



Department of Computer Science
University of Engineering & Technology, Lahore
New Campus

Lab Manual

Operating System Lab

Semester: Spring-2023

Session: 2021

Instructor: Waqas Ali

Contents

Lab No 1	3
Ubuntu Installation and Setting up Virtual Machine.	3
Lab No 2	23
Overview of Ubuntu Directories and Basic Shell Commands	23
Lab No 3	39
Compilation and Makefile	39
Lab No 4	48
Process Creation and Execution (fork() & exec())	48
Lab No 5	57
Inter Process Communication (PIPES)	57
Lab No 6	62
POSIX THREADS	62
Lab No 7	79
Producer-Consumer problem	79
Lab No 8	85
BANKERS ALGORITHM	85
Lab No 9	89
Scheduling	89
Lab No 10	98
Scheduling-II	98
Lab No 11	108
Paging	108
Lab No 12	116
Memory Management Technique	116
Lab No 13	124
File Organization	124

Operating System

Lab No 1

Ubuntu Installation and Setting up Virtual Machine.

CLO: 1

TOPIC TO BE COVERED

- Introduction to Virtual Box
- Installation of Virtual Box
- Setting up the Virtual Machine
- Installing Ubuntu

Objectives

- Student get familiar with Virtual box
- Student able to install Ubuntu in his/her laptop/PC in Virtual Box

Virtual Box – An Introduction

Virtual Box is a powerful x86 and AMD64/Intel64 virtualization product for enterprise as well as home use. Not only is Virtual Box an extremely feature rich, high performance product for enterprise customers, it is also the only professional solution that is freely available as Open Source Software under the terms of the GNU General Public License (GPL) version 2.

When we describe Virtual Box as a "virtualization" product, we refer to "full virtualization", that is, the particular kind of virtualization that allows an unmodified operating system with all of its installed software to run in a special environment, on top of your existing operating system. This environment, called a "virtual machine", is created by the virtualization software by intercepting access to certain hardware components and certain features. The physical computer is then usually called the "host", while the virtual machine is often called a "guest". Most of the guest code runs unmodified, directly on the host computer, and the guest operating system "thinks" it's running on real machine. Learn more at <https://www.virtualbox.org/>

Now we will start Installing Virtual Box in Windows, to install Ubuntu as virtual machine

Installing Virtual Box

You can download Virtual Box from following link.

VirtualBox Download

VirtualBox binaries
By downloading, you agree to the terms and conditions of the respective license.
If you're looking for the VirtualBox 5.1.34 packages, see [VirtualBox 5.1 builds](#). Consider upgrading.

- **VirtualBox 5.2.8 platform packages.** The binaries are released under the terms of the GPL version 2.
 - [Windows hosts](#)
 - [OS X hosts](#)
 - [Linux distributions](#)
 - [Solaris hosts](#)
- **VirtualBox 5.2.8 Oracle VM VirtualBox Extension Pack** [All supported platforms](#)
Support for USB 2.0 and USB 3.0 devices, VirtualBox RDP, disk encryption, NVMe and PXE boot for Intel cards. See [this chapter](#) from the User Manual for an introduction to this Extension Pack.
The Extension Pack binaries are released under the [VirtualBox Personal Use and Evaluation License \(PUEL\)](#).
Please install the extension pack with the same version as your installed version of VirtualBox:
- **VirtualBox 5.2.8 Software Developer Kit (SDK)** [All platforms](#)

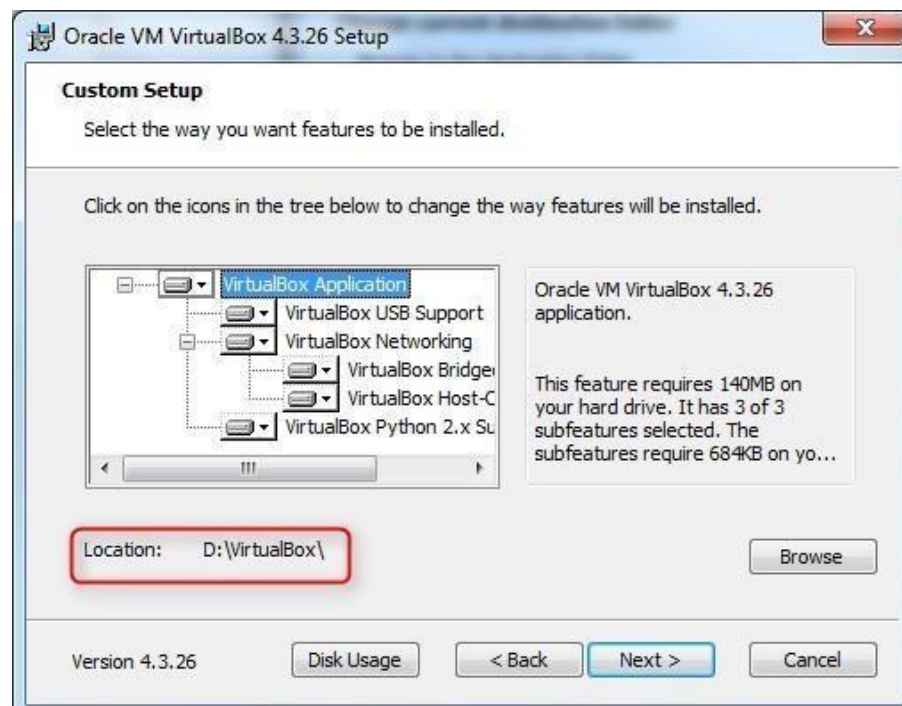
And click on Windows hosts. After completing the download click on exe for starting the installation.

