and burst time of each process:\n");
  for(int i = 0; i < n; i++) {
    printf("\nProcess %d:\n", i + 1);
    printf("Arrival time: ");
    scanf("%d", &arrival_time[i]);
    printf("Burst time: ");
    scanf("%d", &burst_time[i]);
    remaining_time[i] = burst_time[i];
  }

  int time = 0, completed = 0, current_process = -1, current_time = 0;
  while(completed < n) {
    current_time = 0;
    for(int i = 0; i < n; i++) {
      if(remaining_time[i] > 0 && arrival_time[i] <= time) {
        if(current_process == -1 || remaining_time[i] < remaining_time[current_process])
          current_process = i;
      }
    }
    if(current_process == -1) {
      time++;
      continue;
    }
    if(remaining_time[current_process] <= quantum) {
      current_time = remaining_time[current_process];
      remaining_time[current_process] = 0;
      completed++;
    }
    else {
      current_time = quantum;
      remaining_time[current_process] -= quantum;
    }
    time += current_time;
    completion_time[current_process] = time;
  }

  for(int i = 0; i < n; i++) {
    turnaround_time[i] = completion_time[i] - arrival_time[i];
    waiting_time[i] = turnaround_time[i] - burst_time[i];
    total_waiting_time += waiting_time[i];
    total_turnaround_time += turnaround_time[i];
  }

  printf("\nProcess\t\tArrival Time\tBurst Time\tCompletion Time\tTurnaround Time\tWaiting Time\n");
  for(int i = 0; i < n; i++) {
    printf("P%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", i + 1, arrival_time[i], burst_time[i], completion_time[i], turnaround_time[i], waiting_time[i]);
  }
  printf("\nAverage Waiting Time: %f\n", (float)total_waiting_time / n);
  printf("Average Turnaround Time: %f\n", (float)total_turnaround_time / n);
  return 0;
}
```

```
Enter the total number of processes: 4
Enter the time quantum: 2

Enter the arrival time and burst time of each process:

Process 1:
Arrival time: 0
Burst time: 5

Process 2:
Arrival time: 1
Burst time: 2

Process 3:
Arrival time: 2
Burst time: 4

Process 4:
Arrival time: 3
Burst time: 1

Process         Arrival Time    Burst Time      Completion Time Turnaround Time Waiting Time
P1              0               5               2               2                       -3
P2              1               2               4               3                       1
P3              2               4               0               -2                      -6
P4              3               1               0               -3                      -4

