Practical File
Of
Operating System
22CS005

Submitted

in partial fulfillment for the award of the degree

of

BACHELEOR OF ENGINEERING

in

COMPUTER SCIENCE & ENGINEERING

CHITKARA UNIVERSITY

CHANDIGARH-PATIALA NATIONAL HIGHWAY
RAJPURA (PATIALA) PUNJAB-140401 (INDIA)

June, 2024

Submitted To:

Dr. Ravneet Kaur
Assistant Professor
Chitkara University, Punjab

Submitted By:

Komal
2310992042
2^{nd} Sem. - Batch 2023

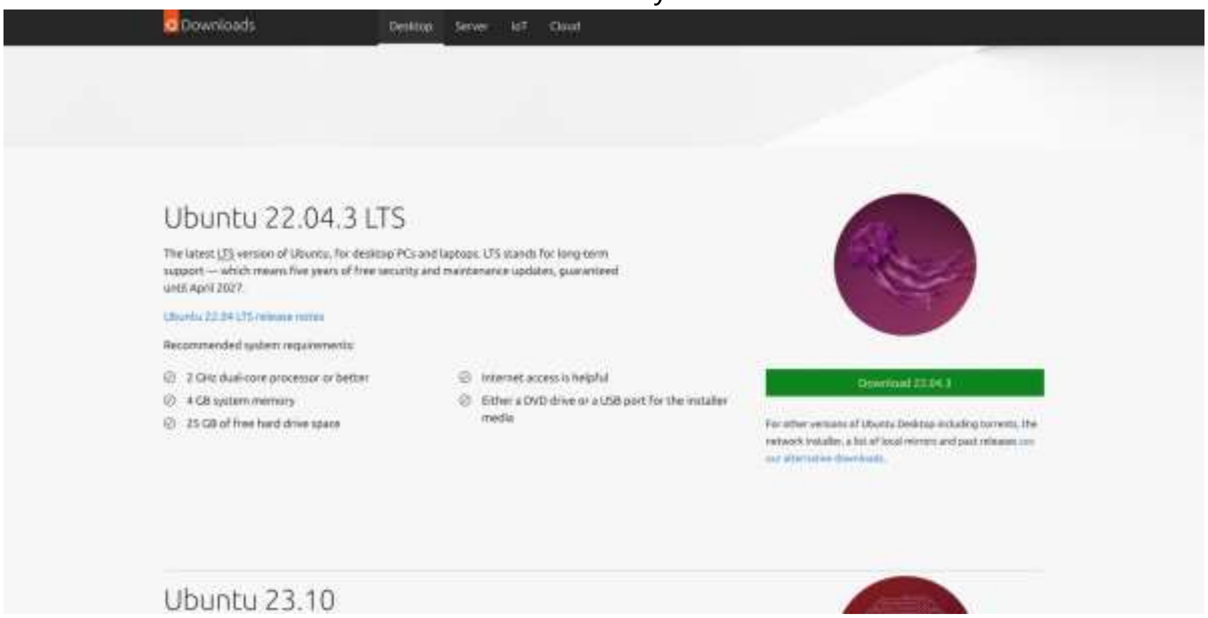
INDEX

Sr. No.	Practical Name	Teacher Sign
1	a) Installation: Configuration & Customizations of Linux b) Introduction to GCC compiler: Basics of GCC, Compilation of program, Execution of program. c) Time stamping in Linux. d) Automating the execution using Make file.	
2	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	
3	Implement the commands that is used for Creating and Manipulating files: cat, cp, mv, rm, ls and its options, touch and their options, which is, where is, what is	
4	Implement Directory oriented commands: cd, pwd, mkdir, rmdir, Comparing Files using diff, cmp, comm.	
5	Write a program and execute the same to demonstrate how to use terminal commands in C program (using system() function)	
6	Write a program to implement process concepts using C language by printing process Id.	
7	Write a program to create and execute process using fork() and exec() system calls.	
8	Write a C program to implement FCFS (First Come First Serve) and SJF (Shortest Job First) scheduling algorithms.	
9	Write a C program to implement Priority Scheduling and RR (RoundRobin) scheduling algorithms.	

Program 1: a) Installation: Configuration & Customizations of Linux

STEP 1-Download Virtual box First, download and install VirtualBox from the official website. Choose the version appropriate for your operating system.

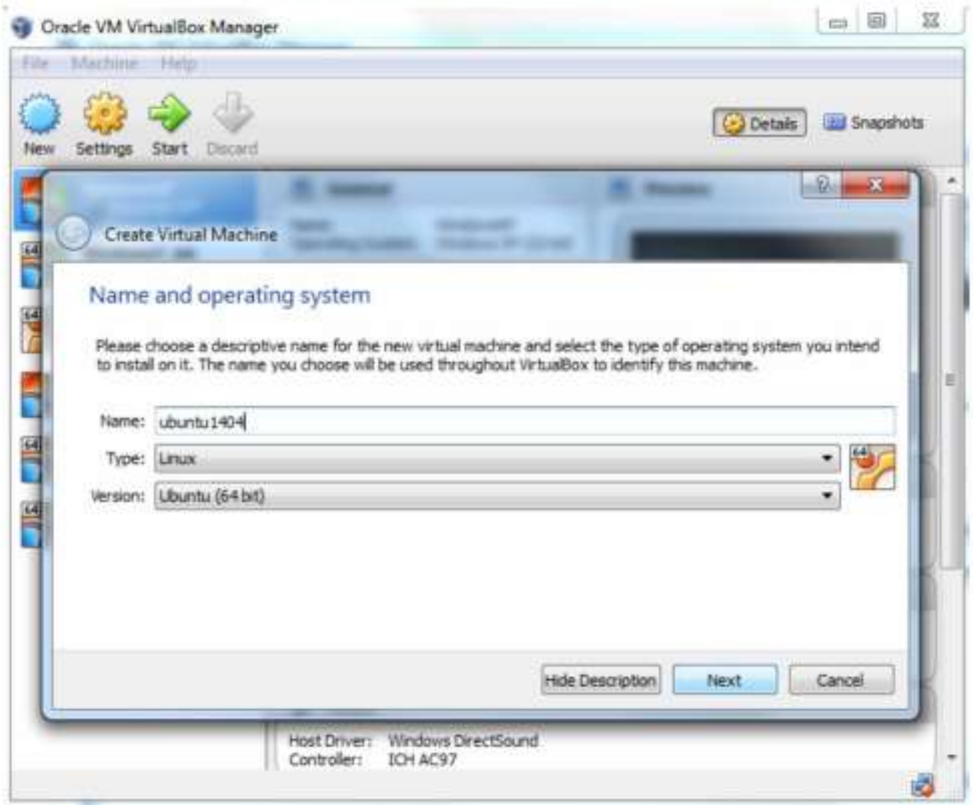
STEP 2- Download Ubuntu ISO Go to the official Ubuntu website and download the ISO file for the version of Ubuntu you want to install.



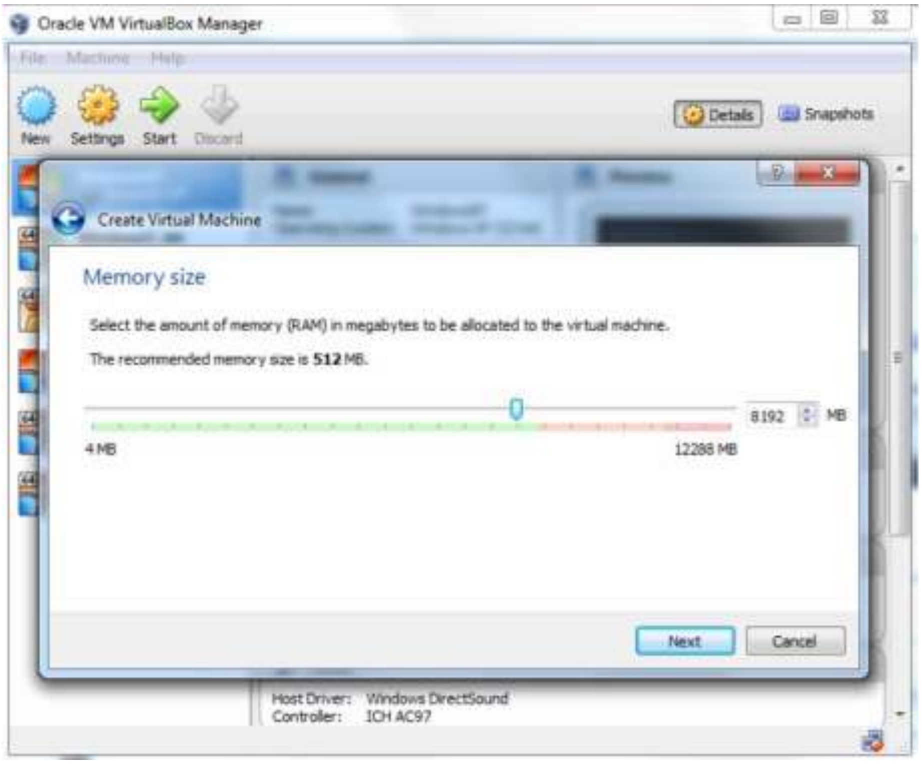
-Create a New Virtual Machine:

- Open VirtualBox and click on the "New" button.
- Name your virtual machine (e.g., Ubuntu) and select the type as "Linux" and version as "Ubuntu (64-bit)" (assuming you're installing a 64-bit version).