Step1: Following Welcome screen will appear.



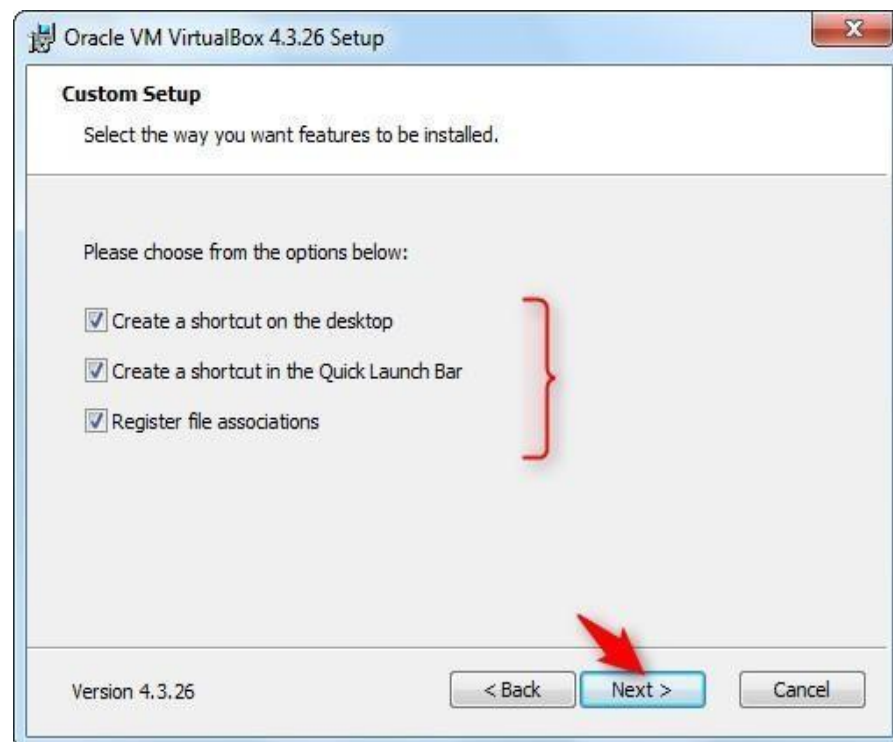
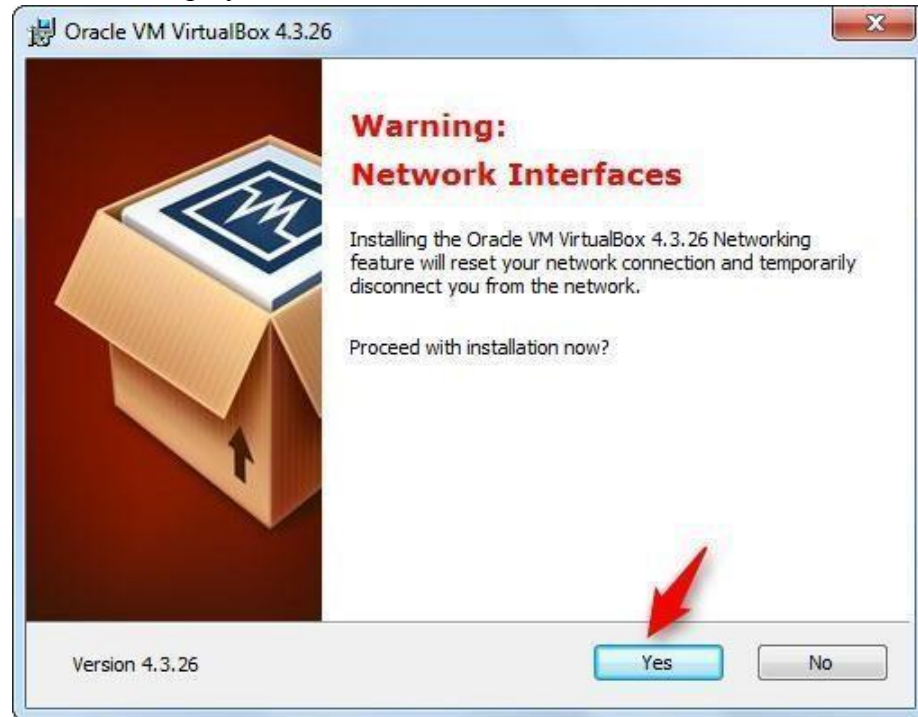
Click “Next” button.

