

Experiment no.1

NAME: Atharv Vijay Deshpande

PRN: 21410044

BATCH: EN – 3

Title: Linux Installation procedure and basic linux commands

❖ Installation Procedure:-

1.Download Ubuntu

- Go to the official Ubuntu website and download the latest version of the Ubuntu ISO file.

2. Create a Bootable USB Drive

- You'll need a USB drive with at least 8 GB of storage.
- Use a tool like Rufus (for Windows) or Etcher (for Windows, macOS, and Linux) to create a bootable USB drive.
- Open the tool, select the downloaded Ubuntu ISO file, and follow the instructions to create the bootable USB.

3. Boot from the USB Drive

- Insert the bootable USB drive into your PC.
- Restart your PC and enter the BIOS/UEFI settings (usually by pressing a key like F2, F12, DEL, or ESC during startup).
- Change the boot order to prioritize the USB drive.
- Save the changes and exit the BIOS/UEFI settings. Your PC should now boot from the USB drive.

4. Start the Installation Process

- Once Ubuntu boots from the USB, you'll see the option to "Try Ubuntu" or "Install Ubuntu." Select "Install Ubuntu."
- Choose your preferred language and click "Continue."

5. Prepare the Installation

- Select your keyboard layout and click “Continue.”
- Choose the installation type. You can either:
 - **Erase disk and install Ubuntu:** This will delete all data on the disk and install Ubuntu as the only operating system.
 - **Install Ubuntu alongside Windows:** This will create a dual-boot setup, allowing you to choose between Ubuntu and Windows at startup.
- Click “Install Now” and confirm your choice.

6. Set Up Your System

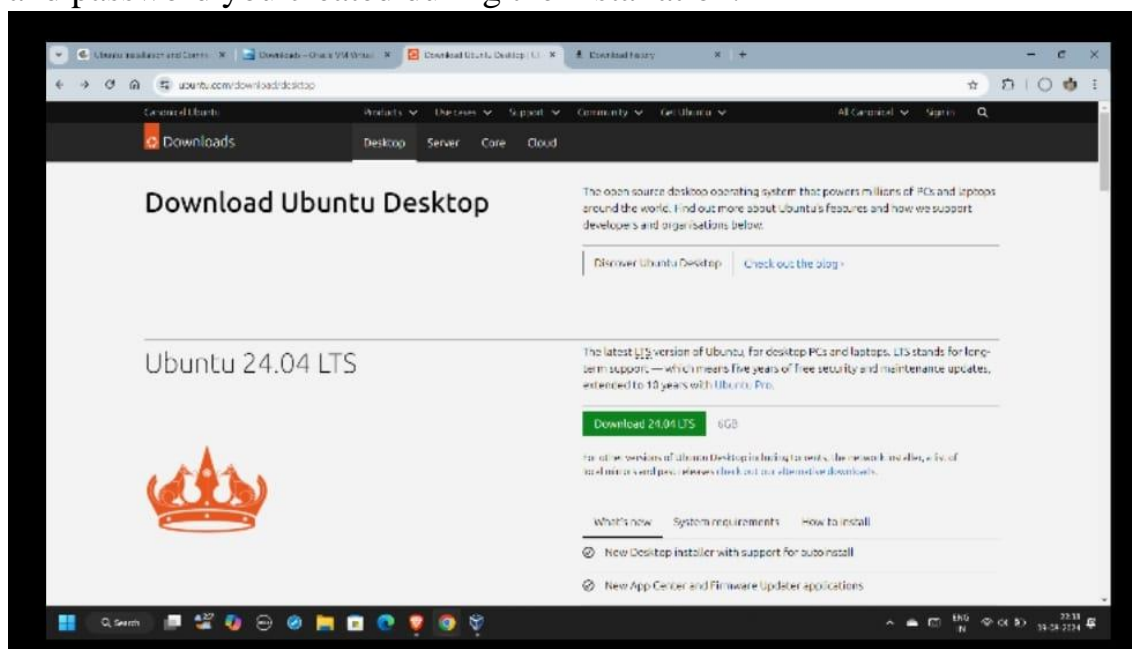
- Select your time zone and click “Continue.”
- Enter your personal details, such as your name, computer name, username, and password. Click “Continue.”

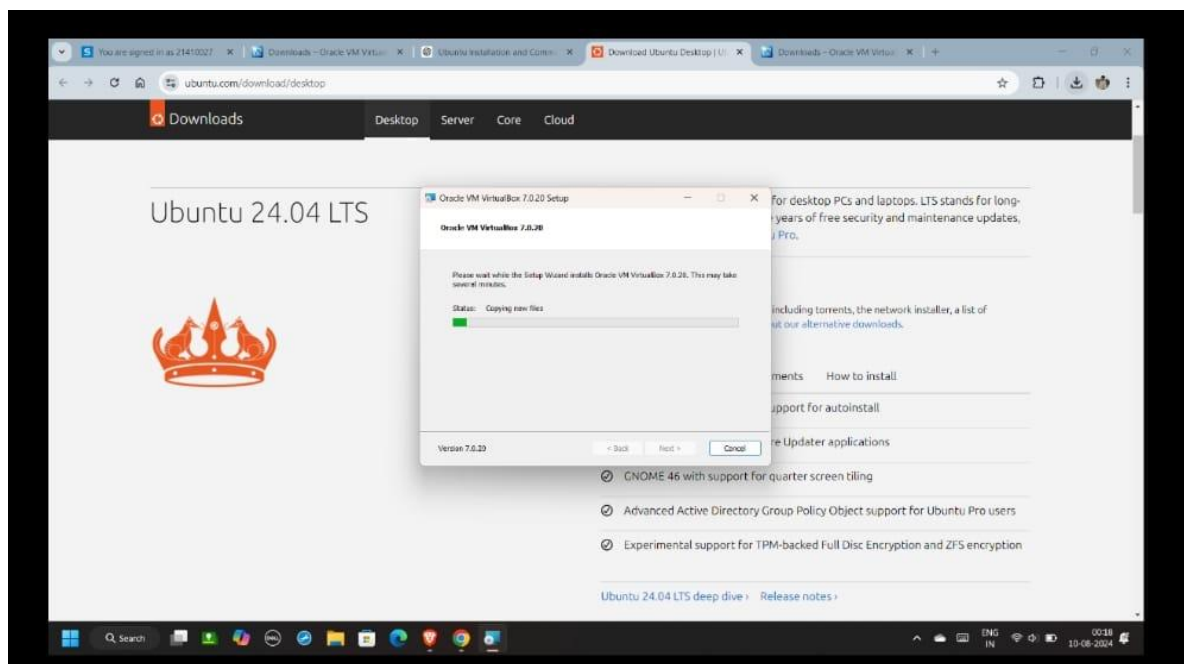
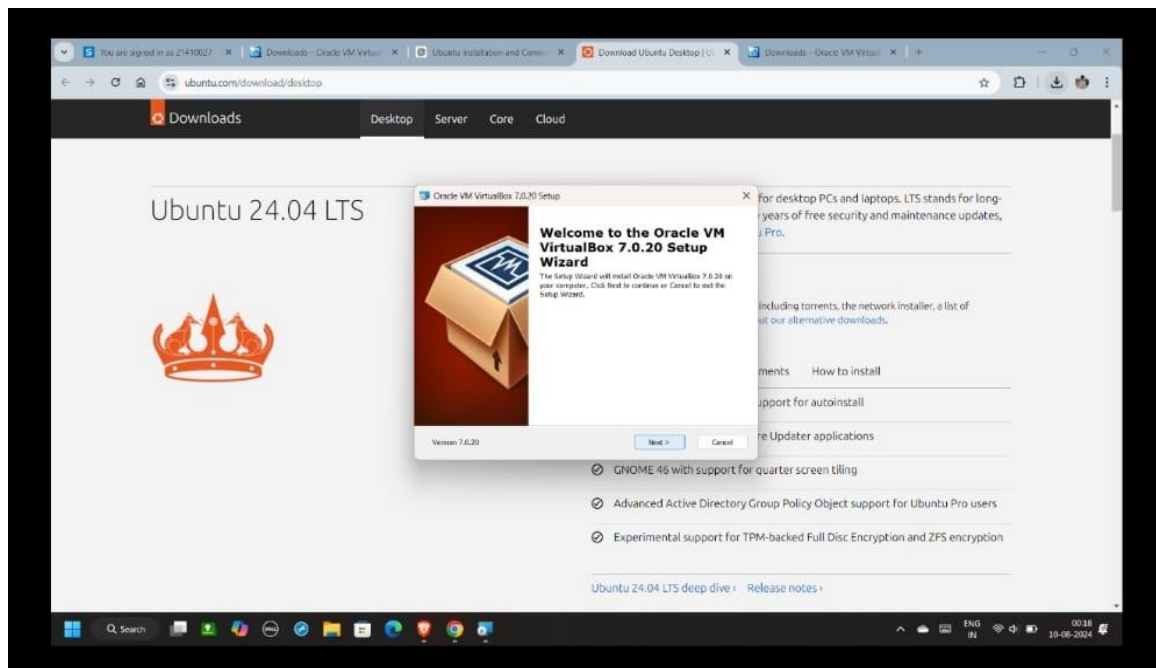
7. Complete the Installation

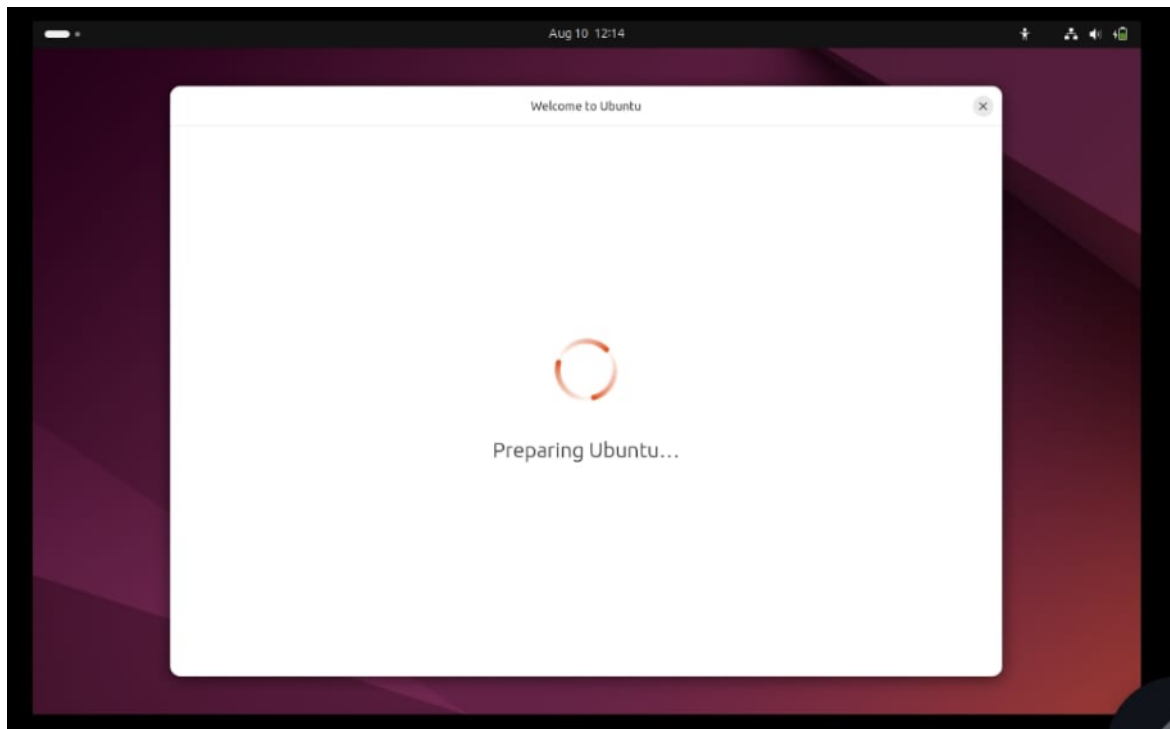
- The installer will copy files and install Ubuntu. This process may take some time.
- Once the installation is complete, you’ll be prompted to restart your PC. Remove the USB drive and press “Enter.”