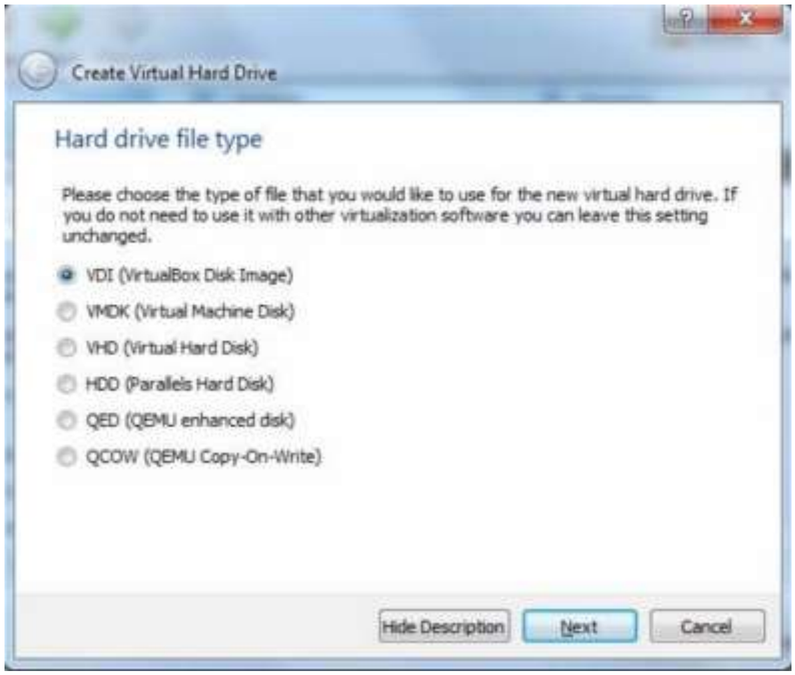
STEP 3



- Allocate memory (RAM) to the virtual machine. It's recommended to allocate at least 2GB for smooth operation.

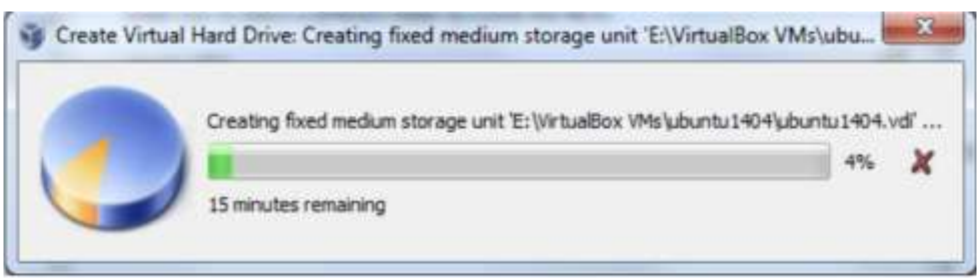


- Create a virtual hard disk now. Choose the recommended size or adjust as needed. Select "VDI (VirtualBox Disk Image)" as the hard disk file type.



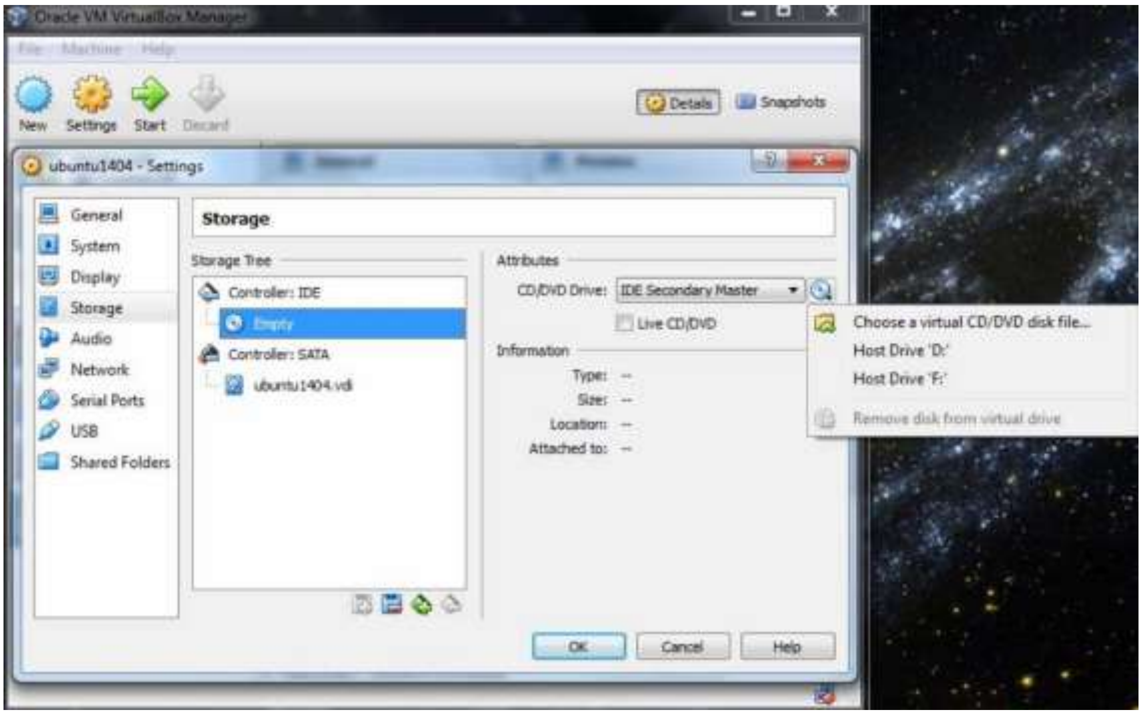
- Choose "Dynamically allocated" for the storage on physical hard disk, unless you have specific needs for fixed size allocation.

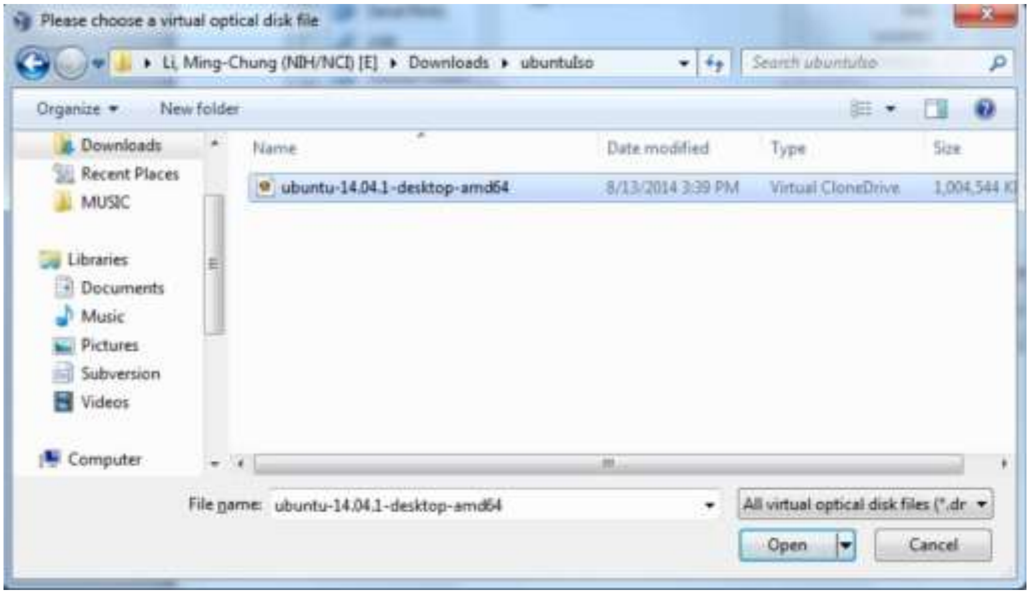
- Click 'Create' button and VirtualBox will generate Ubuntu virtual machine.

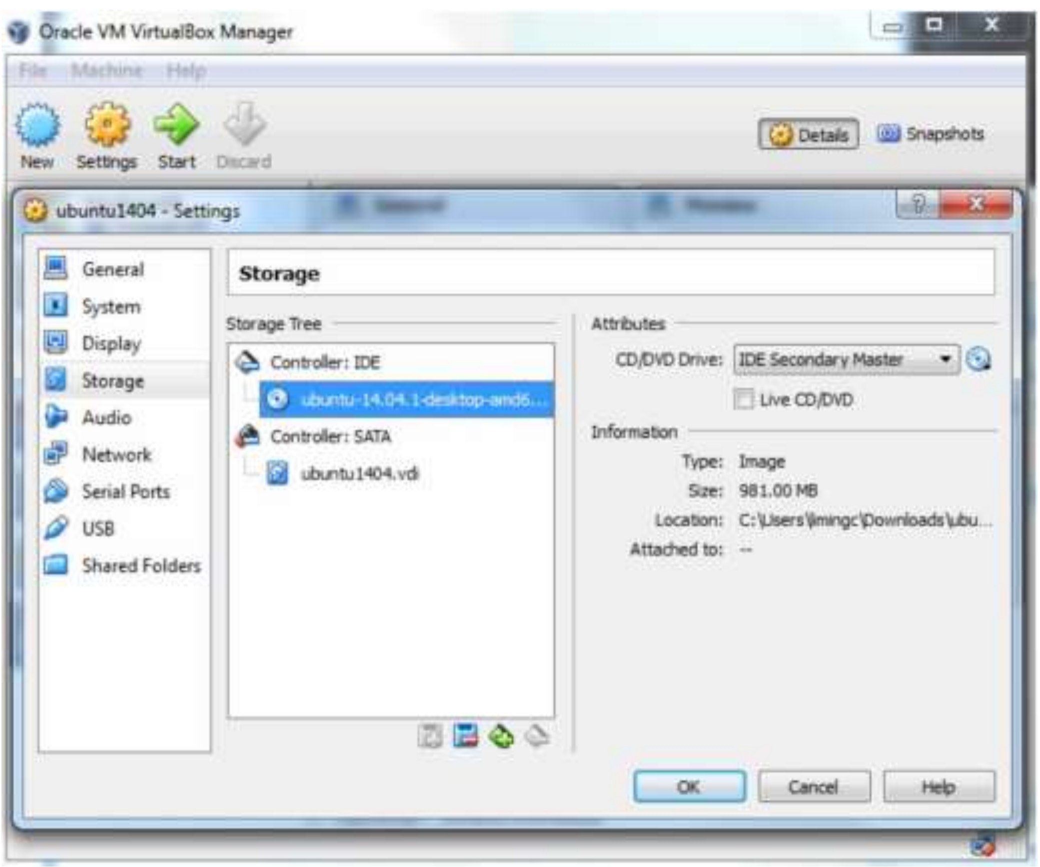


STEP 4- Select your new virtual machine and click 'Settings' button. Click on 'Storage' category and then 'Empty' under Controller: IDE. Click "CD/DVD" icon on right hand side and select the ubuntu ISO file to mount.