Step2: Click Next

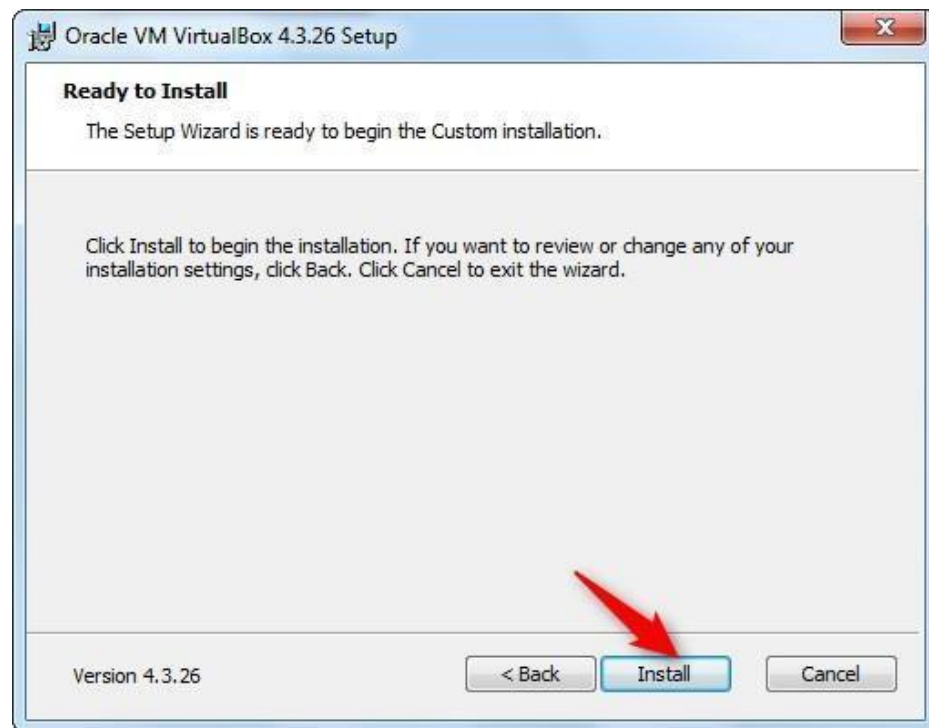


Step3: Custom Setup for different features Window is displayed. No need to change anything, just press “Next” button.

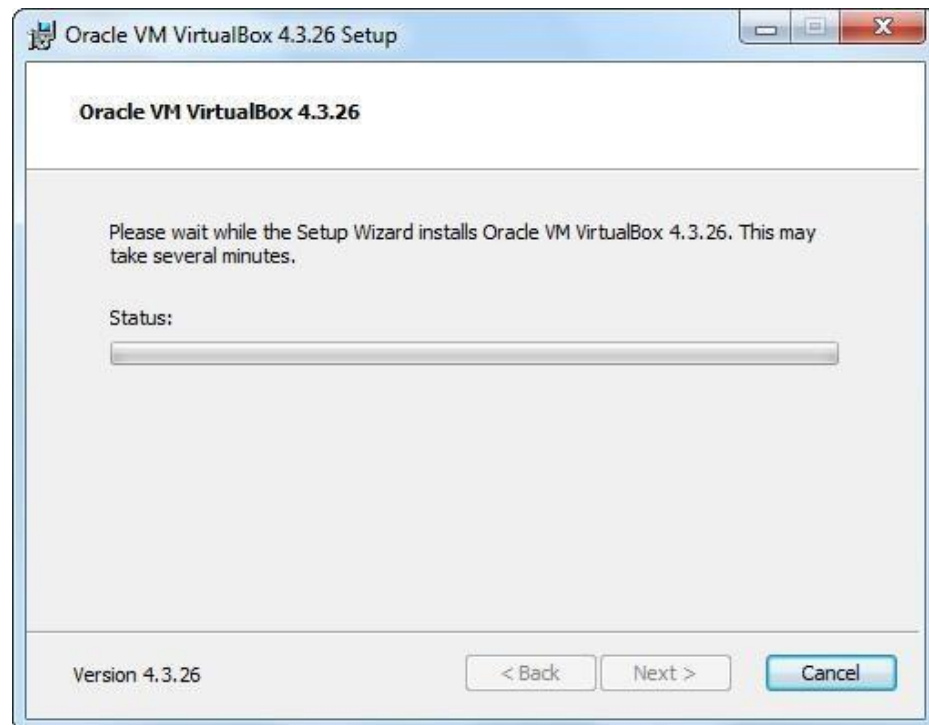
Step 4: Warning window is displayed. Click Yes.