8. First Boot

- After restarting, your PC will boot into Ubuntu. Log in with the username and password you created during the installation.

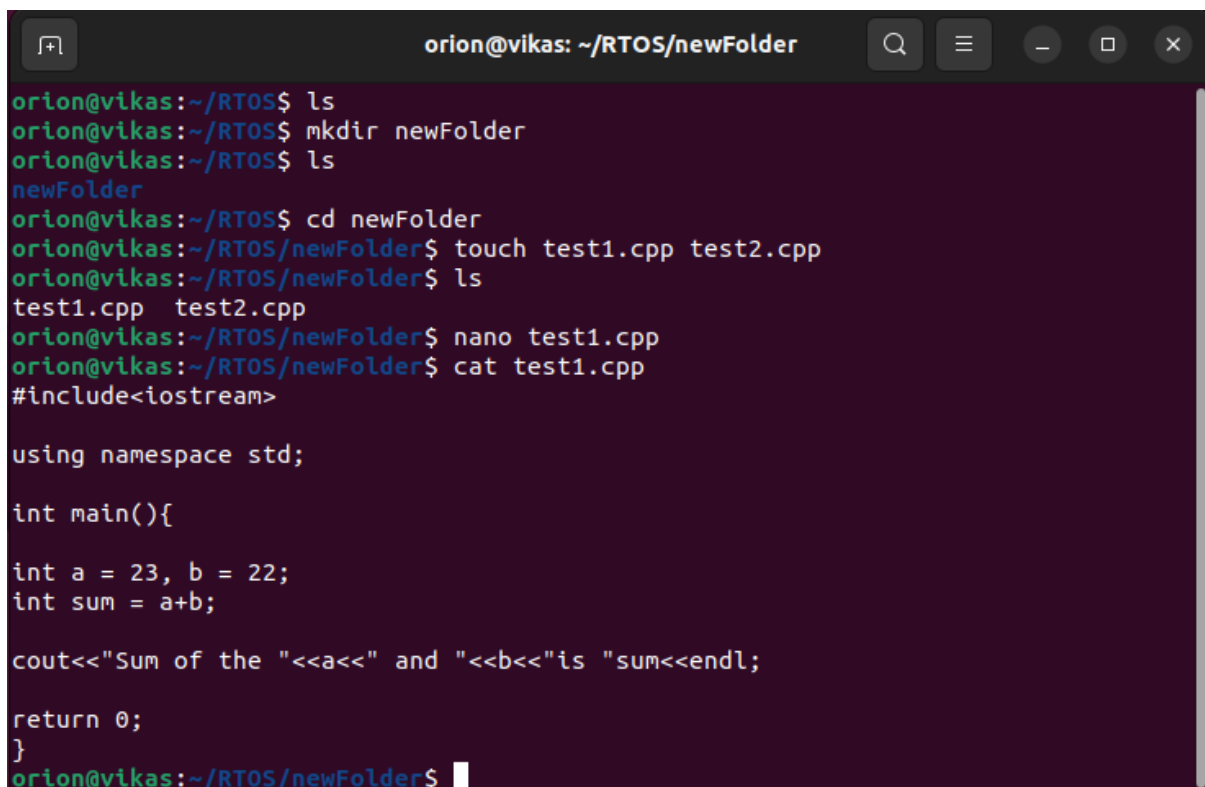






➤ Linux commands:

1. ls: The workhorse of directory listing. Use it to see what files and directories are in your current location.
2. pwd: Print the current working directory
3. cd: Change directory. Navigate around your file system with this command.
4. mkdir: Create a new directory (folder).
5. mv: Move or rename files and directories.
6. cp: Copy files and directories.
7. rm: Delete files or directories
8. touch: Create blank or empty files.
9. Nano: It is a command-line text editor that's simple and easy to use.
10. Gedit: It is a graphical text editor that comes pre-installed with the GNOME desktop environment. It's user-friendly and great for editing text files with a graphical interface.



```
orion@vikas: ~/RTOS/newFolder
orion@vikas:~/RTOS$ ls
orion@vikas:~/RTOS$ mkdir newFolder
orion@vikas:~/RTOS$ ls
newFolder
orion@vikas:~/RTOS$ cd newFolder
orion@vikas:~/RTOS/newFolder$ touch test1.cpp test2.cpp
orion@vikas:~/RTOS/newFolder$ ls
test1.cpp  test2.cpp
orion@vikas:~/RTOS/newFolder$ nano test1.cpp
orion@vikas:~/RTOS/newFolder$ cat test1.cpp
#include<iostream>

using namespace std;

int main(){

int a = 23, b = 22;
int sum = a+b;

cout<<"Sum of the "<<a<<" and "<<b<<"is "sum<<endl;

return 0;
}
orion@vikas:~/RTOS/newFolder$
```

```
orion@vikas: ~/RTOS/newFolder
GNU nano 6.2 test1.cpp
#include<iostream>

using namespace std;

int main(){

int a = 23, b = 22;
int sum = a+b;

cout<<"Sum of the "<<a<<" and "<<b<<"is "sum<<endl;

return 0;
}

[ Read 13 lines ]
^G Help      ^O Write Out  ^W Where Is   ^K Cut        ^T Execute    ^C Location
^X Exit      ^R Read File  ^\ Replace    ^U Paste      ^J Justify    ^_ Go To Line
```

```
orion@vikas: ~/RTOS
orion@vikas:~/RTOS/newFolder$ touch test2.cpp
orion@vikas:~/RTOS/newFolder$ ls
test1.cpp  test2.cpp
orion@vikas:~/RTOS/newFolder$ rm test2.cpp
orion@vikas:~/RTOS/newFolder$ rm test1.cpp
orion@vikas:~/RTOS/newFolder$ ls
orion@vikas:~/RTOS/newFolder$ cd ..
orion@vikas:~/RTOS$ ls
newFolder
orion@vikas:~/RTOS$ rmdir newFolder
orion@vikas:~/RTOS$ ls
orion@vikas:~/RTOS$
```

❖ **Conclusion**

- ❖ We learned how to install Linux Distribution inside a Virtual Machine and how to use it without actually needing the hardware of the machine.
- ❖ We also learned how Linux focuses more on terminal commands rather than GUI, due to which we learned various Linux commands which helped in understanding the Linux OS and how to use it.

Experiment No. 2

Name: Atharv Vijay Deshpande

PRN: 21410044

Batch: EN-3

Title: Execution of C program in Linux


Part A: Single file C program

Step 1: Open a terminal in Linux.

Step 2: Create a new file using the command:

nano addNum.cpp

Step 3: Write the code into addNum.cpp.



```
orion@vikas: ~/rtosLab/exp2
GNU nano 7.2 addNum.cpp
#include <iostream>
using namespace std;

int main() {

    int first_number, second_number, sum;

    cout << "Enter two integers: ";
    cin >> first_number >> second_number;

    // sum of two numbers is stored in variable sumOfTwoNumbers
    sum = first_number + second_number;

    cout << first_number << " + " << second_number << " = " << sum;

    return 0;
}
```

[Read 18 lines]

^G Help ^O Write Out ^W Where Is ^K Cut ^T Execute ^C Location
^X Exit ^R Read File ^_ Replace ^U Paste ^J Justify ^/ Go To Line

Step 4: Save and exit the editor by pressing CTRL + S, CTRL + X, and Enter.

Step 5: Compile the C program using the GCC compiler


```
orion@vikas: ~/rtosLab/exp2
orion@vikas:~/rtosLab/exp2$ nano addNum.cpp
orion@vikas:~/rtosLab/exp2$ g++ addNum.cpp -o addNum
orion@vikas:~/rtosLab/exp2$ ls
addNum  addNum.cpp
orion@vikas:~/rtosLab/exp2$ ./addNum
Enter two integers: 4 5
4 + 5 = 9orion@vikas:~/rtosLab/exp2$
```

Part B: Multifile C Program

2. Procedure (Multifile)

- Step 1: Open a terminal in Linux.
- Step 2: Create the header file functions.h using the command:
`nano functions.h`
- Step 3: Write the content of operations.h and save it.

```
GNU nano 7.2 functions.h
// functions.h
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

extern int sum(int, int);
extern int sub(int, int);
extern int multiply(int, int);

#endif

[ Read 9 lines ]
^G Help      ^O Write Out ^W Where Is  ^K Cut       ^T Execute   ^C Location
^X Exit      ^R Read File ^\ Replace  ^U Paste     ^J Justify   ^_ Go To Line
```

- Step 4: Create the implementation file addNew.cpp:

nano addNew.cpp

- Step 5: Write the content of operations.c and save it.



```
GNU nano 7.2 addNew.cpp
// addNew.cpp
#include "functions.h"

int sum(int a, int b) {
    return a + b;
}

int sub(int a, int b) {
    return a - b;
}

int multiply(int a, int b) {
    return a * b;
}

[ Read 15 lines ]
^G Help      ^O Write Out ^W Where Is  ^K Cut       ^T Execute   ^C Location
^X Exit      ^R Read File ^\ Replace   ^U Paste     ^J Justify   ^_ Go To Line
```