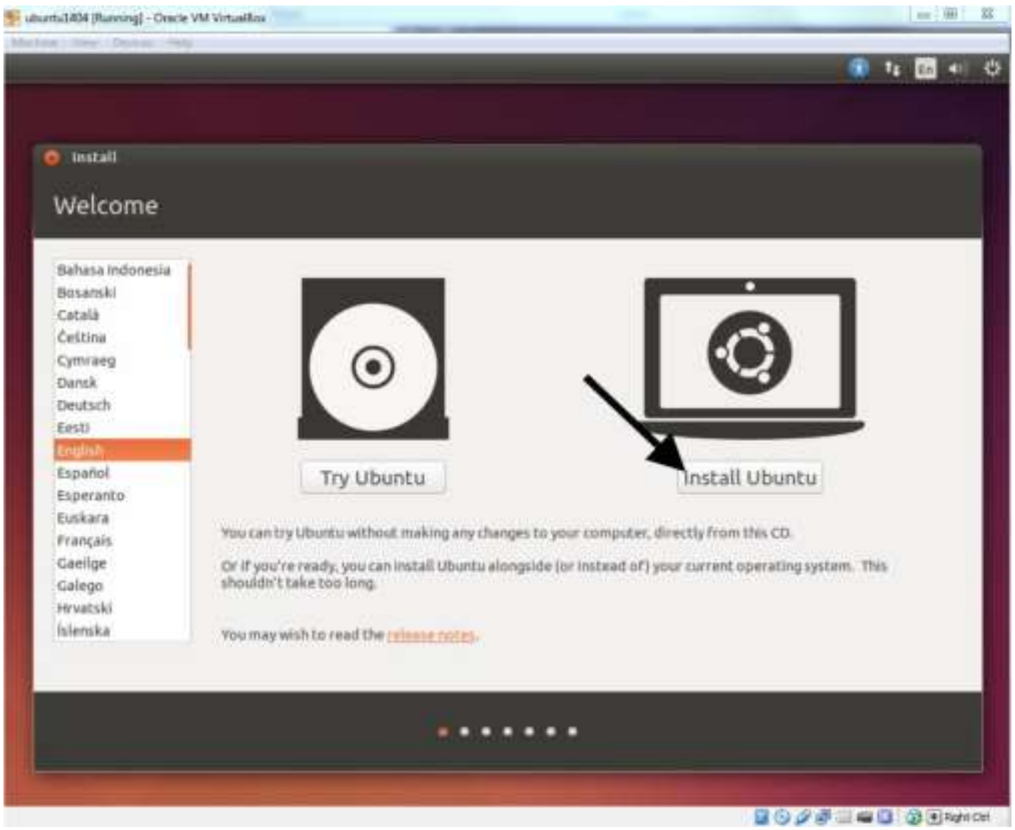
Note that if you have not downloaded 64-bit Ubuntu ISO file, you can check out this page for more information. When downloading Ubuntu ISO file, make sure to select 64-bit version. Also make sure the VT-x/Virtualization Technology has been enabled in your computer's BIOS/Basic Input Output System.

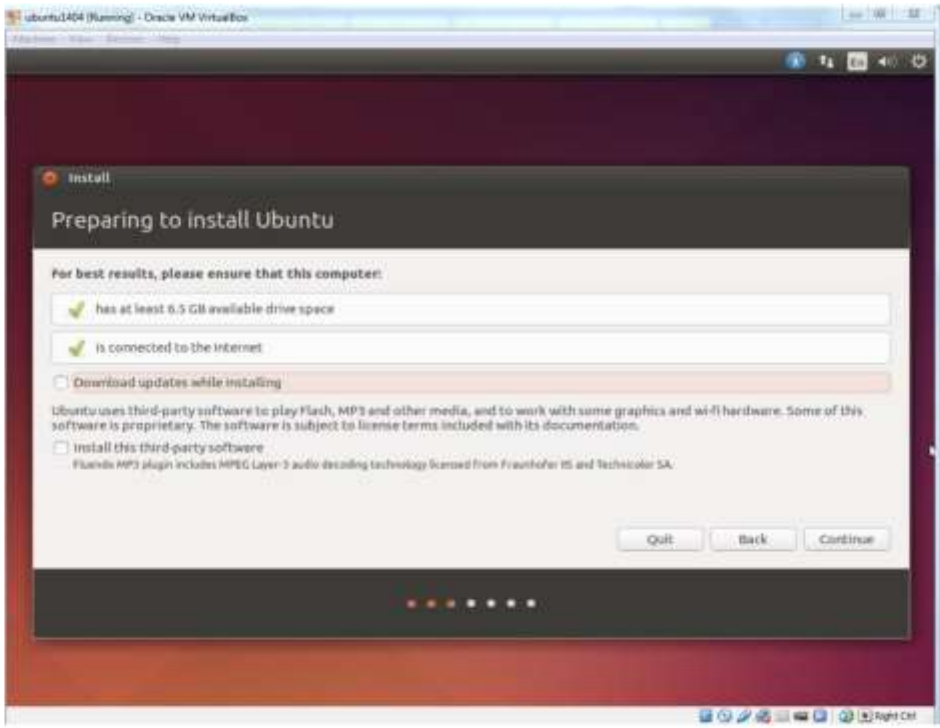



Please choose a virtual optical disk file
Li, Ming-Chung (NIH/NCI) [E] ▸ Downloads ▸ ubuntuIso
Search ubuntuIso
Organize
New folder
Downloads
Recent Places
MUSIC
Libraries
Documents
Music
Pictures
Subversion
Videos
Computer
Name
Date modified
Type
Size
ubuntu-14.04.1-desktop-amd64
8/13/2014 3:39 PM
Virtual CloneDrive
File name:
ubuntu-14.04.1-desktop-amd64
All virtual optical disk files (*.dr
Open
Cancel

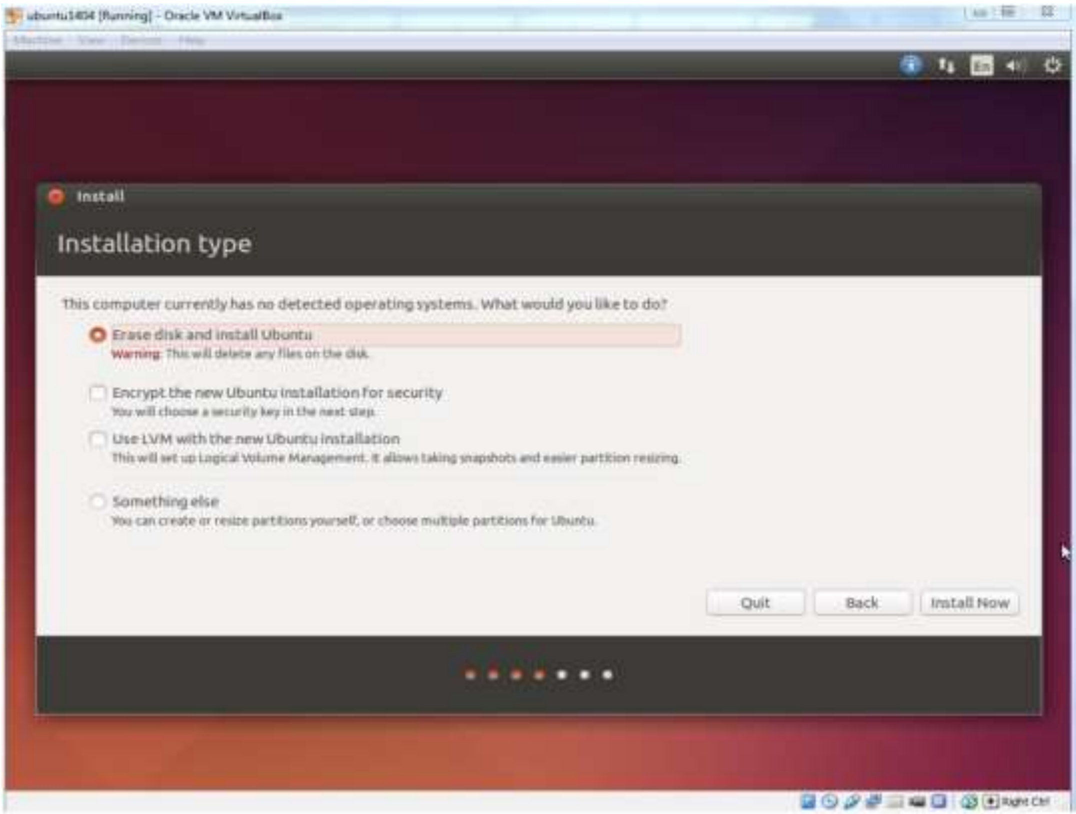

Oracle VM VirtualBox Manager
File
Machine
Help
New
Settings
Start
Discard
Details
Snapshots
ubuntu1404 - Settings
General
System
Display
Storage
Audio
Network
Serial Ports
USB
Shared Folders
Storage
Storage Tree
Controller: IDE
ubuntu-14.04.1-desktop-amd6...
Controller: SATA
ubuntu1404.vdi
Attributes
CD/DVD Drive:
IDE Secondary Master
Live CD/DVD
Information
Type:
Image
Size:
981.00 MB
Location:
C:\Users\limingc\Downloads\ubu...
Attached to:
--
OK
Cancel
Help