Step5: “Ready to Install” window is displayed. Press “Install” button.

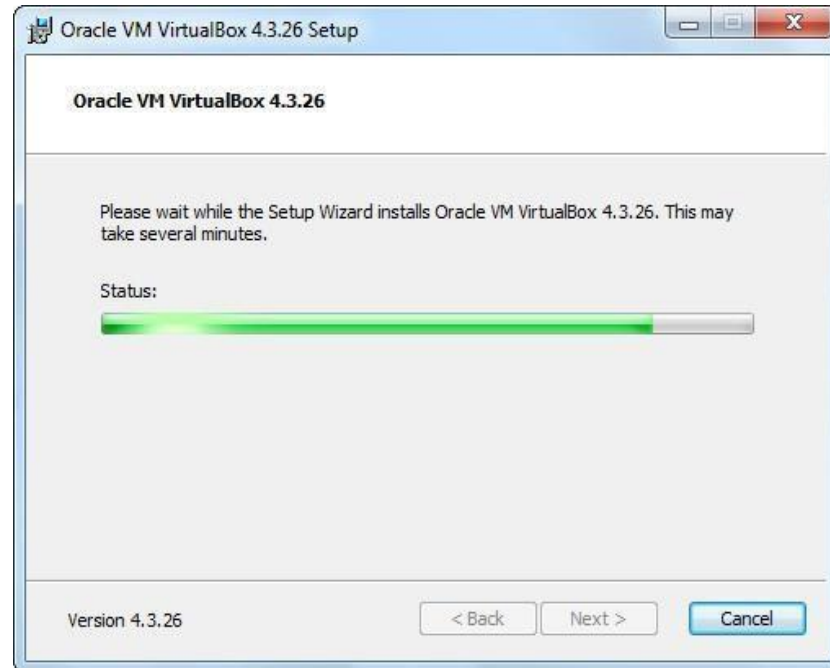


Step6: Finally after setup, installation will start and following window will be displayed



with status bar.

Installation continues...



Meanwhile Windows Security will ask for the confirmation of installing hardware. e.g below.