Average Waiting Time: -3.000000
Average Turnaround Time: 0.000000
~$
```

Experiment – 7

Aim: Implement the basic and user status commands like: su, sudo, man, help, history, who, whoami, id, uname, uptime, free, tty, cal, date, hostname, reboot, clear**.**

1. **su** - Switch user or become superuser
   - Usage: `su [OPTIONS] [USERNAME]`
   - Options:
     - `-` - Start a login shell for the new user
   - Example: `su user1` - Switch to the user account named "user1"
2. **sudo** - Execute a command as another user or as superuser
   - Usage: `sudo [OPTIONS] COMMAND`
   - Options:
     - `-u` - Run the command as another user
   - Example: `sudo apt-get update` - Run the "apt-get update" command with superuser privileges
3. **man** - Display manual pages for a command
   - Usage: `man COMMAND`
   - Options:
     - None
   - Example: `man ls` - Display the manual pages for the "ls" command
4. **help** - Display help information for a command
   - Usage: `help [COMMAND]`
   - Options:
     - None
   - Example: `help cd` - Display help information for the "cd" command
5. **history** - Display command history
   - Usage: `history [OPTIONS]`
   - Options:
     - `-c` - Clear the command history
     - `-a` - Append new commands to the history file immediately
   - Example: `history` - Display the command history for the current use.
6. **who** - Display information about currently logged in users

- Usage: `who [OPTIONS]`
- Options:
  - `-q` - Display only the number of logged in users
- Example: `who` - Display the information about currently logged in users

7. **whoami** - Display the current user name
   - Usage: `whoami`
   - Options:
     - None
   - Example: `whoami` - Display the current user name
8. **id** - Display the current user ID and group ID
   - Usage: `id [OPTIONS] [USERNAME]`
   - Options:
     - `-g` - Display only the group ID
     - `-u` - Display only the user ID
   - Example: `id` - Display the current user ID and group ID
9. **uname** - Display system information
   - Usage: `uname [OPTIONS]`
   - Options:
     - `-a` - Display all system information
     - `-r` - Display only the kernel release information
   - Example: `uname -a` - Display all system information
10. **uptime** - Display system uptime
    - Usage: `uptime [OPTIONS]`
    - Options:
      - None
    - Example: `uptime` - Display the system uptime
11. **free** - Display system memory usage
    - Usage: `free [OPTIONS]`
    - Options:
      - `-h` - Display memory usage in a human-readable format

- `-s` - Display memory usage at regular intervals

- Example: `free -h` - Display system memory usage in a human-readable format

12. **tty** - Display the file name of the terminal

- Usage: `tty`

- Options:

  - None

- Example: `tty` - Display the file name of the terminal

13. **cal** - Display calendar for a month or year

- Usage: `cal [MONTH] [YEAR]`

- Options:

  - None

- Example: `cal` - Display the calendar for the current month

14. **date** - Display the current date and time

- Usage: `date [OPTIONS]`

- Options:

  - `-u` - Display the date and time in UTC (Coordinated Universal Time)

  - `-R` - Display the date and time in RFC 2822 format

- Example: `date` - Display the current date and time

15. **hostname** - Display the system's hostname

- Usage: `hostname [OPTIONS]`

- Options:

  - `-a` - Display all the IP addresses associated with the hostname

  - `-i` - Display the IP address associated with the hostname

- Example: `hostname` - Display the system's hostname.

16. **reboot** - Reboot the system

- Usage: `reboot`

- Options:

  - None

- Example: `reboot` - Reboot the system

17. **clear** - Clear the terminal screen

- Usage: `clear`

- Options:

- None

- Example: `clear` - Clear the terminal screen

CODE EXECUTION:

```
~/1c$ su
Password:
ssu: Authentication failure
~/1c$ sudo
sudo: command not installed, but was located via Nix.
Would you like to run sudo from Nix and add it to your replit.nix file? [Yn]: y
Adding sudo to replit.nix
success
/nix/store/1k145gm933pi0bv9l35v3rd65ac5j8zk-sudo-1.9.12p1
sudo: The "no new privileges" flag is set, which prevents sudo from running as root.
sudo: If sudo is running in a container, you may need to adjust the container configuration to disable the flag.
Detected change in environment, reloading shell...
~/1c$ man
This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, including manpages, you can run the 'unminimize'
command. You will still need to ensure the 'man-db' package is installed.
~/1c$ help
GNU bash, version 5.1.12(1)-release (x86_64-pc-linux-gnu)
These shell commands are defined internally.  Type `help' to see this list.
Type `help name' to find out more about the function `name'.
Use `info bash' to find out more about the shell in general.
Use `man -k' or `info' to find out more about commands not in this list.

A star (*) next to a name means that the command is disabled.

 job_spec [&]                                            history [-c] [-d offset] [n] or history -anrw [filename] or >
 (( expression ))                                        if COMMANDS; then COMMANDS; [ elif COMMANDS; then COMMANDS; >
 . filename [arguments]                                  jobs [-lnprs] [jobspec ...] or jobs -x command [args]
 :                                                       kill [-s sigspec | -n signum | -sigspec] pid | jobspec ... o>
 [ arg... ]                                              let arg [arg ...]
 [[ expression ]]                                        local [option] name[=value] ...
 alias [-p] [name[=value] ... ]                          logout [n]
 bg [job_spec ...]                                       mapfile [-d delim] [-n count] [-O origin] [-s count] [-t] [->
 bind [-lpsvPSVX] [-m keymap] [-f filename] [-q name] [-u name>  popd [-n] [+N | -N]
 break [n]                                               printf [-v var] format [arguments]
 builtin [shell-builtin [arg ...]]                       pushd [-n] [+N | -N | dir]
 caller [expr]                                           pwd [-LP]
 case WORD in [PATTERN [| PATTERN]...) COMMANDS ;;]... esac  read [-ers] [-a array] [-d delim] [-i text] [-n nchars] [-N >
 cd [-L|[-P [-e]] [-@]] [dir]                            readarray [-d delim] [-n count] [-O origin] [-s count] [-t] >
 command [-pVv] command [arg ...]                        readonly [-aAf] [name[=value] ...] or readonly -p
```