- Step 6: Create the main file second.cpp:

nano second.cpp



```
GNU nano 7.2 second.cpp
// second.cpp
#include <iostream>
#include "functions.h"
using namespace std;

int main() {
    int a = 10, b = 5;

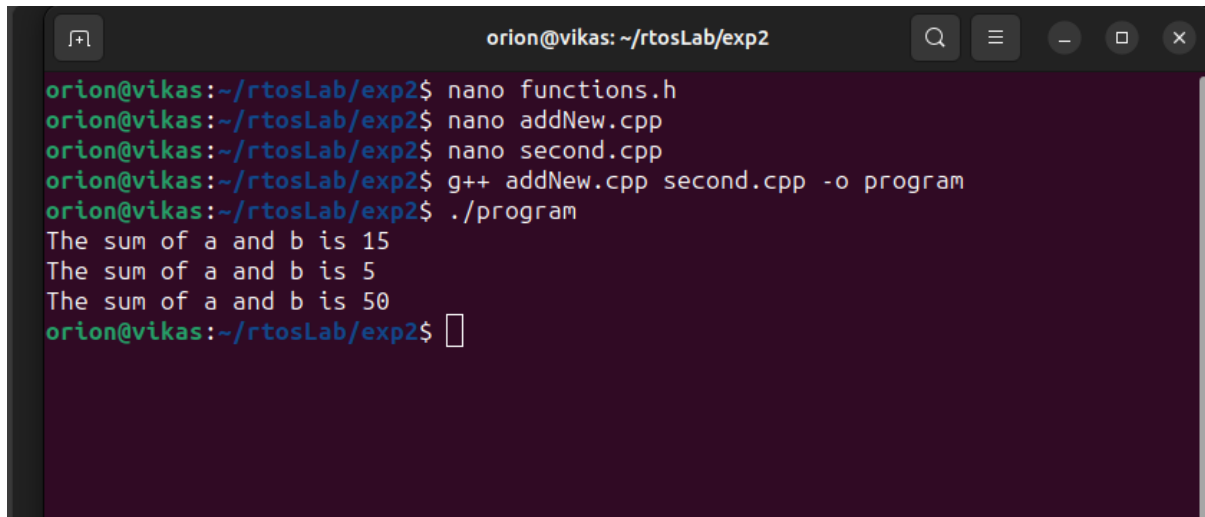
    cout<<"The sum of a and b is "<<sum(a,b)<<endl;
    cout<<"The sum of a and b is "<<sub(a,b)<<endl;
    cout<<"The sum of a and b is "<<multiply(a,b)<<endl;

    return 0;
}

[ Read 16 lines ]
^G Help      ^O Write Out ^W Where Is  ^K Cut       ^T Execute   ^C Location
^X Exit      ^R Read File ^\ Replace   ^U Paste     ^J Justify   ^_ Go To Line
```

- Step 7: Write the content of second.cpp and save it.

- Step 8: Compile the multifile program using the GCC compiler:
- Step 9: Execute the compiled program:
- Step 10: Observe the result.



```

orion@vikas: ~/rtosLab/exp2
orion@vikas:~/rtosLab/exp2$ nano functions.h
orion@vikas:~/rtosLab/exp2$ nano addNew.cpp
orion@vikas:~/rtosLab/exp2$ nano second.cpp
orion@vikas:~/rtosLab/exp2$ g++ addNew.cpp second.cpp -o program
orion@vikas:~/rtosLab/exp2$ ./program
The sum of a and b is 15
The sum of a and b is 5
The sum of a and b is 50
orion@vikas:~/rtosLab/exp2$ 

```

❖ **Conclusion :** In the g++ compilation of single and multiple files in C++, we explored how to compile and link programs effectively. For single-file compilation, we used `g++ filename.cpp -o output`, while for multi-file projects, we combined source files with `g++ file1.cpp file2.cpp -o output`. This ensures modular code development, allowing efficient compilation and linkage of large-scale C++ projects.

Answer to the Following Questions

1. Meaning and Use of GCC in Linux:

- GCC stands for GNU Compiler Collection. It is a compiler system produced by the GNU Project that supports various programming languages, including C, C++, and others. In Linux, GCC is primarily used to compile C programs. When you write a program in C, it is written in a human-readable format called source code. GCC takes this source code and compiles it into machine code (an executable file) that the computer can run.

2. Advantages and Disadvantages of Multifile C Programming:

- **Advantages:**

- **Modularity:** By dividing the program into multiple files, you can keep related functions and definitions together, making the codebase easier to manage and understand. Each file can handle a specific aspect of the program, which simplifies both development and debugging.
- **Reusability:** Functions and definitions written in one file can be reused in other projects or parts of the same project. For example, if you have a set of utility functions, you can place them in one file and reuse them whenever needed.
- **Collaboration:** When working in a team, different team members can work on different files simultaneously without causing conflicts. This is particularly useful in large projects.
- **Easier Debugging:** Since the code is divided into smaller parts, isolating and fixing bugs becomes easier. You can test individual components separately before integrating them.

○ **Disadvantages:**

- **Complexity:** Managing multiple files can be more complex, especially for beginners. You have to ensure that the correct files are compiled and linked together, and that header files are properly included.
- **Dependency Management:** If one file depends on another, changes in one file may necessitate changes in others. This can lead to increased maintenance effort, especially as the project grows.
- **Increased Compilation Time:** In some cases, compiling multiple files can take more time compared to a single-file program, especially if there are many dependencies and large codebases.
- **Error Tracking:** While modularity aids in debugging, it can sometimes make it harder to track errors that span multiple files, requiring careful attention to the interactions between different parts of the program.

Experiment No: 3

Name: Atharv Vijay Deshpande

PRN: 21410044

Batch: EN-3

.....

Title: - Shell Scripting

Part A: Shell Scripting of Arithmetic Operations

- **Step 1:** Open a terminal and create the script file using:
- **Step 2:** Write the above script into calculator.sh and save it.
- **Step 3:** Make the script executable:
- **Step 4:** Run the script:
- **Step 5:** Enter the numbers and the operation when prompted, and observe the result.

Result:

```
ubuntu@Prasadmw:/Shell_Scripts$ chmod +x calculator.sh
ubuntu@Prasadmw:/Shell_Scripts$ ./calculator.sh
Enter first number:
5
Enter second number:
6
Sum:11
Difference:-1
Product:30
Division:0
ubuntu@Prasadmw:/Shell_Scripts$
```

Part B: Shell Scripting of Conditional Statements

- **Step 1:** Open a terminal and create the script file using:
- **Step 2:** Write script into conditional.sh and save it:chmod +x check_age.sh
- **Step 4:** Run the script: ./check_age.sh
- **Step 5:** Enter your age when prompted and observe the result.

Result:

```
ubuntu@ubuntu:~/shell_script$ ls
conditional_statement
ubuntu@ubuntu:~/shell_script$ chmod +x conditional_statement
ubuntu@ubuntu:~/shell_script$ ./conditional_statement
Please enter your age:
22
You are older than 18.
ubuntu@ubuntu:~/shell_script$
```

Part C: Shell Scripting Using Loops and Switch Case

- Step 1: Open a terminal and create the script file using:
- Step 2: Write the above script into switch.sh and save it.
- Step 3: Make the script executable: chmod +x fruit_check.sh
- Step 4: Run the script: ./fruit_check.sh

- Step 5: Enter the name of a fruit when prompted and observe the result.

Result:

```
ubuntu@ubuntu:~/shell_script$ touch switch
ubuntu@ubuntu:~/shell_script$ ls
conditional_statement  switch
ubuntu@ubuntu:~/shell_script$ chmod +x switch
ubuntu@ubuntu:~/shell_script$ ./switch
What is your favorite fruit? (apple/orange/banana)
apple
You like apples!
ubuntu@ubuntu:~/shell_script$
```

Conclusion:

Shell scripting is an essential tool in Linux, offering the ability to automate tasks, streamline processes, and manage system operations efficiently. Through the execution of various scripts for arithmetic operations, conditional statements, and control structures like loops and switch cases, we observed how shell scripting simplifies complex tasks, reduces manual intervention, and enhances productivity.

Answer the following:

1. Significance of Shell Script in Linux:

- Shell scripting in Linux allows users to automate tasks, simplify repetitive commands, and create more efficient workflows. It is a powerful tool for managing system operations, processing files, and configuring software.

2. Types of Shells:

- **Bash (Bourne Again Shell):** The most common shell in Linux, offering extensive scripting capabilities and user-friendly features.
- **Sh (Bourne Shell):** A simple shell used mainly for scripting.
- **Csh (C Shell):** Similar to the C programming language, useful for users familiar with C syntax.
- **Ksh (Korn Shell):** Combines features of Bourne Shell and C Shell, offering powerful scripting options.
- **Zsh (Z Shell):** A highly customizable shell with advanced features for interactive use.

Experiment No. 4

Name: Atharv Vijay Deshpande

PRN: 21410044

Batch: EN-3

.....

Title: Creation of a group and user and deletion into terminal.

Objective:

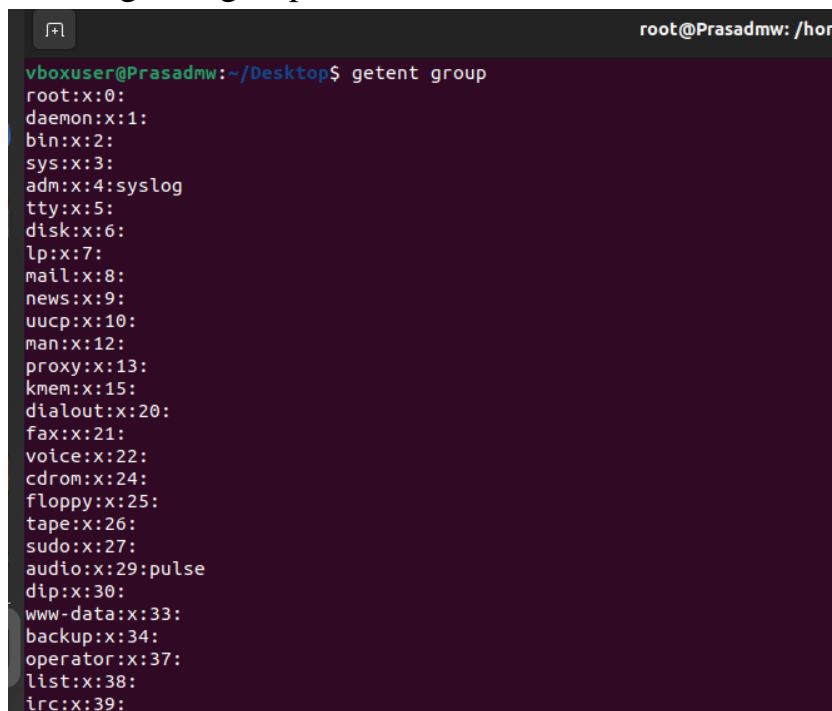
To Create Group and to add user to it using Linux Terminal.