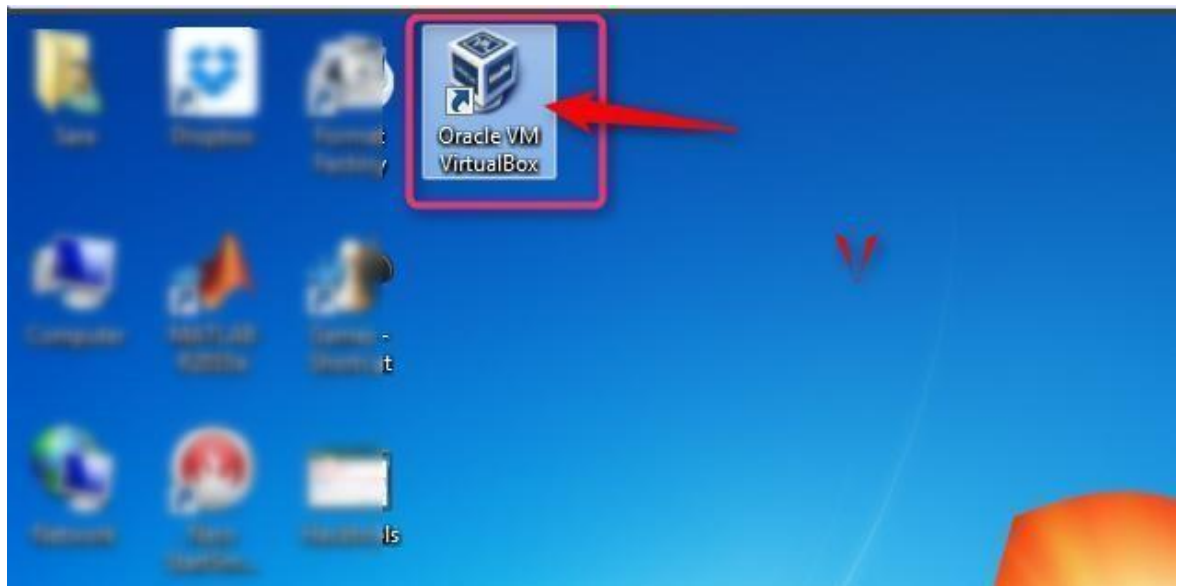


Press Install.

Step7: Installation Complete. Press “Finish” button.

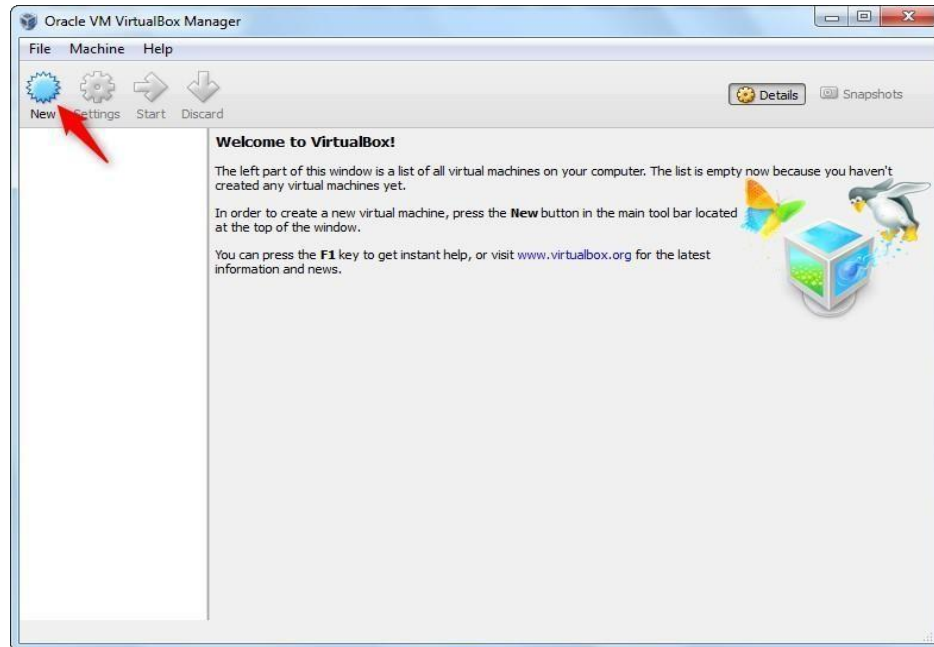


After installation completed, you can find its icon on desktop.



Setting up Virtual Machine

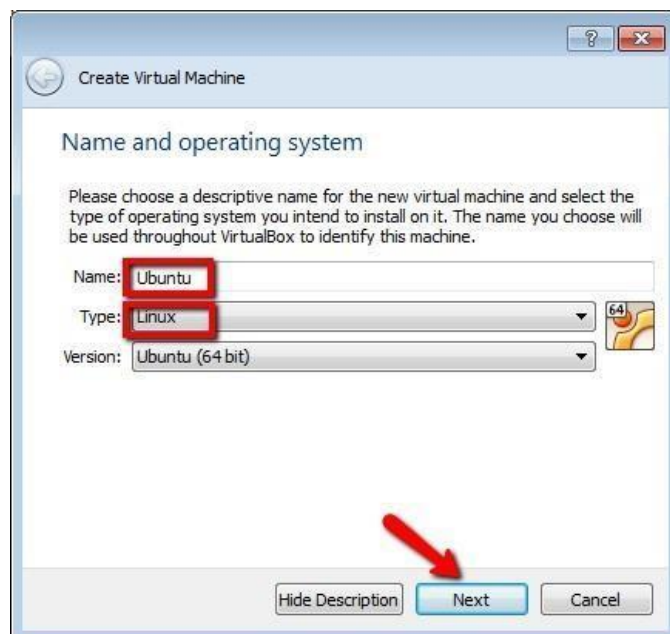
Step1: Click the Virtual Box icon and following window will be opened.