```
~/1c$ history
    1  su
    2  sudo
    3  man
    4  help
    5  history
~/1c$ who
~/1c$ whoami
runner
~/1c$ id
uid=1000(runner) gid=1000(runner) groups=1000(runner)
~/1c$ uname
Linux
~/1c$ uptime
 04:16:16  up   2:01,  0 users,  load average: 8.63, 9.35, 9.03
~/1c$ free
               total        used        free      shared  buff/cache   available
Mem:        65852256    20248292    23540784       23908    22063180    45109276
Swap:             0           0           0
~/1c$ tty
/dev/pts/3
~/1c$ cal
cal: command not installed. Multiple versions of this command were found in Nix.
Select one to run (or press Ctrl-C to cancel):
Adding util-linux.bin to replit.nix
```

```
success
/nix/store/r4wcqdk2ns6xv03m3zqw567fp7sjl02c-util-linux-2.38.1-bin
      May 2023
Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31

Detected change in environment, reloading shell...
~/1c$ date
Fri 12 May 2023 04:16:55 AM UTC
~/1c$ hostname
6017356ea296
~/1c$ reboot
```

Experiment – 8

FILES:

General

A simple description of the UNIX system, also applicable to Linux, is this:

"On a UNIX system, everything is a file; if something is not a file, it is a process."

This statement is true because there are special files that are more than just files (named pipes and sockets, for instance), but to keep things simple, saying that everything is a file is an acceptable generalization. A Linux system, just like UNIX, makes no difference between a file and a directory, since a directory is just a file containing names of other files. Programs, services, texts, images, and so forth, are all files. Input and output devices, and generally all devices, are considered to be files, according to the system.

In order to manage all those files in an orderly fashion, man likes to think of them in an ordered tree-like structure on the hard disk, as we know from MS-DOS (Disk Operating System) for instance. The large branches contain more branches, and the branches at the end contain the tree's leaves or normal files. For now we will use this image of the tree, but we will find out later why this is not a fully accurate image.

Sorts of files

Most files are just files, called *regular* files; they contain normal data, for example text files, executable files or programs, input for or output from a program and so on.

While it is reasonably safe to suppose that everything you encounter on a Linux system is a file, there are some exceptions.

- *Directories*: files that are lists of other files.
- *Special files*: the mechanism used for input and output. Most special files are in /dev, we will discuss them later.
- *Links*: a system to make a file or directory visible in multiple parts of the system's file tree. We will talk about links in detail.
- *(Domain) sockets*: a special file type, similar to TCP/IP sockets, providing inter-process networking protected by the file system's access control.
- *Named pipes*: act more or less like sockets and form a way for processes to communicate with each other, without using network socket semantics.

The -l option to **ls** displays the file type, using the first character of each input line:

```
jaime:~/Documents> ls -l
total 80
-rw-rw-r--   1 jaime   jaime   31744 Feb 21 17:56 intro Linux.doc
-rw-rw-r--   1 jaime   jaime   41472 Feb 21 17:56 Linux.doc
drwxrwxr-x   2 jaime   jaime    4096 Feb 25 11:50 course
```

This table gives an overview of the characters determining the file type:

Table on File types in a long list

Symbol	Meaning
-	Regular file
d	Directory
l	Link
c	Special file
s	Socket
p	Named pipe
b	Block device

In order not to always have to perform a long listing for seeing the file type, a lot of systems by default don't issue just **ls**, but **ls -F**, which suffixes file names with one of the characters "/=*|@" to indicate the file type. To make it extra easy on the beginning user, both the -F and --color options are usually combined. We will use **ls -F** throughout this document for better readability.

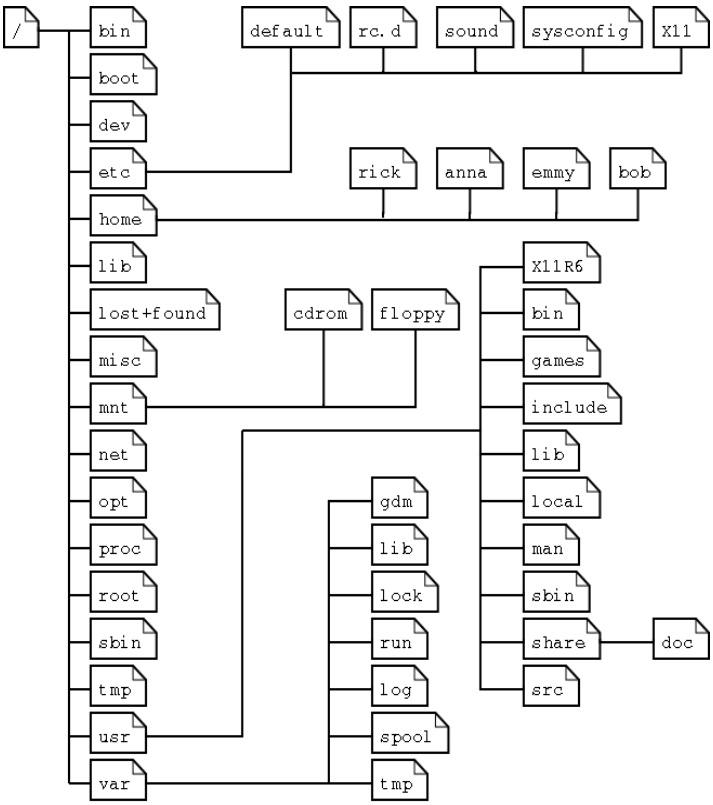
As a user, you only need to deal directly with plain files, executable files, directories and links. The special file types are there for making your system do what you demand from it and are dealt with by system administrators and programmers.

More file system layout

Visual

For convenience, the Linux file system is usually thought of in a tree structure. On a standard Linux system you will find the layout generally follows the scheme presented below.

Linux file system layout



This is a layout from a RedHat system. Depending on the system admin, the operating system and the mission of the UNIX machine, the structure may vary, and directories may be left out or added at will. The names are not even required; they are only a convention.

The tree of the file system starts at the trunk or *slash*, indicated by a forward slash (/). This directory, containing all underlying directories and files, is also called the *root directory* or "the root" of the file system.

Directories that are only one level below the root directory are often preceded by a slash, to indicate their position and prevent confusion with other directories that could have the same name. When starting with a new system, it is always a good idea to take a look in the root directory. Let's see what you could run into:

```
emmy:~> cd /
emmy:/> ls
bin/   dev/  home/    lib/        misc/  opt/    root/  tmp/  var/
boot/  etc/  initrd/  lost+found/  mnt/   proc/   sbin/  usr/
```

Subdirectories of the root directory

Directory	Content
/bin	Common programs, shared by the system, the system administrator and the users.
/boot	The startup files and the kernel, vmlinuz. In some recent distributions also grub data. Grub is the GRand Unified Boot loader and is an attempt to get rid of the many different boot-loaders we know today.
/dev	Contains references to all the CPU peripheral hardware, which are represented as files with special properties.
/etc	Most important system configuration files are in /etc, this directory contains data similar to those in the Control Panel in Windows
/home	Home directories of the common users.
/initrd	(on some distributions) Information for booting. Do not remove!
/lib	Library files, includes files for all kinds of programs needed by the system and the users.
/lost+found	Every partition has a lost+found in its upper directory. Files that were saved during failures are here.
/misc	For miscellaneous purposes.
/mnt	Standard mount point for external file systems, e.g. a CD-ROM or a digital camera.
/net	Standard mount point for entire remote file systems
/opt	Typically contains extra and third party software.
/proc	A virtual file system containing information about system resources. More information about the meaning of the files in proc is obtained by entering the command **man *proc*** in a terminal window. The file proc.txt discusses the virtual file system in detail.
/root	The administrative user's home directory. Mind the difference between /, the root directory and /root, the home directory of the *root* user.
/sbin	Programs for use by the system and the system administrator.
/tmp	Temporary space for use by the system, cleaned upon reboot, so don't use this for saving any work!
/usr	Programs, libraries, documentation etc. for all user-related programs.
/var	Storage for all variable files and temporary files created by users, such as log files, the mail queue, the print spooler area, space for temporary storage of files downloaded from the Internet, or to keep an image of a CD before burning it.

Using the **df** command with a dot (.) as an option shows the partition the current directory belongs to, and informs about the amount of space used on this partition:

```
sandra:/lib> df -h .
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda7             980M  163M  767M  18% /
```

As a general rule, every directory under the root directory is on the root partition, unless it has a separate entry in the full listing from **df** (or **df -h** with no other options).

The file system in reality

For most users and for most common system administration tasks, it is enough to accept that files and directories are ordered in a tree-like structure. The computer, however, doesn't understand a thing about trees or tree-structures.

Every partition has its own file system. By imagining all those file systems together, we can form an idea of the tree-structure of the entire system, but it is not as simple as that. In a file system, a file is represented by an *inode*, a kind of serial number containing information about the actual data that makes up the file: to whom this file belongs, and where is it located on the hard disk.

Every partition has its own set of inodes; throughout a system with multiple partitions, files with the same inode number can exist.

Each inode describes a data structure on the hard disk, storing the properties of a file, including the physical location of the file data. When a hard disk is initialized to accept data storage, usually during the initial system installation process or when adding extra disks to an existing system, a fixed number of inodes per partition is created. This number will be the maximum amount of files, of all types (including directories, special files, links etc.) that can exist at the same time on the partition. We typically count on having 1 inode per 2 to 8 kilobytes of storage.

At the time a new file is created, it gets a free inode. In that inode is the following information:

- Owner and group owner of the file.
- File type (regular, directory, ...)
- Permissions on the file
- Date and time of creation, last read and change.
- Date and time this information has been changed in the inode.
- Number of links to this file (see later in this chapter).
- File size
- An address defining the actual location of the file data.

The only information not included in an inode, is the file name and directory. These are stored in the special directory files. By comparing file names and inode numbers, the system can make up a tree-structure that the user understands. Users can display inode numbers using the -i option to ls. The inodes have their own separate space on the disk.

Experiment – 9

Aim: Implement the commands that is used for Creating and Manipulating files: cat, cp, mv, rm, ls and its options, touch and their options, which, whereis, whatis.

1. **cat** - Concatenate files and print on the standard output
   - Usage: `cat [OPTIONS] [FILE]...`
   - Options:
     - `-n` - Number all output lines
     - `-b` - Number non-blank output lines
     - `-E` - Display a $ at the end of each line
     - `-T` - Display TAB characters as ^I
   - Example: `cat file1.txt file2.txt` - Display the contents of file1.txt and file2.txt on the console
2. **cp** - Copy files and directories
   - Usage: `cp [OPTIONS] SOURCE DEST`
   - Options:
     - `-r` - Copy directories recursively
     - `-v` - Verbose output, display the names of the files as they are copied
     - `-i` - Prompt before overwriting files
   - Example: `cp file1.txt backup/file1.txt` - Copy file1.txt to the backup directory
3. **mv** - Move or rename files and directories
   - Usage: `mv [OPTIONS] SOURCE DEST`
   - Options:
     - `-i` - Prompt before overwriting files
     - `-v` - Verbose output, display the names of the files as they are moved
   - Example: `mv file1.txt backup/file1.txt` - Move file1.txt to the backup directory
4. **rm** - Remove files and directories
   - Usage: `rm [OPTIONS] FILE...`
   - Options:
     - `-r` - Remove directories and their contents recursively

- `-f` - Force the removal of files without prompting

- Example: `rm file1.txt` - Remove file1.txt

5. **ls** - List directory contents

- Usage: `ls [OPTIONS] [FILE]...`

- Options:

- `-a` - Include hidden files in the listing

- `-l` - Long format listing, display detailed information about each file

- `-h` - Display file sizes in a human-readable format

- Example: `ls -l` - List all files in the current directory with long format listing

6. **touch** - Create an empty file or update the modification timestamp of an existing file

- Usage: `touch [OPTIONS] FILE...`

- Options:

- `-a` - Change the access time only

- `-m` - Change the modification time only

- Example: `touch file1.txt` - Create an empty file1.txt or update its modification time.

7. **which** - Display the location of an executable file in the system path

- Usage: `which [OPTIONS] COMMAND...`

- Options:

- `-a` - Display all locations of the command in the system path

- Example: `which python` - Display the location of the python executable file in the system path

8. **whereis** - Locate the binary, source, and manual page files for a command

- Usage: `whereis [OPTIONS] COMMAND...`

- Options:

- None

- Example: `whereis python` - Display the locations of the binary, source, and manual page files for the python command.

9. **whatis** - Display a brief description of a command

- Usage: `whatis [OPTIONS] COMMAND...`

- Options:

- None

- Example: `whatis python` - Display a brief description of the python command.

COMMANDS:

```
~$ nano new.c
~$
~$ gcc new.c
~$ ./a.out
Hello World!~$
~$ cat new.c
#include <stdio.h>
int main()
{
        printf("Hello World!");
        return 0;
}
~$ mkdir dir1 dir2 dir3 new1.c
~$ ls
2023-05-21-file-1.term  a.out  dir1  dir2  dir3  new.c  new1.c
~$ rmdir dir3
~$ ls
2023-05-21-file-1.term  a.out  dir1  dir2  new.c  new1.c
```

```
~$ cp -v new.c new1.c
'new.c' -> 'new1.c/new.c'
~$ mv -v dir1 new.c
mv: cannot overwrite non-directory 'new.c' with directory 'dir1'
~$ mv -v new.c dir1
renamed 'new.c' -> 'dir1/new.c'
~$ ls
2023-05-21-file-1.term  a.out  dir1  dir2  new1.c
```

Experiment – 10

Aim: Implement File system commands: Comparing Files using diff, cmp, comm.

1. **diff** - Compare files line by line
   - Usage: `diff [OPTIONS] FILE1 FILE2`
   - Options:
     - `-i` - Ignore case when comparing lines
     - `-b` - Ignore changes in the amount of whitespace
     - `-w` - Ignore all whitespace
     - `-u` - Display unified context format
   - Example: `diff file1.txt file2.txt` - Compare the contents of file1.txt and file2.txt line by line
2. **cmp** - Compare two files byte by byte
   - Usage: `cmp [OPTIONS] FILE1 FILE2`
   - Options:
     - `-i` - Ignore case when comparing bytes
     - `-l` - Display differences by byte position and value
     - `-s` - Silent mode, do not display any output
   - Example: `cmp file1.txt file2.txt` - Compare the contents of file1.txt and file2.txt byte by byte
3. **comm** - Compare two sorted files line by line
   - Usage: `comm [OPTIONS] FILE1 FILE2`
   - Options:
     - `-1` - Suppress lines unique to FILE1
     - `-2` - Suppress lines unique to FILE2
     - `-3` - Suppress lines common to both files
   - Example: `comm file1.txt file2.txt` - Compare the contents of file1.txt and file2.txt line by line and display the lines that are unique to each file and the lines that are common to both files

CODE EXECUTION:

1. diff

```
m2m-sd@m2m-sds-MacBook-Pro os % echo "This is file 1." > file1.txt
m2m-sd@m2m-sds-MacBook-Pro os % echo "This is file 2." > file2.txt
m2m-sd@m2m-sds-MacBook-Pro os % diff file1.txt file2.txt
1c1
< This is file 1.
---
> This is file 2.
m2m-sd@m2m-sds-MacBook-Pro os %
```

2. cmp