Procedure:

Procedure steps:

Step 1: List all groups name with username

Command: getent group

A terminal window with a dark purple background. The prompt is 'root@Prasadmw: /hor'. The command 'vboxuser@Prasadmw:~/Desktop\$ getent group' has been entered. The output lists system groups and their members: root:x:0:, daemon:x:1:, bin:x:2:, sys:x:3:, adm:x:4:syslog, tty:x:5:, disk:x:6:, lp:x:7:, mail:x:8:, news:x:9:, uucp:x:10:, man:x:12:, proxy:x:13:, kmem:x:15:, dialout:x:20:, fax:x:21:, voice:x:22:, cdrom:x:24:, floppy:x:25:, tape:x:26:, sudo:x:27:, audio:x:29:pulse, dip:x:30:, www-data:x:33:, backup:x:34:, operator:x:37:, list:x:38:, and irc:x:39:.

```
root@Prasadmw: /hor
vboxuser@Prasadmw:~/Desktop$ getent group
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:syslog
tty:x:5:
disk:x:6:
lp:x:7:
mail:x:8:
news:x:9:
uucp:x:10:
man:x:12:
proxy:x:13:
kmem:x:15:
dialout:x:20:
fax:x:21:
voice:x:22:
cdrom:x:24:
floppy:x:25:
tape:x:26:
sudo:x:27:
audio:x:29:pulse
dip:x:30:
www-data:x:33:
backup:x:34:
operator:x:37:
list:x:38:
irc:x:39:
```

```
root@Prasadmw:~# cat /etc/passwd | grep :x:
nogroup:x:65534:
systemd-journal:x:101:
systemd-network:x:102:
systemd-resolve:x:103:
crontab:x:104:
messagebus:x:105:
systemd-timesync:x:106:
input:x:107:
sgx:x:108:
kvm:x:109:
render:x:110:
syslog:x:111:
_ssh:x:112:
tss:x:113:
bluetooth:x:114:
ssl-cert:x:115:
uudd:x:116:
systemd-oom:x:117:
tcpdump:x:118:
avahi-autoipd:x:119:
netdev:x:120:
avahi:x:121:
lpadmin:x:122:
rtkit:x:123:
whoopsie:x:124:
sssd:x:125:
fwupd-refresh:x:126:
nm-openvpn:x:127:
scanner:x:128:saned
saned:x:129:
colord:x:130:
geoclue:x:131:
```

Step 2: Add Group

Command : `sudo addgroup groupname`

```
vboxuser@Prasadmw:~/Desktop$ su
Password:
root@Prasadmw:/home/vboxuser/Desktop# sudo addgroup pmw1
Adding group `pmw1' (GID 1001) ...
Done.
root@Prasadmw:/home/vboxuser/Desktop#
```

Step 3: Delete Group

Command : `sudo delgroup groupname`

```
root@Prasadmw:/home/vboxuser/Desktop# sudo addgroup pmw1
Adding group `pmw1' (GID 1001) ...
Done.
root@Prasadmw:/home/vboxuser/Desktop# sudo delgroup pmw1
Removing group `pmw1' ...
Done.
root@Prasadmw:/home/vboxuser/Desktop#
```

Step 4: Add user

Command : `sudo adduser username`

```
root@Prasadmw:/home/vboxuser/Desktop# sudo adduser pmw1
Adding user `pmw1' ...
Adding new group `pmw1' (1001) ...
Adding new user `pmw1' (1001) with group `pmw1' ...
Creating home directory `/home/pmw1' ...
Copying files from `/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for pmw1
Enter the new value, or press ENTER for the default
  Full Name []: Prasad W
    Room Number []: 2
    Work Phone []: 7666237113
    Home Phone []: 999999999
      Other []: 0
Is the information correct? [Y/n] y
root@Prasadmw:/home/vboxuser/Desktop# getent group
root:x:0:
daemon:x:1:
bin:x:2:
sys:x:3:
adm:x:4:syslog
tty:x:5:
disk:x:6:
lp:x:7:
mail:x:8:
news:x:9:
uucp:x:10:
man:x:12:
proxy:x:13:
kmem:x:15:
```

```
root@Prasadmw: /hom
nngroup:x:65534:
systemd-journal:x:101:
systemd-network:x:102:
systemd-resolve:x:103:
crontab:x:104:
messagebus:x:105:
systemd-timesync:x:106:
input:x:107:
sgx:x:108:
kvm:x:109:
render:x:110:
syslog:x:111:
_ssh:x:112:
tss:x:113:
bluetooth:x:114:
ssl-cert:x:115:
uuidd:x:116:
systemd-oom:x:117:
tcpdump:x:118:
avahi-autoipd:x:119:
netdev:x:120:
avahi:x:121:
lpadmin:x:122:
rtkit:x:123:
whoopsie:x:124:
sssd:x:125:
fwupd-refresh:x:126:
nm-openvpn:x:127:
scanner:x:128:saned
saned:x:129:
colord:x:130:
geoclue:x:131:
```

Step 5: Delete user

Command : `sudo deluser username`

```
root@Prasadmw:/home/vboxuser/Desktop# sudo deluser pmw1
Command 'sudp' not found, did you mean:
  command 'ssdp' from snap ssdp (0.0.1)
  command 'sup' from deb sup (20100519-3)
  command 'sudo' from deb sudo (1.9.9-1ubuntu2.4)
  command 'sudo' from deb sudo-ldap (1.9.9-1ubuntu2.4)
  command 'sfdp' from deb graphviz (2.42.2-6ubuntu0.1)
See 'snap info <snapname>' for additional versions.
root@Prasadmw:/home/vboxuser/Desktop# sudo deluser pmw1
Removing user `pmw1' ...
Warning: group `pmw1' has no more members.
Done.
root@Prasadmw:/home/vboxuser/Desktop#
```

Conclusion : In the performance of users and groups practical in Linux, we efficiently managed user accounts, assigned group permissions, and controlled access to resources, ensuring a secure multi-user environment. By using commands and modifying file permissions, we streamlined user management and collaboration. This practice enhances system security and improves the organization of resources in a shared Linux environment.

Answer the following.

1. Significance of creating user and group.

→

Creating a user and group in Linux is significant for several reasons:

1. **Access Control and Security:** Users and groups help in managing access permissions to files, directories, and system resources. By assigning specific users to groups, administrators can control who has read, write, or execute permissions.
2. **Multi-User Environment:** Linux is designed as a multi-user system, and each user should have a unique account. Groups allow users to share permissions and collaborate on files or projects efficiently.
3. **Privilege Management:** Users can be given specific roles or levels of access. For example, an admin group might have elevated privileges (like sudo access), while regular users have restricted access.
4. **System Administration:** By organizing users into groups, system administrators can apply settings, run scripts, and assign permissions collectively to a group rather than individually, improving efficiency.

Experiment No. 5

Name: Atharv Vijay Deshpande

PRN: 21410044

Batch: EN-3

.....

Title: Use of Makefile

Procedure:

1. Install make (if not already installed)

2. Create a file named Makefile (no extension) in your project directory. This file will contain rules and commands to compile and manage your project. touch Makefile

The structure of a Makefile typically includes:

- **Targets:** What you want to build (e.g., the final executable or object files).
- **Dependencies:** Files that the target depends on.
- **Commands:** Actions to execute (usually commands to compile or link code).

3. Run the Makefile

- In the terminal, navigate to the directory containing the Makefile:
- `cd /path/to/your/project`
- Run make to build the target specified in the Makefile.
- This will execute the commands in the Makefile and build your program. By default, it looks for the first target (in this case, all).

4. You can also run specific targets, for example, to clean the build files:
make clean

```
orion@vikas: ~/rtosLab/makeFileLab
GNU nano 7.2 main.cpp
#include<iostream>
#include "totalFile.h"
//#include<file1.cpp>
//#include<file2.cpp>

using namespace std;

int main(){

cout<<"This line is printing from the main file"<<endl;
printFile1();
printFile2();
return 0;
}

[ Read 15 lines ]
^G Help      ^O Write Out ^W Where Is  ^K Cut       ^T Execute   ^C Location
^X Exit      ^R Read File ^\ Replace   ^U Paste     ^J Justify   ^_ Go To Line
```

```
orion@vikas: ~/rtosLab/makeFileLab
GNU nano 7.2 file1.cpp
#include<iostream>
#include "totalFile.h"
using namespace std;

void printFile1(){

cout<<"This line is printing from the file1"<<endl;
cout<<"File 1 is also updated"<<endl;
cout<<"File1 again updated"<<endl;
}

[ Read 10 lines ]
^G Help      ^O Write Out ^W Where Is  ^K Cut       ^T Execute   ^C Location
^X Exit      ^R Read File ^\ Replace   ^U Paste     ^J Justify   ^_ Go To Line
```

```
GNU nano 7.2 file2.cpp
#include<iostream>
#include "totalFile.h"
using namespace std;

void printFile2(){

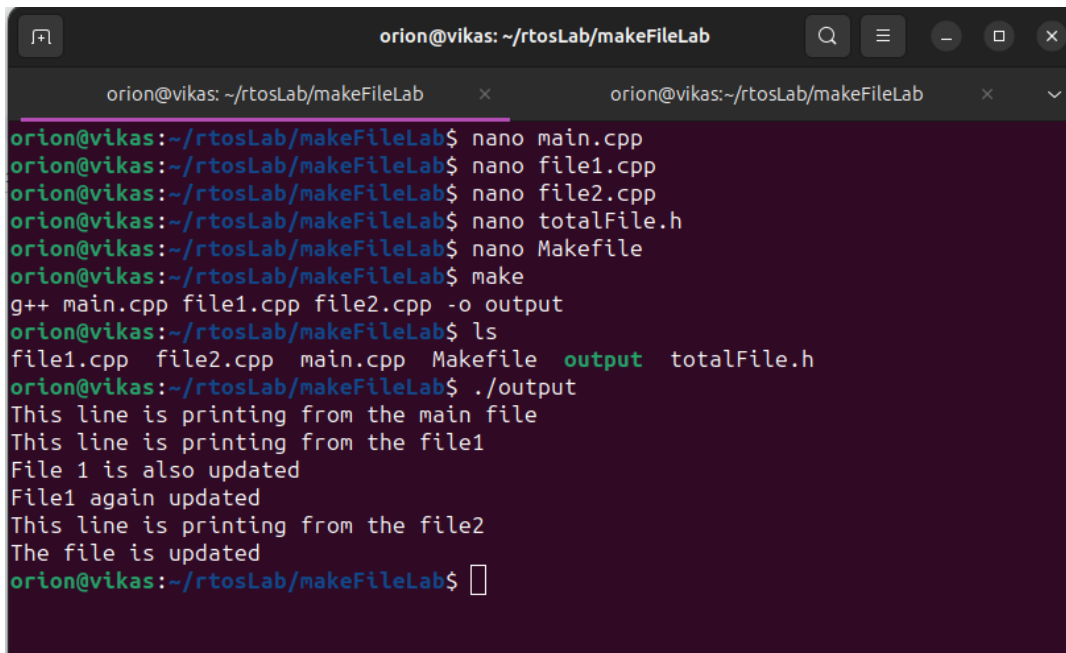
cout<<"This line is printing from the file2"<<endl;
cout<<"The file is updated"<<endl;
}
```