STEP 3- Install Ubuntu

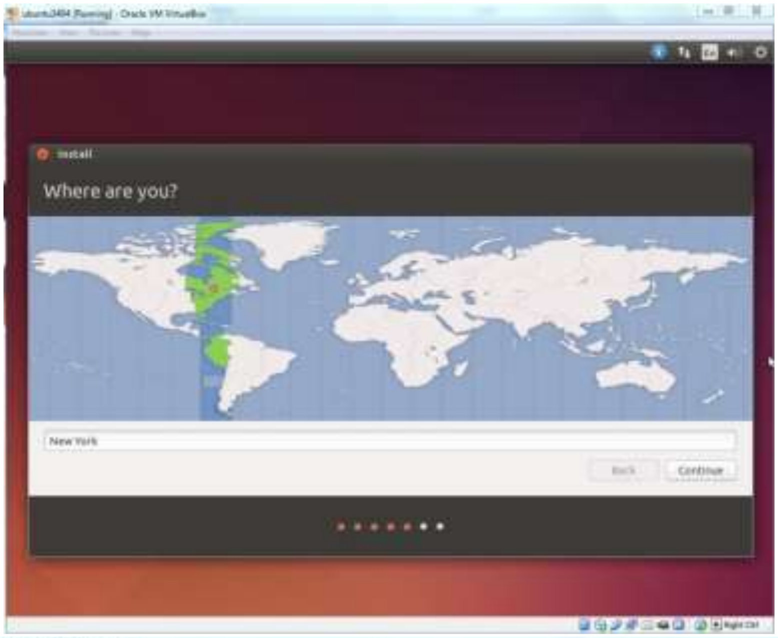


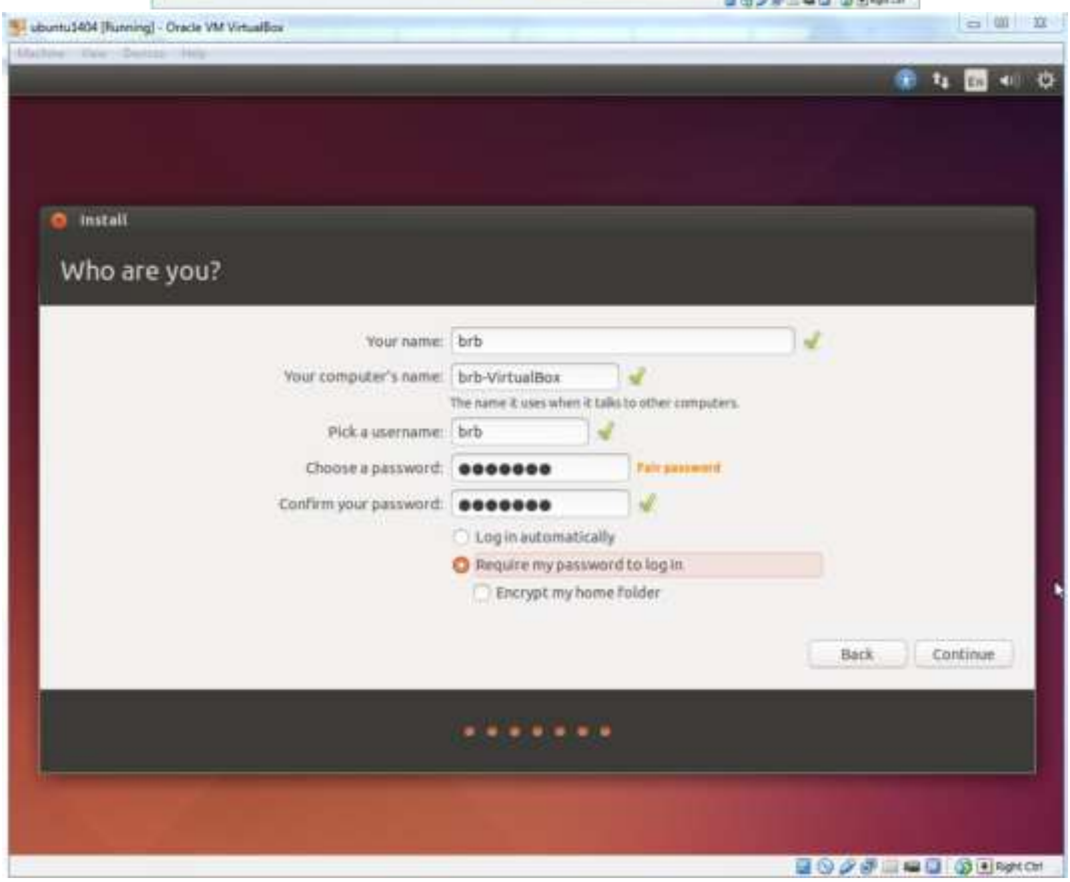


Make sure 'Erase disk and install Ubuntu' option is selected and click 'Install Now' button.



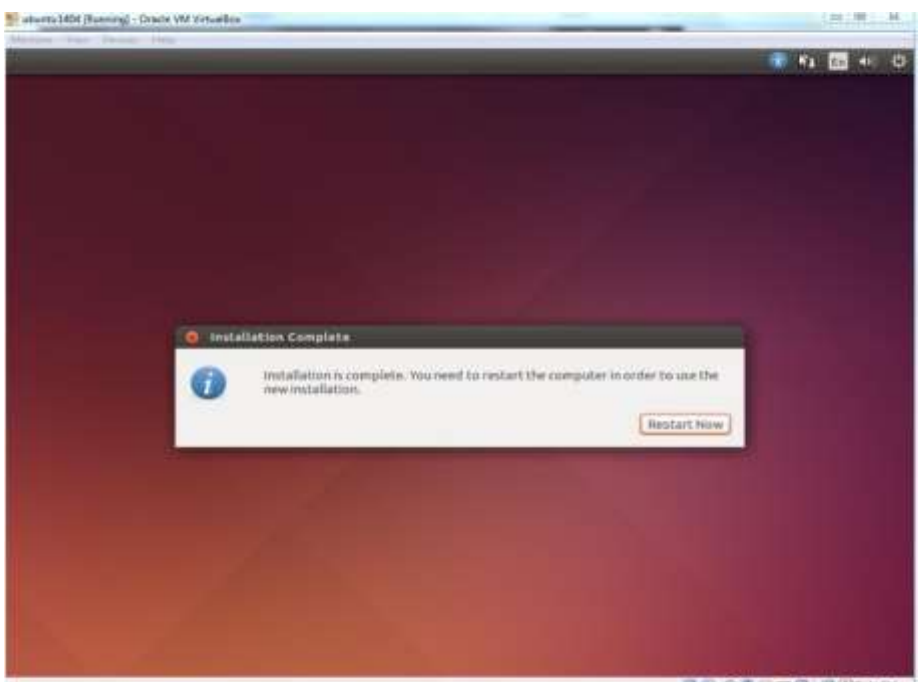
Ubuntu will ask you a few questions. If the default is good, click 'Continue' button.



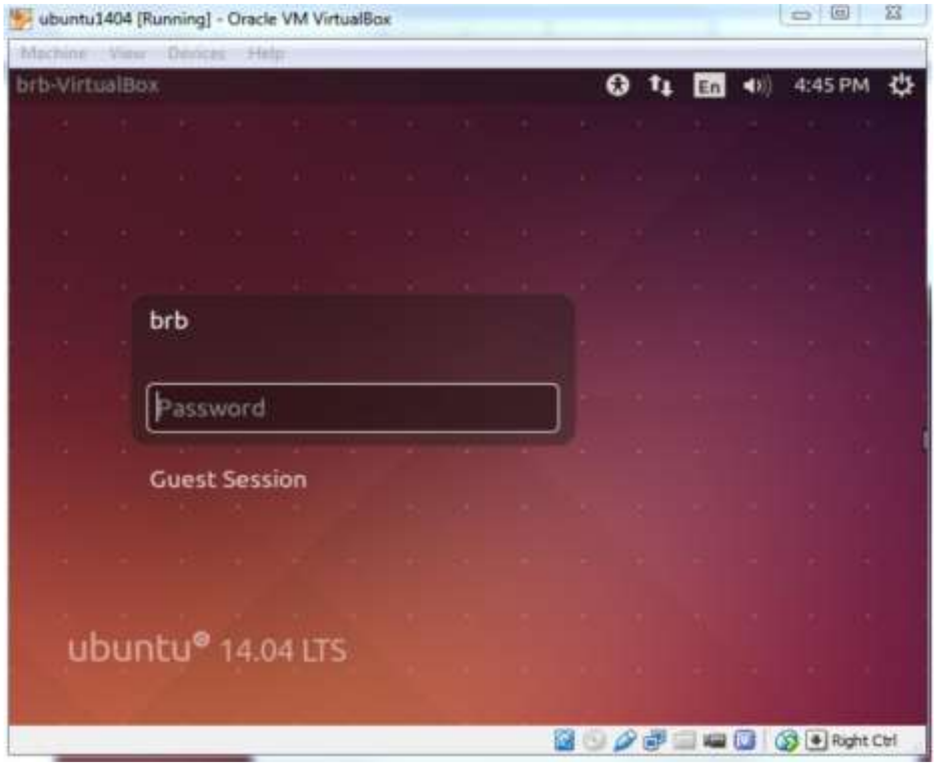


The installation will continue until it is finished.