```
m2m-sd@m2m-sds-MacBook-Pro os % cmp file1.txt file2.txt
file1.txt file2.txt differ: char 14, line 1
m2m-sd@m2m-sds-MacBook-Pro os %
```

3. comm

```
m2m-sd@m2m-sds-MacBook-Pro os % comm file1.txt file2.txt
This is file 1.
        This is file 2.
m2m-sd@m2m-sds-MacBook-Pro os %
```

Experiment – 11

Aim: Implement deadlock in C by using shared variable.

Algorithm:

1. Declare and initialize two mutex locks: lock1 and lock2.
2. Declare a global shared variable, initialized to zero.
3. Define a function thread1, which takes no arguments and returns void*.
4. In thread1:
    a. Acquire lock1 using pthread_mutex_lock().
    b. Print "Thread 1 acquired lock 1".
    c. Sleep for 1 second using sleep(1).
    d. Acquire lock2 using pthread_mutex_lock().
    e. Print "Thread 1 acquired lock 2".
    f. Increment the shared variable.
    g. Release lock2 using pthread_mutex_unlock().
    h. Release lock1 using pthread_mutex_unlock().
5. Define a function thread2, which takes no arguments and returns void*.
6. In thread2:
    a. Acquire lock2 using pthread_mutex_lock().
    b. Print "Thread 2 acquired lock 2".
    c. Sleep for 1 second using sleep(1).
    d. Acquire lock1 using pthread_mutex_lock().
    e. Print "Thread 2 acquired lock 1".
    f. Decrement the shared variable.
    g. Release lock1 using pthread_mutex_unlock().
    h. Release lock2 using pthread_mutex_unlock().
7. In the main function:
    a. Initialize lock1 and lock2 using pthread_mutex_init().
    b. Create two threads using pthread_create(), passing thread1 and thread2 as the function arguments.

c. Wait for both threads to finish using pthread_join().

d. Destroy lock1 and lock2 using pthread_mutex_destroy().

e. Return 0.

Code:

```
#include <stdio.h>
#include <pthread.h>
int shared_variable = 0;
void *thread_1(void *arg) {
  pthread_mutex_lock(&shared_variable);
  printf("Thread 1 acquired shared variable\n");
  sleep(1);
  pthread_mutex_unlock(&shared_variable);
  return NULL;
}
void *thread_2(void *arg) {
  pthread_mutex_lock(&shared_variable);
  printf("Thread 2 acquired shared variable\n");
  sleep(1);
  pthread_mutex_unlock(&shared_variable);
  return NULL;
}
int main() {
  pthread_t thread_1_id, thread_2_id;
  pthread_mutex_init(&shared_variable, NULL);
  pthread_create(&thread_1_id, NULL, thread_1, NULL);
  pthread_create(&thread_2_id, NULL, thread_2, NULL);
  pthread_join(thread_1_id, NULL);
  pthread_join(thread_2_id, NULL);
  printf("Threads joined\n");
  return 0;
```

}

Output:

```
~$ ./a.out
Thread 1 acquired shared variable
Thread 2 acquired shared variable
Threads joined
~$
```

Experiment – 12

Aim: Implement Directory oriented commands: ls, cd, pwd, mkdir, rmdir.

1. pwd Command

The pwd command is used to display the location of the current working directory.

Syntax:

1. pwd

2. mkdir Command

The mkdir command is used to create a new directory under any directory.

Syntax:

1. mkdir <directory name>

3. rmdir Command

The rmdir command is used to delete a directory.

Syntax:

1. rmdir <directory name>

4. ls Command

The ls command is used to display a list of content of a directory.

Syntax:

1. ls

5. cd Command

The cd command is used to change the current directory.

Syntax:

1. cd <directory name>

OUTPUT:

```
~$ pwd
/home/user
~$ mkdir dir1 dir2 dir3
~$ ls
2023-05-22-file-1.term  dir1  dir2  dir3
~$ rmdir dir3
~$ ls
2023-05-22-file-1.term  dir1  dir2
~$ cd
~$ cd dir1
~/dir1$
```