[Read 9 lines]

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

```
GNU nano 7.2 totalFile.h
void printFile1();
void printFile2();
```

```
GNU nano 7.2 Makefile
output:
    g++ main.cpp file1.cpp file2.cpp -o output
```


A terminal window titled 'orion@vikas: ~/rtosLab/makeFileLab' showing a series of commands and their outputs. The user creates four files (main.cpp, file1.cpp, file2.cpp, totalFile.h) using nano, then runs 'make'. The output shows the compilation of these files into 'output'. Finally, the user runs './output', which prints messages from each source file, demonstrating that the Makefile successfully built and executed the program.

```
orion@vikas:~/rtosLab/makeFileLab$ nano main.cpp
orion@vikas:~/rtosLab/makeFileLab$ nano file1.cpp
orion@vikas:~/rtosLab/makeFileLab$ nano file2.cpp
orion@vikas:~/rtosLab/makeFileLab$ nano totalFile.h
orion@vikas:~/rtosLab/makeFileLab$ nano Makefile
orion@vikas:~/rtosLab/makeFileLab$ make
g++ main.cpp file1.cpp file2.cpp -o output
orion@vikas:~/rtosLab/makeFileLab$ ls
file1.cpp file2.cpp main.cpp Makefile output totalFile.h
orion@vikas:~/rtosLab/makeFileLab$ ./output
This line is printing from the main file
This line is printing from the file1
File 1 is also updated
File1 again updated
This line is printing from the file2
The file is updated
orion@vikas:~/rtosLab/makeFileLab$
```

❖ Conclusion

The experiment of usage of makefile is performed successfully with proper screenshot as expected. Using the Makefile we can run multiple commands and achieve automation.

Answer the following:

Significance of use of makefile

- **Automation of Tasks:** A Makefile automates repetitive tasks, such as compiling, linking, and cleaning up, making it easier to build projects consistently.
- **Dependency Management:** It handles dependencies between source files, ensuring that only the changed files are recompiled, thus saving time.
- **Build Process Customization:** It allows users to customize the build process by defining different rules for compilation, linking, and cleaning, among other actions.
- **Cross-Platform Development:** Makefiles are widely used in cross-platform development, providing a uniform build process across different operating systems.

- **Simplifies Complex Projects:** For large projects with many files, a Makefile simplifies the compilation process, as all instructions are consolidated in one file.
- **Error Reduction:** By automating tasks, it reduces human errors, especially in complex, multi-step build processes.

Experiment No : 6

Name: Atharv Vijay Deshpande

PRN: 21410044

Batch: EN-3

Title : Execution of Python file in Linux

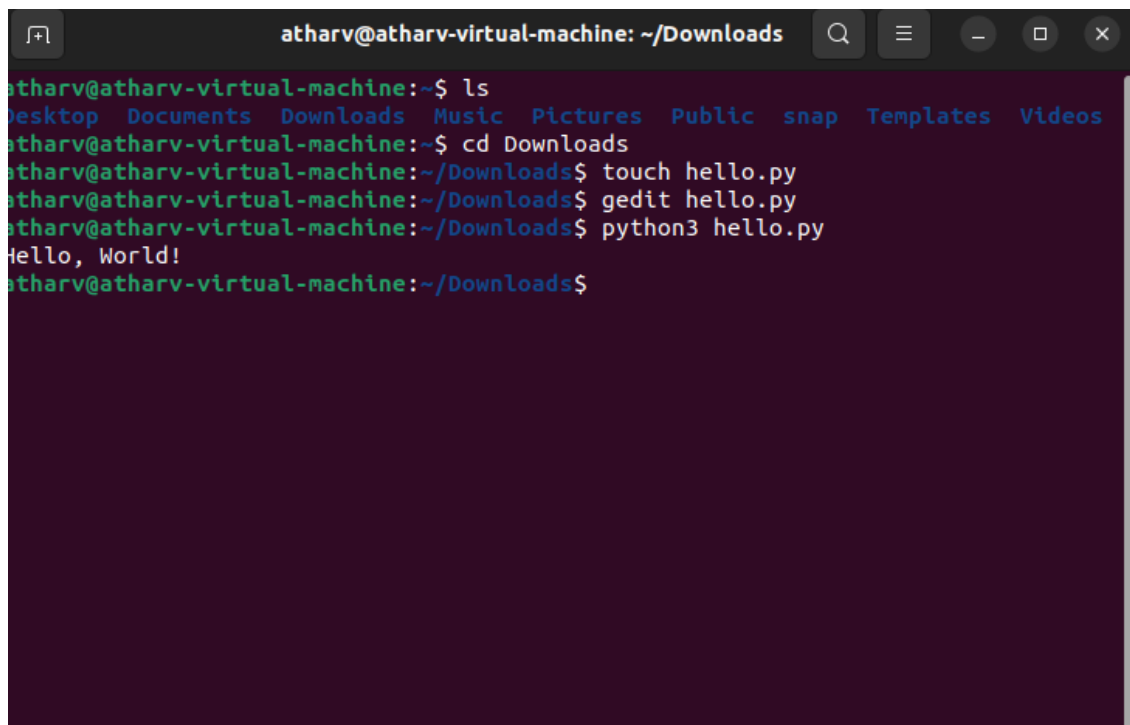
Objective: To create and execute single Python file and multiple Python files using && operator as well as using shellscript.

Procedure:

- **Procedure Steps: For Single Python File**

Steps: Create a Python file:

- Open the terminal.
- Use a text editor like nano, vim, or gedit to create a Python file.
- Use python3 command to execute the python file.



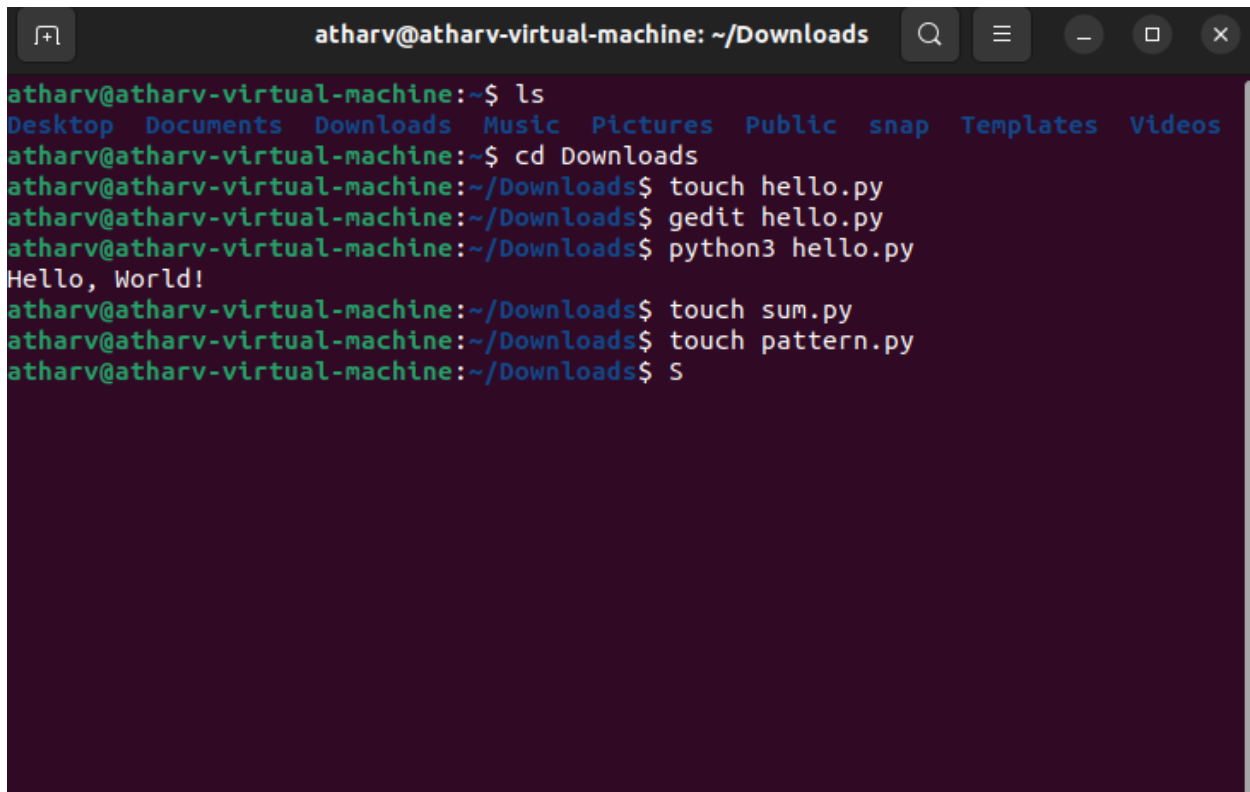
```
atharv@atharv-virtual-machine: ~/Downloads
atharv@atharv-virtual-machine:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  snap  Templates  Videos
atharv@atharv-virtual-machine:~$ cd Downloads
atharv@atharv-virtual-machine:~/Downloads$ touch hello.py
atharv@atharv-virtual-machine:~/Downloads$ gedit hello.py
atharv@atharv-virtual-machine:~/Downloads$ python3 hello.py
Hello, World!
atharv@atharv-virtual-machine:~/Downloads$
```

Procedure Steps :

- **For multiple Python Files**

Steps : Create multiple python files using touch command in terminal

- Open the terminal.
- Use touch command to create multiple python files.

A terminal window titled 'atharv@atharv-virtual-machine: ~/Downloads' with standard window controls. The terminal shows a sequence of commands: 'ls' listing directories, 'cd Downloads' changing to the Downloads directory, 'touch hello.py' creating a file, 'gedit hello.py' opening it in a text editor, and 'python3 hello.py' running it, which outputs 'Hello, World!'. Finally, 'touch sum.py' and 'touch pattern.py' are used to create more files, followed by a prompt 'S' on the next line.

```
atharv@atharv-virtual-machine:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  snap  Templates  Videos
atharv@atharv-virtual-machine:~$ cd Downloads
atharv@atharv-virtual-machine:~/Downloads$ touch hello.py
atharv@atharv-virtual-machine:~/Downloads$ gedit hello.py
atharv@atharv-virtual-machine:~/Downloads$ python3 hello.py
Hello, World!
atharv@atharv-virtual-machine:~/Downloads$ touch sum.py
atharv@atharv-virtual-machine:~/Downloads$ touch pattern.py
atharv@atharv-virtual-machine:~/Downloads$ S
```

- Use a text editor like nano, vim, or gedit to write code in a Python file.