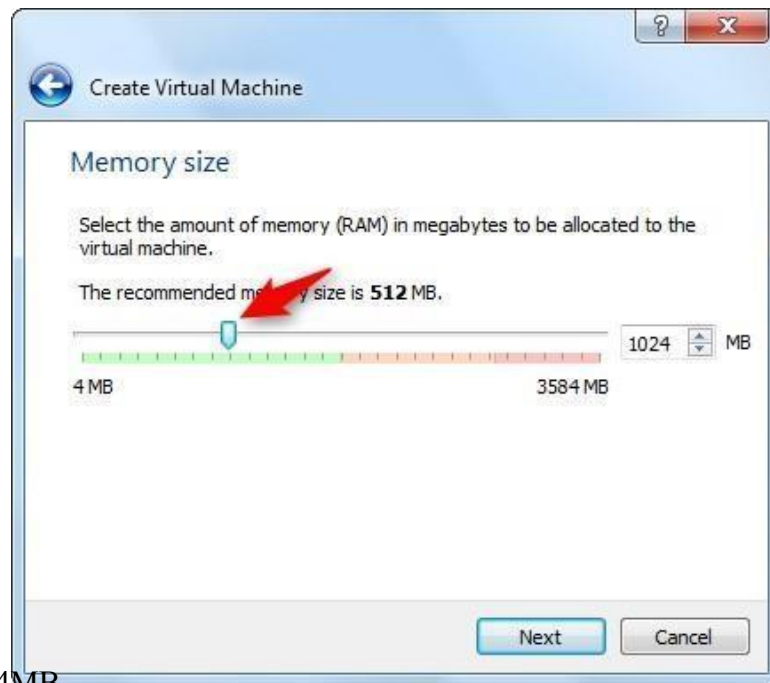
Click on “New” button on top of menu bar.

Step2: Step1 will lead you to setup a new virtual machine.

Following window will be opened to setup virtual machine name and type. Press Next.



Step3: By pressing “Next”, following window will be opened to setup memory size.



Minimum size is 1024MB

Press Next.

Step4: By pressing “Next”, “Hard drive” window will be opened. Select “Create a virtual hard drive now”.



Press “Create” button.

Step 5: By default, VDI option is selected. Do not make any change and just press

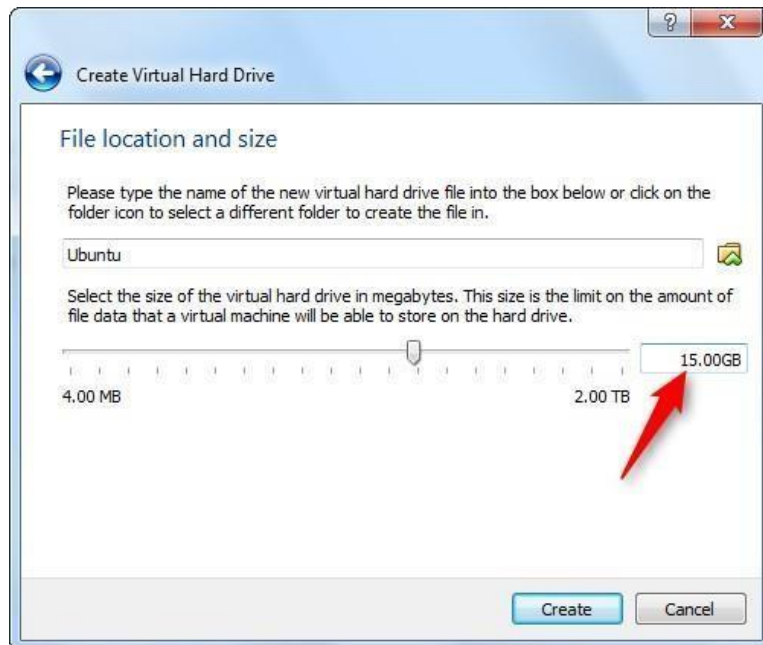


“Next”.

Step6: Select “Dynamically allocated”.