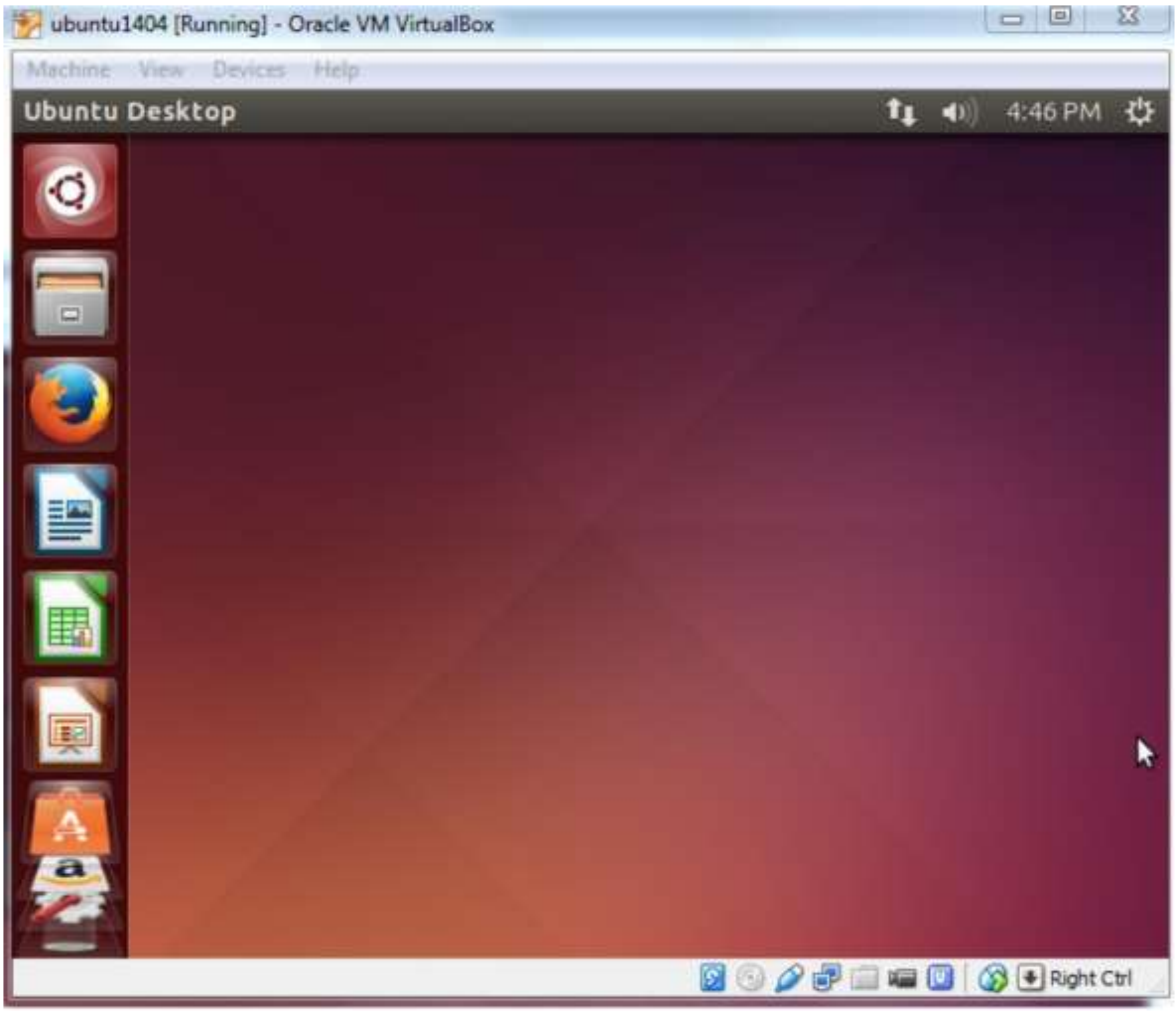
After installation is complete, click 'Restart Now' button. When you see a screen with a black background saying 'Please remove installation media and close the tray (if any) then press ENTER:', just follow it.



Enter the password you have chosen and press 'Enter'.



The Ubuntu Desktop OS is ready. You may find the desktop screen is too small. Don't worry. You can solve this easily with "VirtualBox Guest Additions".


ubuntu1404 [Running] - Oracle VM VirtualBox
Machine View Devices Help
Ubuntu Desktop
4:46 PM
Right Ctrl

Program 1: b) Introduction to GCC compiler: Basics of GCC, Compilation of program, Execution of program

GCC (GNU Compiler Collection) is a widely used open-source compiler system developed by the GNU Project. It supports various programming languages, including C, C++, Objective-C, Fortran, Ada, and Go. GCC is renowned for its robustness, portability, and compliance with various language standards.

- ## A brief history of GCC

The original author of the GNU C Compiler (GCC) is Richard Stallman, the founder of the GNU Project. The GNU project was started in 1984 to create a complete Unix-like operating system as free software, in order to promote freedom and cooperation among computer users and programmers. Every Unix-like operating system needs a C compiler, and as there were no free compilers in existence at that time, the GNU Project had to develop one from scratch. The work was funded by donations from individuals and companies to the Free Software Foundation, a non-profit organization set up to support the work of the GNU Project. The first release of GCC was made in 1987. This was a significant breakthrough, being the first portable ANSI C optimizing compiler released as free software. Since that time, GCC has become one of the most important tools in the development of free software. A major revision of the compiler came with the 2.0 series in 1992, which added the ability to compile C++. In 1997, an experimental branch of the compiler (EGCS) was created to improve optimization and C++ support. Following this work, EGCS was adopted as the new mainline of GCC development, and these features became widely available in the 3.0 release of GCC in 2001. Over time, GCC has been extended to support many additional languages, including Fortran, ADA, Java, and Objective-C. The acronym GCC is now used to refer to the "GNU Compiler Collection". Its development is guided by the GCC Steering Committee, a group composed of representatives from GCC user communities in industry, research, and academia.

- ## Basics of GCC

Installation: GCC is usually pre-installed on Unix-like operating systems such as Linux. However, if it's not installed, you can easily install it using your package manager. For example, on Ubuntu, you can install GCC by running:

```
└─$ sudo apt install gcc
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Suggested packages:
  gcc-multilib autoconf automake libtool flex bison gdb gcc-doc
The following NEW packages will be installed:
  gcc
0 upgraded, 1 newly installed, 0 to remove and 1391 not upgraded.
Need to get 0 B/5232 B of archives.
After this operation, 48.1 kB of additional disk space will be used.
Selecting previously unselected package gcc.
(Reading database ... 391909 files and directories currently installed.)
Preparing to unpack .../gcc_4%3a13.2.0-2_arm64.deb ...
Unpacking gcc (4:13.2.0-2) ...
Setting up gcc (4:13.2.0-2) ...
Processing triggers for man-db (2.11.2-3) ...
Processing triggers for kali-menu (2023.4.3) ...
```

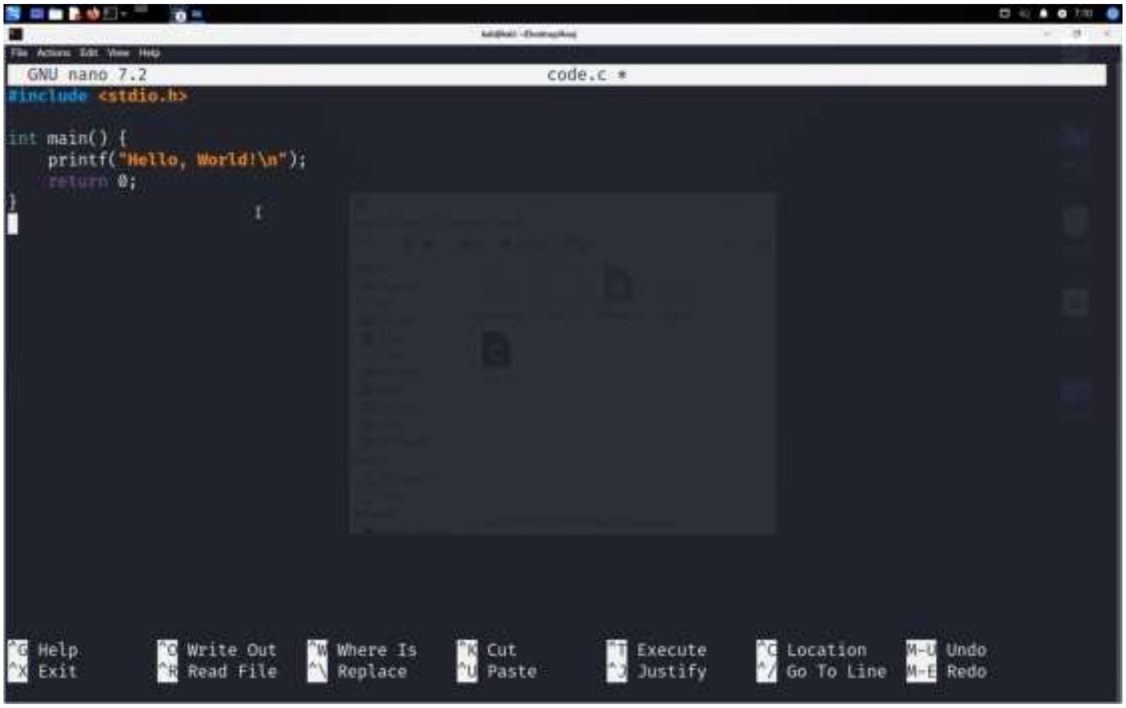
- ## Compilation of program

1. Create a source code file**:**

```
└─$ touch code.c
```

2. Edit the Source Code file**:**

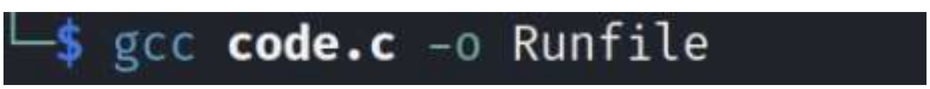
```
└─$ nano code.c
```



Save changes in Nano:

- After writing your C code, press Ctrl + O to save the changes.
- Press Enter to confirm the filename.
- Press Ctrl + X to exit Nano.