A screenshot of a text editor window titled "sum.py" located at "~Downloads". The window has a toolbar with "Open", "Save", and window control buttons. The code is as follows:

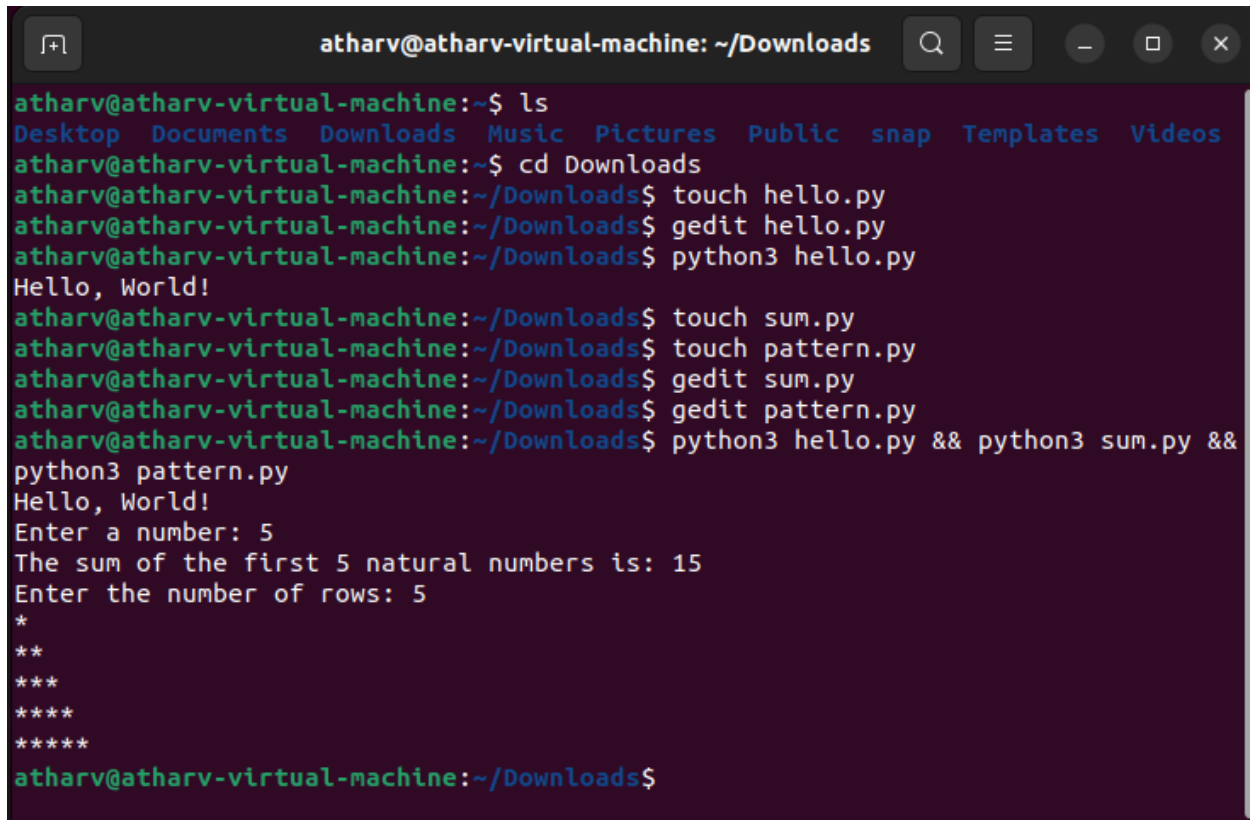
```
1 n = int(input("Enter a number: "))
2 sum_natural = n * (n + 1) // 2
3 print(f"The sum of the first {n} natural numbers is: {sum_natural}")
4
```



A screenshot of a text editor window titled "*pattern.py" located at "~Downloads". The window has a toolbar with "Open", "Save", and window control buttons. The code is as follows:

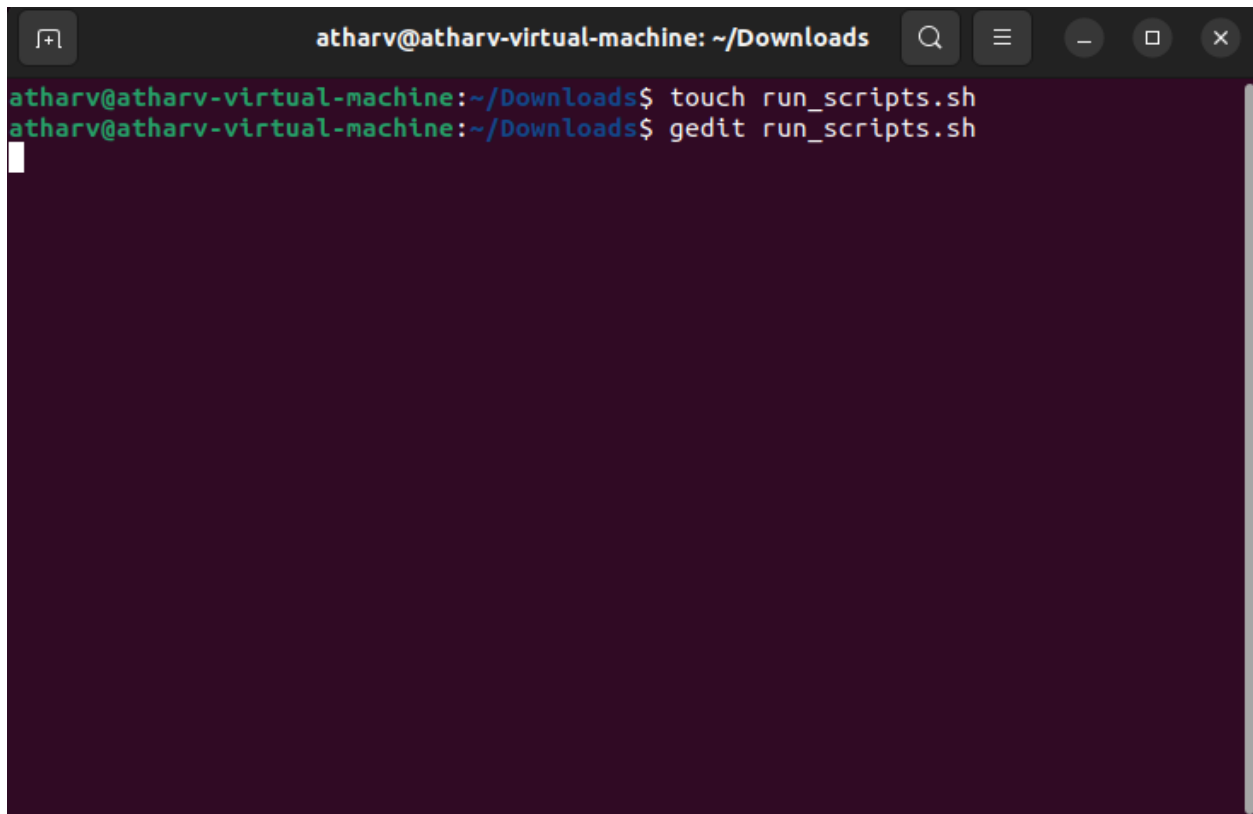
```
1 n = int(input("Enter the number of rows: "))
2 for i in range(1, n + 1):
3     print("*" * i)
4
```

- ❖ Using && operator for execution of multiple files.



```
atharv@atharv-virtual-machine: ~/Downloads
atharv@atharv-virtual-machine:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  snap  Templates  Videos
atharv@atharv-virtual-machine:~$ cd Downloads
atharv@atharv-virtual-machine:~/Downloads$ touch hello.py
atharv@atharv-virtual-machine:~/Downloads$ gedit hello.py
atharv@atharv-virtual-machine:~/Downloads$ python3 hello.py
Hello, World!
atharv@atharv-virtual-machine:~/Downloads$ touch sum.py
atharv@atharv-virtual-machine:~/Downloads$ touch pattern.py
atharv@atharv-virtual-machine:~/Downloads$ gedit sum.py
atharv@atharv-virtual-machine:~/Downloads$ gedit pattern.py
atharv@atharv-virtual-machine:~/Downloads$ python3 hello.py && python3 sum.py &&
python3 pattern.py
Hello, World!
Enter a number: 5
The sum of the first 5 natural numbers is: 15
Enter the number of rows: 5
*
**
***
****
*****
atharv@atharv-virtual-machine:~/Downloads$
```

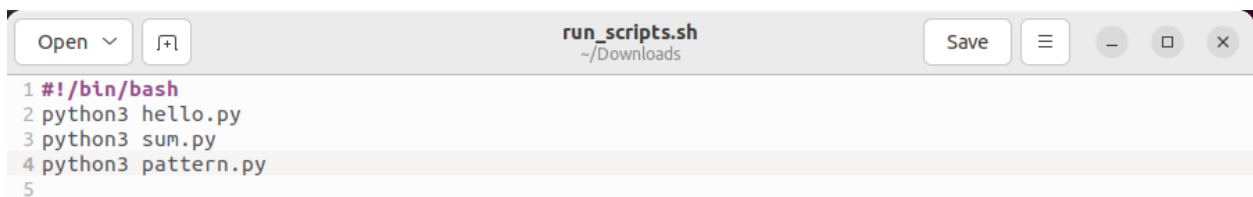
- ❖ Executing Python Files Using a Shell Script
 - Create a shell script file using touch.
 - Open the shell script file using gedit.



A terminal window titled "atharv@atharv-virtual-machine: ~/Downloads". The prompt is "atharv@atharv-virtual-machine:~/Downloads\$". The first command entered is "touch run_scripts.sh". The second command entered is "gedit run_scripts.sh". The terminal background is dark purple.

```
atharv@atharv-virtual-machine:~/Downloads$ touch run_scripts.sh
atharv@atharv-virtual-machine:~/Downloads$ gedit run_scripts.sh
```

- Write the shell script in gedit.