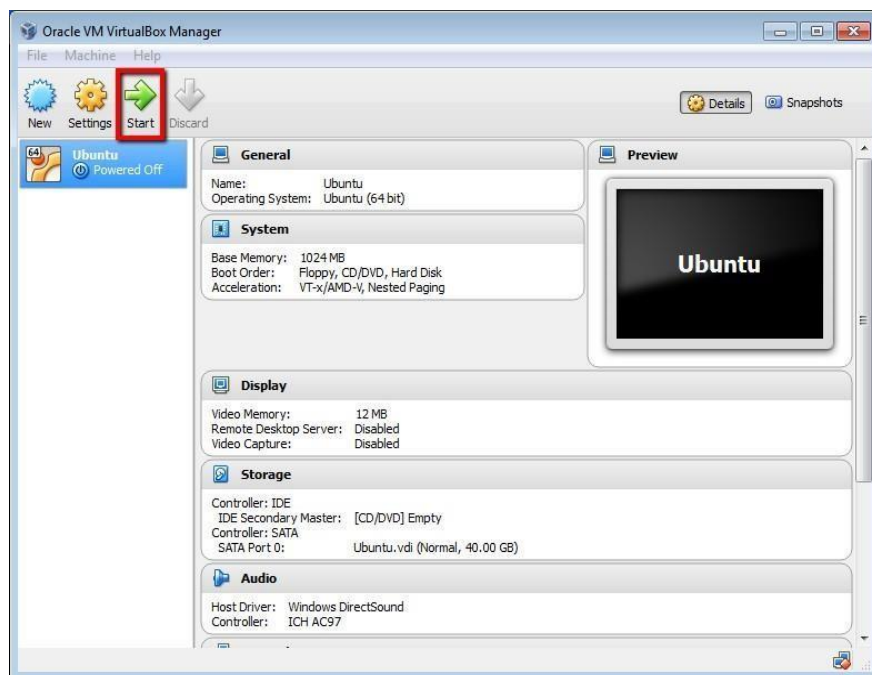


Step7: Following window will be opened to setup virtual hard drive size. Here 15 GB is selected but you must select it greater than 100GB if free space is available.



Press “Create” button.

Step8: Following window will appear with one virtual machine named “Ubuntu” will be added in left panel. Click on “Start” button highlighted.



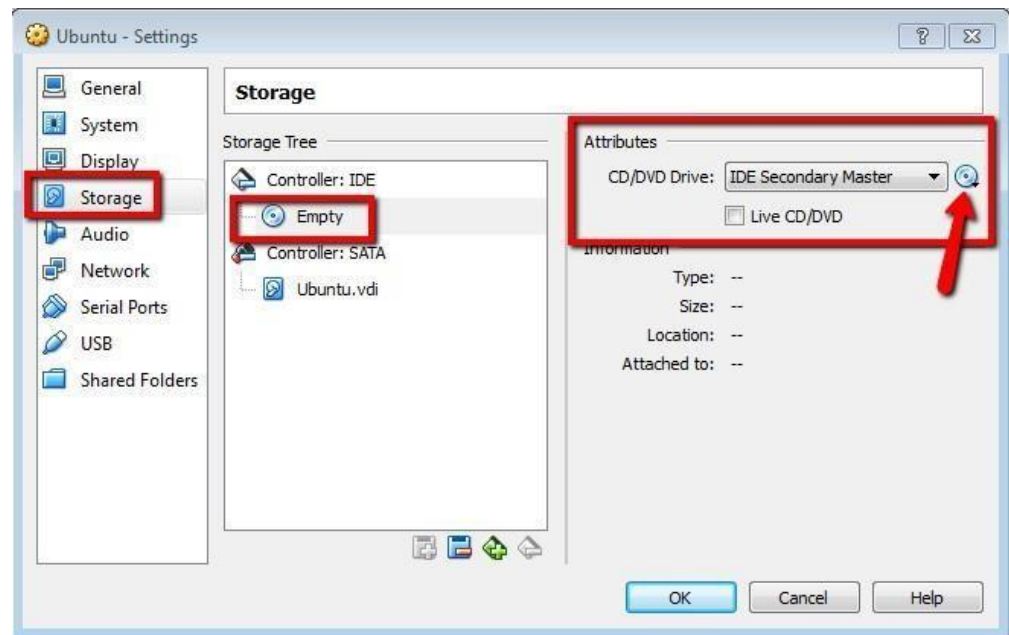
Step9: In order to attach ISO image with this virtual machine.