3. Compile the Program using GCC:



- **Execute the Compiled Program:**

This command executes the compiled program

Program 1: c) Time Stamping in Linux

In Linux, you can use the **date** command to display or set the system date and time. To add a timestamp to your C program, you can use the **time()** function from the **time.h** header file.

Source Code:

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

int main() {
   // Get the current time
time_t currentTime;     struct
tm *localTime;
time(&currentTime);
   localTime = localtime(&currentTime);

   // Print the timestamp
   printf("Timestamp: %s", asctime(localTime));

   return 0;
}
```

Explaination:

- **time(&currentTime)**: This function gets the current system time (in seconds since the Epoch) and stores it in the **currentTime** variable.
- **localtime(&currentTime)**: This function converts the time in seconds to a structure representing a local time, and stores it in the **localTime** variable.
- **asctime(localTime)**: This function converts the **localTime** structure to a human-readable string representation of the time, and returns a pointer to this string. This string includes the date, time, and timezone information.
- **printf("Timestamp: %s", ...)**: This line prints the timestamp obtained from **asctime()** to the console.

Output:

```
└─$ gcc code.c -o Runfile
└─$ ./Runfile
Timestamp: Sat Feb 10 07:46:59 2024
```

Program 1: d) Automating the Execution using Make File

To automate the execution of your C program using a Makefile, you can define rules that specify how to compile and run your program.

Source Code:

```
# Define compiler
CC = gcc

# Define compiler flags
CFLAGS = -Wall -Wextra

# Define target executable
TARGET = Runfile

# Define source files
SRCS = code.c

# Define objects
OBJS = $(SRCS:.c=.o)

# Default rule to build the executable
all: $(TARGET)

# Rule to build the executable
$(TARGET): $(OBJS)
   $(CC) $(CFLAGS) -o $@ $^

# Rule to compile source files
%.o: %.c
   $(CC) $(CFLAGS) -c $< -o $@

# Rule to clean up generated files clean:
   rm -f $(TARGET) $(OBJS)
```

To use this Makefile:

1. Save it as **Makefile** (case-sensitive) in the same directory as your source code (**code.c**).
2. Run **make** command in the terminal to build the executable.
3. Run **./Runfile** to execute the program.

Output:

```
└─$ make
gcc -Wall -Wextra -c code.c -o code.o
gcc -Wall -Wextra -o myprogram code.o
└─$ ./myprogram
This code is printed using make command
```

Program 2: 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

Output:

1. ***su :***

   The **su** command stands for "switch user" or "substitute user".

```
└─$ su kali
Password:
```

2. ***sudo :***

   Execute a single command with elevated privileges**:**

```
└─$ sudo apt install nano
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
nano is already the newest version (7.2-1).
0 upgraded, 0 newly installed, 0 to remove and 1387 not upgraded.
```

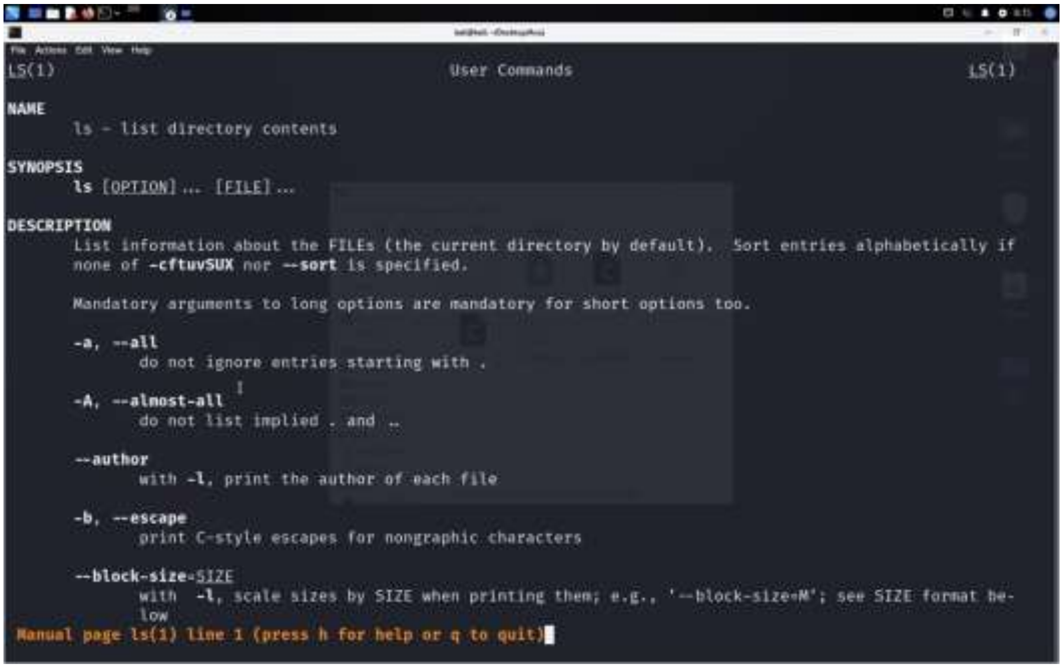
   Open a root shell:

```
└─$ sudo -i
┌──(root㉿kali)-[~]
└─#
```

3. ***man :***

   The **man** command is used in Unix-like operating systems to display the manual pages for a given command.

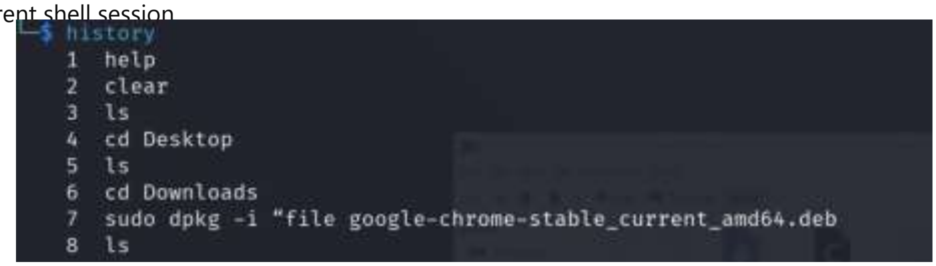
```
└─$ man ls
```



```
LS(1)                          User Commands                          LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List information about the FILEs (the current directory by default).  Sort entries alphabetically if
       none of -cftuvSUX nor --sort is specified.

       Mandatory arguments to long options are mandatory for short options too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..

       --author
              with -l, print the author of each file

       -b, --escape
              print C-style escapes for nongraphic characters

       --block-size=SIZE
              with  -l, scale sizes by SIZE when printing them; e.g., '--block-size=M'; see SIZE format be-
              low
 Manual page ls(1) line 1 (press h for help or q to quit)
```

4. *history :*

The **history** command in Unix-like operating systems, , is used to display a list of previously executed commands in the current shell session



```
└─$ history
    1  help
    2  clear
    3  ls
    4  cd Desktop
    5  ls
    6  cd Downloads
    7  sudo dpkg -i "file google-chrome-stable_current_amd64.deb
    8  ls
```

5. *who :*

used to display information about users who are currently logged in to the system.

```
└─$ who
kali     tty7         2024-02-10 05:10 (:0)
```

6. ***whoami:***

```
└─$ whoami
kali
```

7. ***id :***

```
└─$ id
uid=1000(kali) gid=1000(kali) groups=1000(kali),4(adm),20(dialout),21(fax),24(cdrom),25(floppy),26(tape),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(users),106(netdev),118(wireshark),121(bluetooth),134(scanner),141(kaboxer),995(sambashare)
```

8. ***uname :***

```
└─$ uname
Linux
```

9. ***uptime:***

```
└─$ uptime
 08:29:19 up  3:07,  1 user,  load average: 1.03, 1.03, 1.00
```

10. ***free :***

```
└─$ free
               total        used        free      shared  buff/cache   available
Mem:         4015256      946876     2140820       50976     1159244     3068380
Swap:         999420           0      999420
```

11. ***tty :***

```
└─$ tty
/dev/pts/3
```

12. ***date :***

The **date** command is useful for various tasks, including displaying timestamps in scripts, setting system clocks, and troubleshooting time-related issues.

```
└─$ date
Sat Feb 10 08:34:08 PST 2024
```

1. Display the date and time in a specific format:

   date +"format_string"

   Replace format_string with the desired format for displaying the date and time. You can use special format specifiers to customize the output. For example:

   date +"%Y-%m-%d %H:%M:%S"

   This will display the date and time in the format YYYY-MM-DD HH:MM:SS.
2. Set the system date and time:

   date "MMDDhhmm[[CC]YY][.ss]"

   Replace "MMDDhhmm[[CC]YY][.ss]" with the desired date and time. The format is MM for month, DD for day, hh for hour (24-hour format), mm for minute, [[CC]YY] for year (optional), and [.ss] for second (optional). You need to have root privileges (using sudo) to set the system date and time.

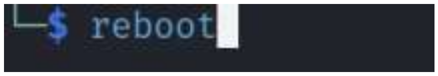
13.hostname :

```
└─$ hostname
kali
```

14.reboot :

It initiates a system reboot, causing the operating system to shut down all processes, stop the kernel, and then restart the system

It's a simple command that helps improve readability by removing previous output from the terminal, making it easier to focus on new commands and output.

15.clear : 

Program 3: a) Implement the commands that is used for Creating and Manipulating files: cat, cp, mv, rm, ls and its options, touch and their options, which is, where is, what is

Output:

1. cat :

The `cat` command in Linux is a utility used for concatenating files and printing the contents of files to the standard output (usually the terminal). Its name is derived from "concatenate".