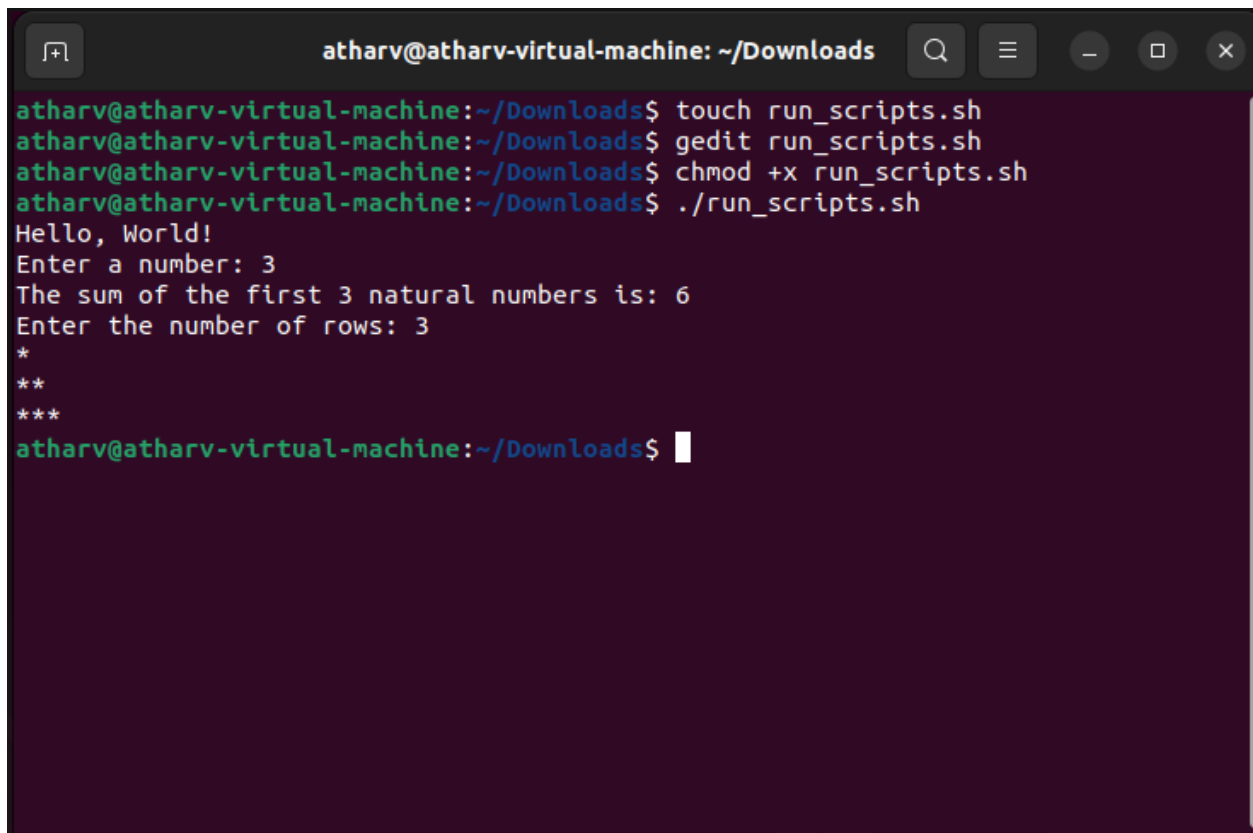


A Gedit editor window titled "run_scripts.sh" with the path "~/Downloads". The window has a light gray header bar with "Open", "Save", and window control buttons. The text area contains a shell script with four lines of code, numbered 1 to 4. Line 5 is empty.

```
1 #!/bin/bash
2 python3 hello.py
3 python3 sum.py
4 python3 pattern.py
5
```

tu Software

- Make the shell script executable using chmod command.
- Execute the shell script.

A terminal window titled 'atharv@atharv-virtual-machine: ~/Downloads' with standard window controls. The terminal shows a series of commands and their outputs. The commands are: 'touch run_scripts.sh', 'gedit run_scripts.sh', 'chmod +x run_scripts.sh', and './run_scripts.sh'. The outputs are: 'Hello, World!', 'Enter a number: 3', 'The sum of the first 3 natural numbers is: 6', and 'Enter the number of rows: 3'. This is followed by three lines of asterisks (*, **, ***) and a final prompt 'atharv@atharv-virtual-machine:~/Downloads\$' with a cursor.

```
atharv@atharv-virtual-machine:~/Downloads$ touch run_scripts.sh
atharv@atharv-virtual-machine:~/Downloads$ gedit run_scripts.sh
atharv@atharv-virtual-machine:~/Downloads$ chmod +x run_scripts.sh
atharv@atharv-virtual-machine:~/Downloads$ ./run_scripts.sh
Hello, World!
Enter a number: 3
The sum of the first 3 natural numbers is: 6
Enter the number of rows: 3
*
**
***
atharv@atharv-virtual-machine:~/Downloads$
```

Conclusion;

The experiment demonstrated the steps to create and execute Python files in a Linux environment using basic terminal commands. By leveraging tools like touch for file creation and gedit for file editing, we successfully executed both single and multiple Python files. The && operator was used to run multiple files sequentially, ensuring error-free execution of each before proceeding to the next. Additionally, the creation of a shell script provided an efficient way to automate the execution of multiple Python files. Overall, the experiment emphasized the simplicity and flexibility of managing Python files in Linux.

Answer the following:

Que: Write short note on interpreter and compiler in the context of linux.

Ans:

In Linux, **interpreters** and **compilers** are tools used to execute or convert code written in high-level programming languages into machine code that the system can understand.

1. Interpreter:

- An **interpreter** translates code **line by line** and executes it immediately. This means it reads the source code, interprets it, and runs it directly, without needing to compile the entire code first.
- Popular interpreters in Linux include Python, Bash, and Ruby.
- Since interpreters process code at runtime, errors are detected and displayed when they are encountered.
- Example: Executing a Python script with the `python3 script.py` command runs the script line by line.

2. Compiler:

- A **compiler** translates the entire source code into machine code (binary) **all at once** before execution. The output is usually an executable file that can be run on the system.
- Common Linux compilers include `gcc` (for C/C++), `javac` (for Java).
- Compilers produce a binary executable, and errors are flagged at the time of compilation, not execution.
- Example: Compiling a C program using `gcc program.c -o program` creates an executable named `program`, which can then be run.

EXPERIMENT NO.8

Name: Atharv Vijay Deshpande

PRN: 21410044

Batch: EN-3

Title: Concept of multitasking, virtual parallelism using RTOS.

Objective:

1. Understanding the concept of multitasking.
2. Understanding the concept of virtual parallelism.
3. Installation of μ Cos-II based application.
4. Significance of all folders and files.
5. Creating of 3 tasks and observing waveforms using logic analyzer.

Multitasking:

Multitasking, as the name suggests, is a technique used to handle the execution of multiple tasks. It can be defined as, “the execution of multiple software routines in pseudo-parallel. Each routine represents a separate thread of execution. The OS simulates parallelism by dividing the CPU processing time to each individual thread. The second term to define is real-time. A real-time system is a system based on stringent time constraints. For a system to be real-time, there must be a guaranteed time frame where within a task must execute. In a multitasking RTOS the task scheduling, switching and execution elements are key to the effectiveness of the OS in achieving its real-time requirements.

Concept of virtual parallelism:

1. Virtual parallelism is a technique used in RTOS to emulate the behaviour of running multiple tasks in parallel, even when the underlying hardware is single core. The idea is to create the illusion that several tasks are executing simultaneously, despite the limitation of the hardware.
2. Virtual parallelism is a concept related to parallel computing, which involves the simultaneous execution of multiple tasks or processes to improve computational efficiency and speed. Virtual parallelism, however, does not necessarily involve physical parallel processing units like multiple processors or cores. Instead, it is a technique that makes it appear as if parallelism exists, even when the underlying hardware might not support true parallel execution.
3. In above example we observed that, in first waveform it looks like each task execute at the same time, but after zooming in we can observe significant difference between execution of first task, second task, third task. It is practical example of virtual parallelism.
4. Virtual parallelism is particularly useful in scenarios where hardware constraints limit the degree of parallelism that can be achieved. For example, on a single core processor, multiple tasks can be scheduled in such a way that they share processing time, giving the appearance of parallelism. This approach can improve overall system responsiveness

and throughput, especially in applications where tasks may spend a significant amount of time waiting for external resources (I/O operations, network requests, etc.).

5. Virtual parallelism can be achieved by task decomposition, task scheduling, concurrency, resource management, and synchronization. In task decomposition, first task or problem is divided into smaller, independent task or threads, and then executed. Task scheduling is required at the time of single processor, a scheduler or task manager is used to interleave the execution of these sub-tasks. In concurrency, tasks are not truly executing in parallel on separate processing units, they are being executed concurrently, meaning that multiple tasks are taking turns running in a way that creates the impression of parallelism. Virtual parallelism often involves managing shared resources, like memory and input/output operations, to ensure that tasks do not interfere with each other and cause conflicts. To maintain data consistency and avoid race conditions, synchronization mechanisms are used to coordinate the execution of tasks.

Installation:

Installation of uCOS-II based application requires IDE i.e., Integrated Development Environment, and for that here we use Keil uVision 4.

Step 1: Download the setup of Keil from below link, after downloading extract this file

<https://www.keil.com/demo/eval/armv4.htm>

Step 2: Install the mdk410 setup file , then click on Next, then Next, then agree to terms and condition

Step 3: Then do further process, and fill information in customer information. After completing this , click on Next, it will start to setup and display status about it.

Step 4: After this finish the setup

Significance of folder:

1) ARM:

In this folder all device dependent files are present as we are using ARM processor. These files include processor and implementation specific constants, these files are in c and assembly language.

In Os_cpu_c.c folder contain many files related to interfaces like lcd, led, uart, config file, relay etc. which includes program of each interface and which includes basic functions.

2) μ COS-II:

In this folder all device independent files are present, which needs to include as system can be of any type or of any system, so for compatibility it must be included in project.

It consists of various .c files with respect to real time operating system.

It includes file like OS_core.c, OS_FLAG.c, OS_SEM.c , OS_TASK.c , OS_TIME.c and many more, which includes functions related to OS.

3) MD2148:

It includes a board specific folder which contains function related to interfacing board.

Like lcd.c , led.c, uart.c etc.

4) Main:

It includes folder for application programmer. This folder contains all application programs.