```
└─$ cat first.txt second.txt > combined.txt

└─$ cat combined.txt
this text is in first file
This text is in second File
```

2. *mv :*

The `cp` command in Linux is used to copy files and directories from one location to another. Here's how it's commonly used:

```
ls
  orignal.txt
└─$ mv orignal.txt renamed_file.txt
ls
  renamed_file.txt
```

3. *rm :*

```
└─$ rm renamed_file.txt
```

The `rm` command is used to remove files or directories in Unix-like operating systems, including Linux and macOS. It stands for "remove".

4. *ls :*

The ls command in Unix-like operating systems (such as Linux and macOS) is used to

```
└─$ ls
Desktop    Downloads  Pictures  Templates
Documents  Music      Public    Videos
```

5. *touch :*

list the files and directories in the current directory.

The **touch** command in Unix-like operating systems (such as Linux and macOS) is used to create a new empty file or update the timestamp of an existing file.

To create a new empty file using **touch**, simply open your terminal or command prompt and type:

```
└─$ touch new_created.txt
```

6. *whereis :*

The **whereis** command in Unix-like operating systems is used to locate the binary, source, and manual page files for a command. It helps to find out the location of executable files, source code, and manual pages associated with a given command.

```
└─$ whereis ls
ls: /usr/bin/ls /usr/share/man/man1/ls.1.gz
```

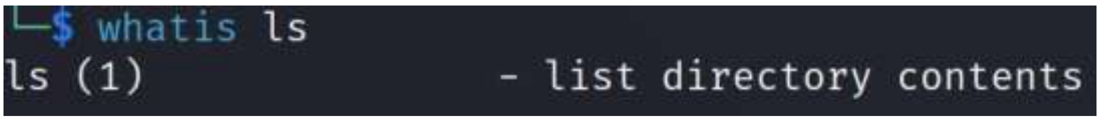
7. *which :*

The **which** command is used in Unix-like operating systems (such as Linux and macOS) to locate the executable file associated with a given command.

```
└─$ which ls
ls: aliased to ls --color=auto
```

8. whatis :

The whatis command in Unix-like operating systems is used to display a brief description of a command, function, or system call. It provides a concise summary of what the specified item does without displaying its entire manual page.



Program 4: Implement Directory oriented commands: cd, pwd, mkdir, rmdir, Comparing Files using diff, cmp, comm.

Output:

1. cd :

To use the **cd** command, you simply type **cd** followed by the path of the directory you want to change to. Here are a few examples:

```
┌──(kali㉿kali)-[~]
└─$ cd Desktop
```

2. *pwd :*

The **pwd** command stands for "print working directory." It is used in Unix-like operating systems (such as Linux, macOS, and Unix) to display the current directory path or present working directory.

```
└─$ pwd
/home/kali/Desktop/Anuj
```

3. *mkdir :*

The **mkdir** command is used to create a new directory (folder) in Unix-like operating systems, including Linux, macOS, and Unix. It stands for "make directory."

```
└─$ mkdir new_folder
```

4. *rmdir :*

The **rmdir** command is used to remove directories (folders) in Unix-like operating systems, including Linux, macOS, and Unix. However, it can only remove directories that are empty. If a directory contains files or subdirectories, you'll need to use the **rm** command with the **-r** option to recursively remove its contents before using **rmdir**.

```
└─$ rmdir new1
```

5. *Comparing Files using diff :*

The **diff** command is used in Unix-like operating systems to compare the contents of two files line by line. It displays the differences between the files in a human-readable format.

```
└─$ diff file.txt file1.txt
1c1
< first file
---
> second file
```

6. *Cmp* :

The **cmp** command is used to compare two files byte by byte and report the first differing byte and its offset (position) in each file. It is commonly used to check whether two files are identical or to find the differences between them.

```
└─$ cmp file.txt file1.txt
file.txt file1.txt differ: byte 1, line 1
```

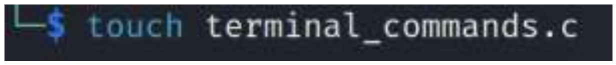
7. *comm* :

The **comm** command is used to compare two sorted files line by line. It can be particularly useful when you want to find lines that are unique to one file or lines that are common to both files.

```
└─$ comm file.txt file1.txt
first file
        second file
```

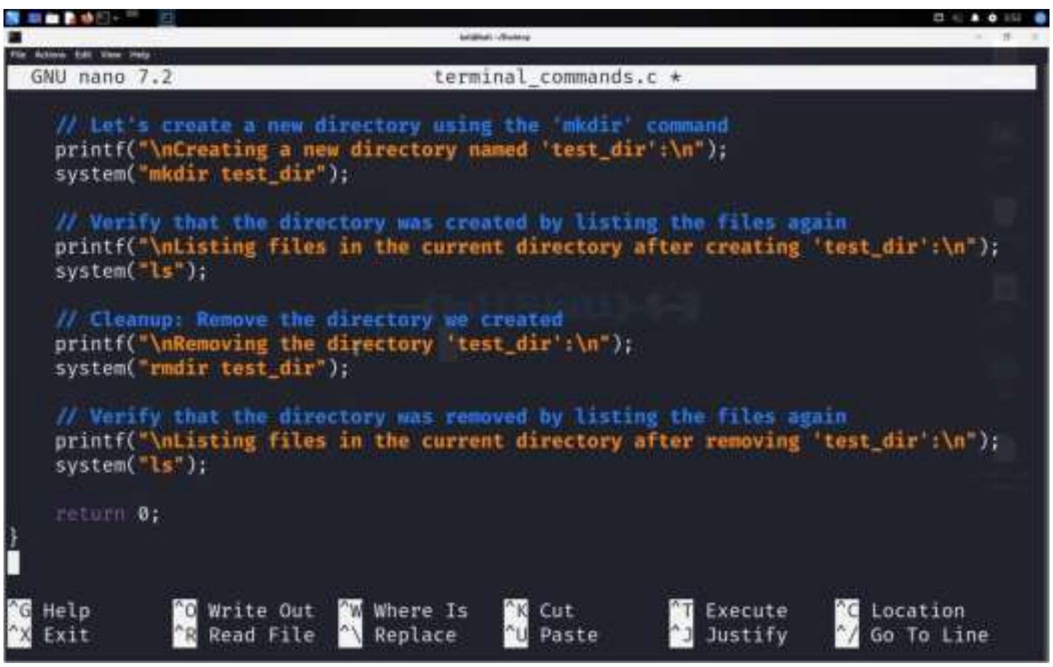
Program 5: Write a program and execute the same to demonstrate how to use terminal commands in C program (using system() function)

STEP 1- let's create a C program file named **terminal_commands.c**. You can do this using a by using the **touch** command in the terminal:



STEP 2- Now, open **terminal_commands.c** in nano text Editor using this command **nano terminal_commands.c** and add the following code to it:

Press ctrl + X to save the file



SOURCE CODE:

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

int main() {
   // Use the system() function to execute terminal commands
   // For example, let's list the files in the current directory using the 'ls' command
printf("Listing files in the current directory:\n");    system("ls");

   // Let's create a new directory using the 'mkdir' command
printf("\nCreating a new directory named 'test_dir':\n");    system("mkdir
test_dir");

   // Verify that the directory was created by listing the files again    printf("\nListing
files in the current directory after creating 'test_dir':\n");    system("ls");

   // Cleanup: Remove the directory we created
printf("\nRemoving the directory 'test_dir':\n");    system("rmdir
test_dir");

   // Verify that the directory was removed by listing the files again    printf("\nListing
files in the current directory after removing 'test_dir':\n");    system("ls");

   return 0;
}
```

OUTPUT:

Compile the C Program: Now, let's compile the C program using a C compiler. We'll use `gcc` for this:

```
└─$ gcc -o terminal_commands terminal_commands.c
```

Execute the Program: Finally, let's run the compiled executable:

6: Write a program to implement process concepts using C language by printing process Id.

ALGORITHM:

1. The header files stdio.h and unistd.h are included to use the printf() function and getpid() function, respectively.
2. The main() function is declared.
3. The getpid() function is called to get the process ID, and the result is stored in the variable pid.
4. The process ID is printed using the printf() function.
5. The main function is ended with the return 0 statement.
6. It will print the process ID.