Program for 3 tasks of LED Blinking with different delay:

Code:

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

#define TaskStkLengh 64 //Define the Task0 stack length

OS_STK TaskStk0 [TaskStkLengh]; //Define the Task stack
OS_STK TaskStk1 [TaskStkLengh]; //Define the Task stack
OS_STK TaskStk2 [TaskStkLengh]; //Define the Task stack

void Task0(void *pdata);
void Task1(void *pdata);
void Task2(void *pdata);

// Required for sending time to serial
port char buffer[25];

int main (void)
{
    LED_init();

    TargetInit();
    OSInit ();

    OSTaskCreate (Task0,(void *)0, &TaskStk0[TaskStkLengh - 1], 6);
    OSTaskCreate (Task1,(void *)0, &TaskStk1[TaskStkLengh - 1], 7);
    OSTaskCreate (Task2,(void *)0, &TaskStk2[TaskStkLengh - 1], 8);
    OSStart();
```

```

        return 0;

    }

void Task0 (void *pdata)
{
    pdata = pdata;                                /* Dummy data */

    while(1)
    {

        LED_on(0);
        OSTimeDly(10);
        LED_off(0);
        OSTimeDly(10);

    }
}

void Task1 (void *pdata)
{
    pdata = pdata;                                /* Dummy data */

    while(1)
    {
        LED_on(1);
        OSTimeDly(20);
        LED_off(1);
        OSTimeDly(20);

    }
}

void Task2 (void *pdata)
{
    pdata = pdata;                                /* Dummy data */

    while(1)
    {
        LED_on(2);
        OSTimeDly(30);
        LED_off(2);
        OSTimeDly(30);

    }
}

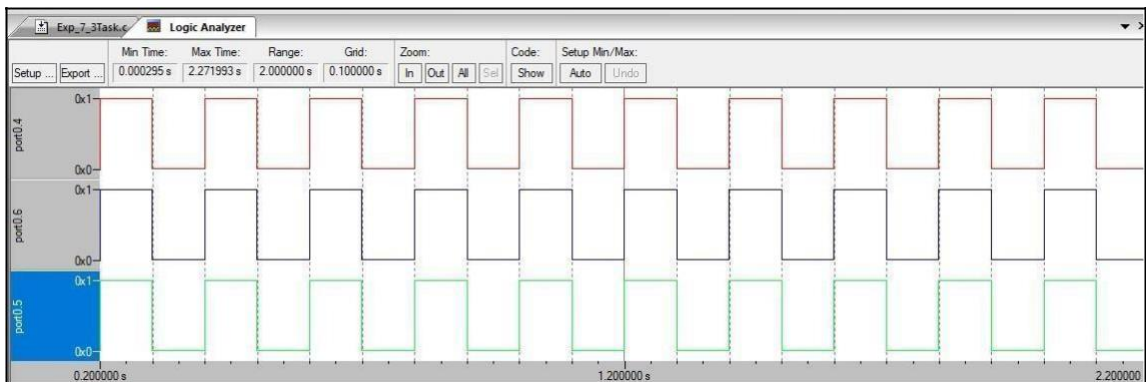
```

Observations:

1) Different time delay:



2) Same time delay:



If we zoom in the above waveforms, it can be seen that all tasks are not carried out at same time



Here we can see difference between two waveform is in microsec. It looks like waveforms are executing at the same time, but it is not. So, it can be understood using concept of virtual parallelism.

Conclusion:

- 1) From the output of the code, we can see that multiple tasks are running at the same time. Hence it is a multitasking.
- 2) Virtual parallelism means, to create an illusion that several tasks are running simultaneously.

EXPERIMENT NO. 9

Name: Atharv Vijay Deshpande

PRN: 21410044

Batch: EN-3

Title: Program to illustrate need of semaphore.

Objective-

1. Understand the need of semaphore and its functionality.
2. Understand the concept of priority inversion.

What is semaphore?

A semaphore is a protocol mechanism offered by most multitasking kernels. Semaphores are used to:

- a) Control the access of the shared resources (mutual exclusion);
- b) Allow two tasks to synchronize the activities.
- c) Signal the occurrence of an event.

A semaphore is a key that your code acquires in order to continue execution. If semaphore is already in use, the requesting task is suspended until the semaphore is released by its current owner.

There are two types of semaphores:

- a) Binary Semaphore: The binary semaphores are like counting semaphores but their value is restricted to 0 and 1.
- b) Counting Semaphore: These are integer value semaphores and can be any non-negative integer.

1. Code without Semaphore:

```
#include "config.h"

#include "stdlib.h"

#include <stdio.h>


#define TaskStkLengh 64           //Define Task stack length
OS_STK TaskStk0 [TaskStkLengh];  //Define the Task0 stack
OS_STK TaskStk1 [TaskStkLengh];  //Define the Task1 stack

// declare the tasks
```



```

void Task0(void *pdata);

void Task1(void *pdata);

OS_EVENT *MySem;

unsigned char err;


int main (void)
{
    LED_init();
    UART0_Init();
    TargetInit();
    OSInit ();
    MySem = OSSemCreate(1);
    OSTaskCreate (Task0,(void *)0, &TaskStk0[TaskStkLengh - 1], 6);
    OSTaskCreate (Task1,(void *)0, &TaskStk1[TaskStkLengh - 1], 7);
    OSStart();
    return 0;
}

void Task0 (void *pdata)
{
    pdata = pdata;
    while(1)
    {
        //Uncomment pend and post to see semaphores in action
        //OSSemPend(MySem, 0, &err);
        UART0_SendData ("*****In Task 0\n");
        OSTimeDly(4);
        UART0_SendData ("***** Task 0 Completed\r");
        //OSSemPost(MySem);
    }
}

```

```

void Task1 (void *pdata)
{
    pdata = pdata;
    while (1)
    {
        //Uncomment pend and post to see semaphores in action
        //OSSemPend(MySem, 0, &err);

        OSTimeDly(4);

        UART0_SendData ("%%%% In Task 1...\n");

        OSTimeDly(4);

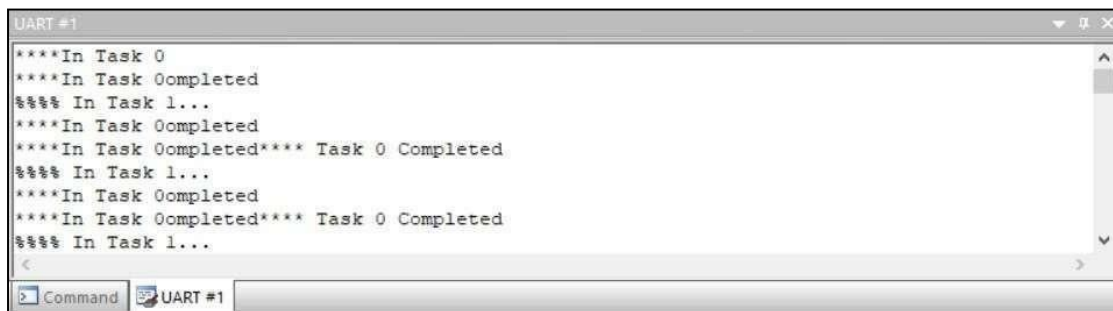
        UART0_SendData ("%%%% Task 1 Completed");

        //OSSemPost(MySem);

    }
}

```

Output:



2. Code with Semaphore:

```

#include "config.h"

#include "stdlib.h"

#define TaskStkLengh64      //Define the Task0 stack length

OS_STK      TaskStk0 [TaskStkLengh];      //Define the Task0 stack
OS_STK      TaskStk1 [TaskStkLengh];      //Define the Task0 stack

```

```

void Task0(void *pdata);           // Task0
void Task1(void *pdata);           // Task1

OS_EVENT *MySem;
unsigned char err;

int main (void)
{
    LED_init();
    lcd_init();
    UART0_Init();
    UART0_SendData("\n\r*****\n\r");
    UART0_SendData ("*Program for semaphore demo*\n\r");
    UART0_SendData ("*****\n\r");

    TargetInit();
    OSInit ();
    MySem = OSSemCreate(1);
    OSTaskCreate (Task0,(void *)0, &TaskStk0[TaskStkLengh - 1], 6);
    OSTaskCreate (Task1,(void *)0, &TaskStk1[TaskStkLengh - 1], 7);

    OSStart();
    return 0;
}

```

```

void Task0    (void *pdata)
{
    int i;

    pdata = pdata;                                /* Dummy data */
    while(1)
    {
        UART0_SendData("\n\rTask 0 waiting for Semaphore");
        lcd_command(0x80);
        LCD_SendData("          ");
        lcd_command(0x80);
        LCD_SendData("Task 0 waiting");
        OSSemPend(MySem, 0, &err);
        UART0_SendData("\n\rTask 0 got the Semaphore");
        UART0_SendData("\n\rTask 0 now flashing LED-0 for 15 times");

        lcd_command(0x80);
        LCD_SendData("          ");
        lcd_command(0x80);
        LCD_SendData("Task 0 got sem");

        for(i=0; i<15;i++)
        {
            LED_on(0);
            OSTimeDly(10);
            LED_off(0);
            OSTimeDly(10);
        }

        UART0_SendData("\n\rTask 0 released Semaphore \n\r");
    }
}

```

```

        lcd_command(0x80);
        LCD_SendData("      ");
        lcd_command(0x80);
        LCD_SendData("Task 0 released");

        OSSemPost(MySem);
        //OSTimeDly(150);
    }
}

void Task1 (void *pdata)
{
    int i;
    pdata = pdata;                                /* Dummy data */

    while (1)
    {
        UART0_SendData("\n\rTask 1 waiting for Semaphore");

        lcd_command(0xC0);
        LCD_SendData("      ");
        lcd_command(0xC0);
        LCD_SendData("Task 1 waiting");

        OSSemPend(MySem, 0, &err);
        UART0_SendData("\n\rTask 1 got the Semaphore");
        UART0_SendData("\n\rTask 1 now flashing LED-1 for 10 times");

        lcd_command(0xC0);
        LCD_SendData("      ");

```

```

        lcd_command(0xC0);

        LCD_SendData("Task 1 got sem");

        for(i=0; i<15;i++)
        {

            LED_on(1);

            OSTimeDly(10);

            LED_off(1);

            OSTimeDly(10);

        }

        UART0_SendData("\n\rTask 1 released Semaphore\n\r");

        lcd_command(0xC0);

        LCD_SendData("          ");

        lcd_command(0xC0);

        LCD_SendData("Task 1 released");

        OSSemPost(MySem);

    }

}

```

Output:



The screenshot shows a UART terminal window titled "UART #1". The output text is as follows:

```

*****
*Program for semaphore demo*
*****
Task 0 waiting for Semaphore
Task 0 got the Semaphore
Task 0 now flashing LED-0 for 15 times
Task 1 waiting for Semaphore

```

The terminal window has a "Command" input field and a "UART #1" label at the bottom.

Priority Inversion:

Priority Inversion is an operating system scenario in which a higher priority process is preempted by a lower priority process. This implies the inversion of priorities of the two tasks.

It is not possible to totally avoid it. But its effect can be minimized by:

- a. Priority ceiling
- b. Disabling interrupts
- c. Priority inheritance
- d. No blocking
- e. Random boosting

Conclusion:

- When we don't use semaphore in the code the output will be not in an order. That means any other function will start running before the completion of current function.
- When we use semaphore, it allows tasks to synchronise their activity.