SOURCE CODE:

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

int main() {
   pid_t p;

   p = fork();

   if (p < 0) {
      printf("Error in forking\n");
   } else if (p == 0) {
      printf("Child process. PID=%d\n", getpid());
   } else {
      printf("Parent process. Child PID=%d\n", p);
   }

   return 0;
}
```

Program

OUTPUT:

```
Practical % gcc os.c
Practical % ./a.out
Parent process. Child PID=5661
Child process. PID=5661
```

7: Write a program to create and execute process using fork() and exec() system calls.

ALGORITHM:

1. We will create two files with the extension of ".c." After their creation, we will write respective codes in them and execute them to see the result.
2. We have used the "unistd.h" header as it contains all information of families of exec function.
3. We have printed the current process's id.
4. We have created a character array having NULL in the end for the termination.
5. We have called the execfile function.
6. The output of the respective codes can be obtained by using the following commands.

- gcc -o execFileDemo.c execFileDemo
- gcc execfile.c -o execfile
- ./execFileDemo
- ./execfile

SOURCE CODE:

1. execfile.c

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

int main(int argc, char *argv[]) {
printf("We are in execfile.c\n");
printf("PID of execfile.c = %ld\n", (long)getpid());
printf("--------------------------\n");
printf("--------------------------\n");

return 0;
}
```

2. execFileDemo.c

Program

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

int main(int argc, char *argv[]) {
    printf("PID of execFileDemo.c = %d\n", getpid());
    char *args[] = {"h", "c", "hello", NULL};
execv("./execfile", args);      printf("Back to
execFileDemo.c\n");     printf("------------------
---------\n");     printf("---------------------------
\n");     return 0;
}
```

OUTPUT:

```
Practical % gcc -o exefile execfile.c
Practical % gcc -o execFiledemo execFiledemo.c
Practical % ./execFileDemo
PID of execFileDemo.c = 6092
Back to execFileDemo.c
We are in execfile.c
PID of execfile.c = 6093
```

8: Write a C program to implement FCFS (First Come First Serve) and SJF (Shortest Job First) scheduling algorithms.

FCFS: First-Come, First-Served Scheduling Algorithm

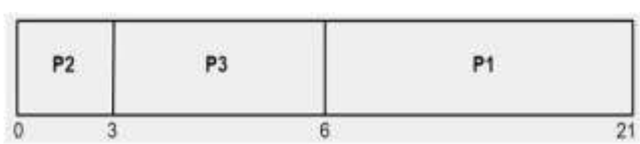
First Come, First Served (FCFS) is the CPU scheduling algorithm in which the CPU is allocated to the processes in the order they are queued in the ready queue. FCFS follows non-pre emptive scheduling which mean once the CPU is allocated to a process it does not leave the CPU until the process will not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

Example

Let's say, there are four processes arriving in the sequence as P2, P3, P1 with their corresponding execution time as shown in the table below. Also, taking their arrival time to be 0.

Process	Order of arrival	Execution time in msec
P1	3	15
P2	1	3
P3	2	3

Gantt chart showing the waiting time of processes P1, P2 and P3 in the system



As shown above,

The waiting time of process P2 is 0

Program

The waiting time of process P3 is 3 The waiting time

of process P1 is 6

Average AWT time = (0 + 3 + 6) / 3 = 3 msec.

As we have taken arrival time to be 0 therefore turnaround time and completion time will be same.

SOURCE CODE:

```
#include <stdio.h>

int waitingtime(int proc[], int n, int burst_time[], int wait_time[]) {
wait_time[0] = 0;     for (int i = 1; i < n; i++)
        wait_time[i] = burst_time[i - 1] + wait_time[i - 1];      return
0;
}

int turnaroundtime(int proc[], int n, int burst_time[], int wait_time[], int tat[]) {
for (int i = 0; i < n; i++)
        tat[i] = burst_time[i] + wait_time[i];     return
0;
}

int avgtime(int proc[], int n, int burst_time[]) {     int
wait_time[n], tat[n], total_wt = 0, total_tat = 0;

    waitingtime(proc, n, burst_time, wait_time);
    turnaroundtime(proc, n, burst_time, wait_time, tat);

    printf("Processes  Burst   Waiting Turn around \n");     for (int
i = 0; i < n; i++) {
        total_wt = total_wt + wait_time[i];
total_tat = total_tat + tat[i];
        printf(" %d\t  %d\t\t %d \t%d\n", i + 1, burst_time[i], wait_time[i], tat[i]);
    }
    printf("Average waiting time = %f\n", (float)total_wt / (float)n);
printf("Average turn around time = %f\n", (float)total_tat / (float)n);

    return 0;
}
int main() {
    int proc[] = {1, 2, 3};
    int n = sizeof proc / sizeof proc[0];
int burst_time[] = {5, 8, 12};
avgtime(proc, n, burst_time);     return 0;
}
```

Program

OUTPUT:

```
: practical % gcc os.c
: practical % ./a.out
Processes  Burst   Waiting Turn around
 1        5             0      5
 2        8             5      13
 3        12            13     25
Average waiting time = 6.000000
Average turn around time = 14.333333
```

SJF: Shortest Job First Scheduling Algorithm

The Shortest Job First (SJF) algorithm is a non-**pre-emptive** CPU scheduling algorithm designed to minimize the average waiting time of processes. It operates by sorting all processes based on their arrival times. From this sorted list, the algorithm selects the process with the shortest burst time, signifying the time it requires to complete its task. The CPU is then allocated to this selected process, allowing it to execute for its designated burst time. Upon completion of its execution, the process is removed from the list, and the algorithm iterates, selecting the next shortest job until all processes are completed. This method ensures that shorter processes are prioritized, potentially reducing overall waiting times and improving system efficiency.

SOURCE CODE:

```
#include<stdio.h>

int main() {
   int bt[20], p[20], wt[20], tat[20], i, j, n, total = 0, pos, temp;     float
avg_wt, avg_tat;
   printf("Enter number of processes:");     scanf("%d",
&n);
   printf("\nEnter Burst Time:\n");
   for(i = 0; i < n; i++) {
printf("P%d:", i + 1);        scanf("%d",
&bt[i]);
      p[i] = i + 1;
   }
```

```
   for(i = 0; i < n; i++) {
pos = i;
      for(j = i + 1; j < n; j++) {
if(bt[j] < bt[pos])              pos
= j;
      }
      temp = bt[i];
bt[i] = bt[pos];
      bt[pos] = temp;

      temp = p[i];
p[i] = p[pos];
      p[pos] = temp;
   }
   wt[0] = 0;
   for(i = 1; i < n; i++) {
wt[i] = 0;        for(j = 0; j
< i; j++)           wt[i] +=
bt[j];        total += wt[i];
   }
   avg_wt = (float)total / n;
total = 0;
   printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
for(i = 0; i < n; i++) {        tat[i] = bt[i] + wt[i];        total += tat[i];
      printf("\nP%d\t%d\t\t%d\t\t%d", p[i], bt[i], wt[i], tat[i]);
   }
   avg_tat = (float)total / n;
   printf("\n\nAverage Waiting Time = %f", avg_wt);
printf("\nAverage Turnaround Time = %f\n", avg_tat);     return 0;
}
```

OUTPUT:

```
Enter number of processes:5

Enter Burst Time:
P1:4
P2:3
P3:7
P4:2
P5:1

Process Burst Time      Waiting Time    Turnaround Time

P5      1               0               1
P4      2               1               3
P2      3               3               6
P1      4               6               10
P3      7               10              17

Average Waiting Time = 4.000000
Average Turnaround Time = 7.400000
```

Program 9: Write a C program to implement Priority Scheduling and RR (Round-Robin) scheduling algorithms.

ROUND ROBIN : Round Robin Scheduling Algorithm

The Round Robin (RR) scheduling algorithm is a preemptive CPU scheduling technique commonly used in operating systems. In RR, each process is allocated a fixed time slice, known as a time quantum, during which it can execute on the CPU. Processes are placed in a ready queue, and the scheduler selects the process at the front of the queue to run on the CPU. If a process completes within its time quantum, it is removed from the system. Otherwise, it is preempted and placed at the end of the ready queue to await its next turn. RR ensures fairness by providing each process with an equal share of CPU time, making it suitable for time-sharing systems where multiple users need concurrent access to system resources. However, the choice of time quantum affects the trade-off between fairness and responsiveness, as shorter time slices lead to more frequent context switches.

SOURCE CODE:

```
#include<stdio.h>
int main() {
int i,limit,total=0,x,counter=0,time_quantum;
int wait_time=0,turnaround_time=0,arrival_time[10],burst_time[10],temp[10];
float average_wait_time,average_turnaround_time; printf("\nEnter Total Number
of Processes:"); scanf("%d",&limit); x=limit;
for(i=0;i<limit;i++) {
printf("\nEnter Details of Process[%d]\n",i+1);
printf("Arrival Time:");
scanf("%d",&arrival_time[i]); printf("Burst
Time:"); scanf("%d",&burst_time[i]);
temp[i]=burst_time[i];
}
printf("\nEnter Time Quantum:"); scanf("%d",&time_quantum);
printf("\nProcess ID\tBurst Time\tTurnaround Time\tWaiting Time\n"); for(total=0,i=0;x!=0;) {
if(temp[i]<=time_quantum&&temp[i]>0) { total=total+temp[i];
```

```
temp[i]=0;
counter=1;
}
else if(temp[i]>0) {
temp[i]=temp[i]-time_quantum;
total=total+time_quantum;
}
if(temp[i]==0&&counter==1) { x--;
printf("\nProcess[%d]\t%d\t\t%d\t\t%d",i+1,burst_time[i],total-arrival_time[i],total-arrival_time[i]-burst_time[i]);
wait_time=wait_time+total-arrival_time[i]-burst_time[i]; turnaround_time=turnaround_time+total-arrival_time[i];
counter=0;
}
if(i==limit-1) {
i=0;
}
else if(arrival_time[i+1]<=total) { i++;
}
else {
i=0;
}
}
average_wait_time=wait_time*1.0/limit;
average_turnaround_time=turnaround_time*1.0/limit; printf("\n\nAverage
Waiting Time:%f",average_wait_time); printf("\nAverage Turnaround
Time:%f\n",average_turnaround_time); return 0;
}
```

OUTPUT:

```
Enter Total Number of Processes:4

Enter Details of Process[1]
Arrival Time:0
Burst Time:4

Enter Details of Process[2]
Arrival Time:1
Burst Time:5

Enter Details of Process[3]
Arrival Time:2
Burst Time:3

Enter Details of Process[4]
Arrival Time:3
Burst Time:6

Enter Time Quantum:2

Process ID      Burst Time      Turnaround Time Waiting Ti
me

Process[1]      4               10              6
Process[3]      3               11              8
Process[2]      5               15              10
Process[4]      6               15              9

Average Waiting Time:8.250000
Average Turnaround Time:12.750000
```