9.1 Click on “Settings” button. Following settings window will open.

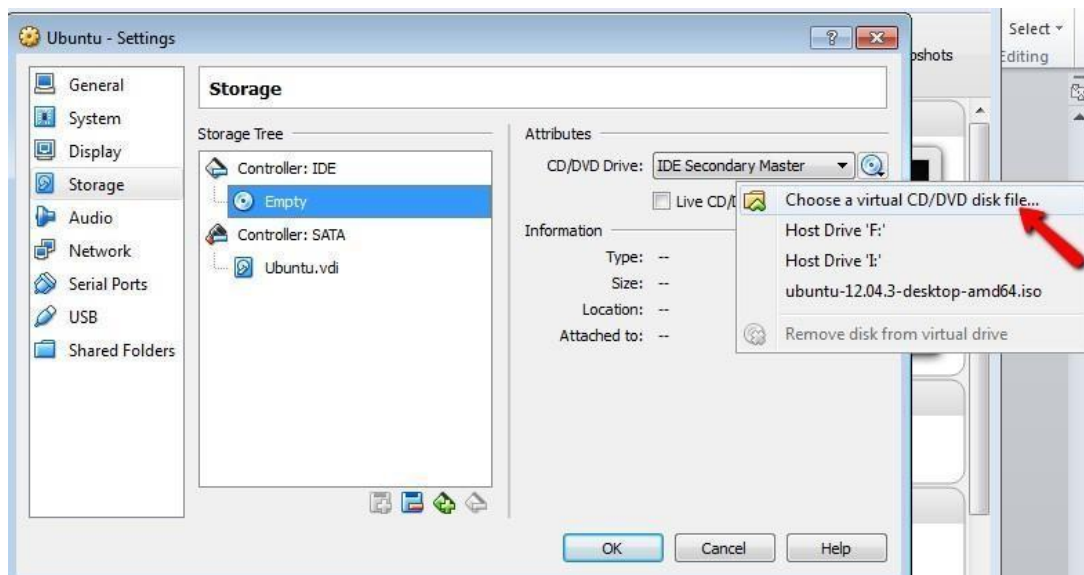
9.2 Go to Storage Option. Under “Storage Tree” go to Controller: IDE.

9.3 Select “Empty” option. By selecting “Empty” Attributes will be changed.

9.4 Click on CD icon pointed by arrow in below image.

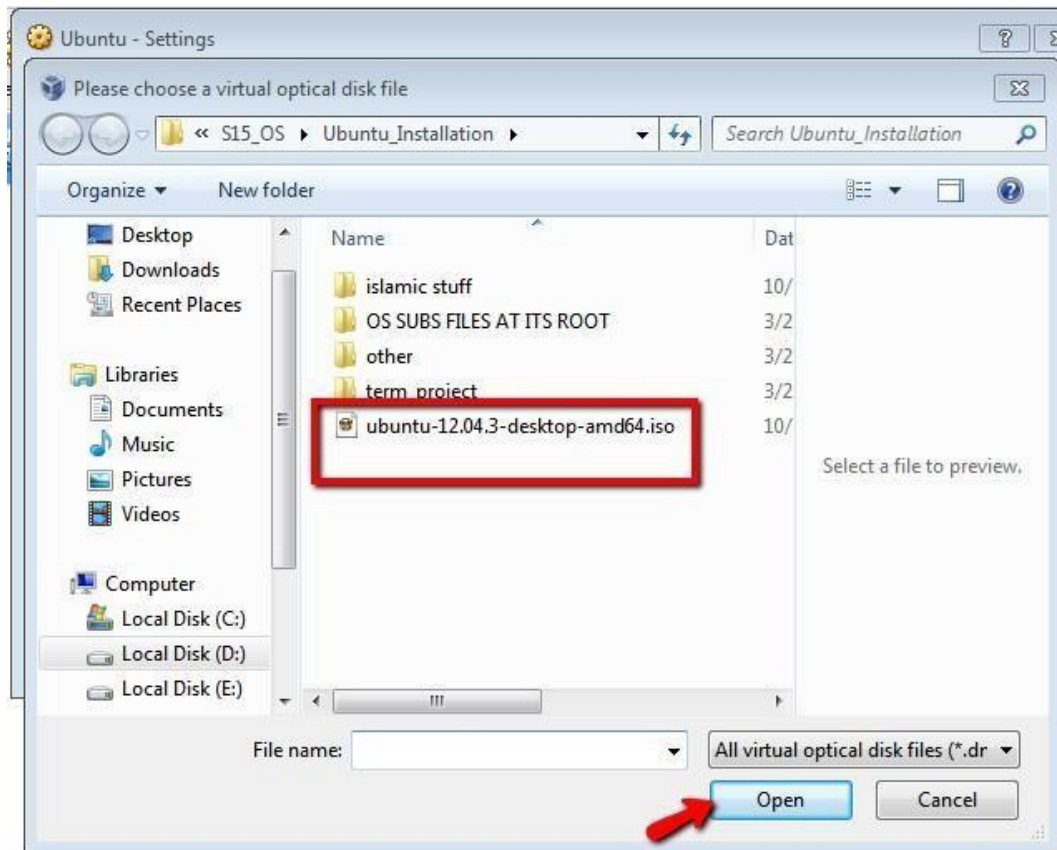


9.5 By clicking at icon of CD, following menu will be displayed.

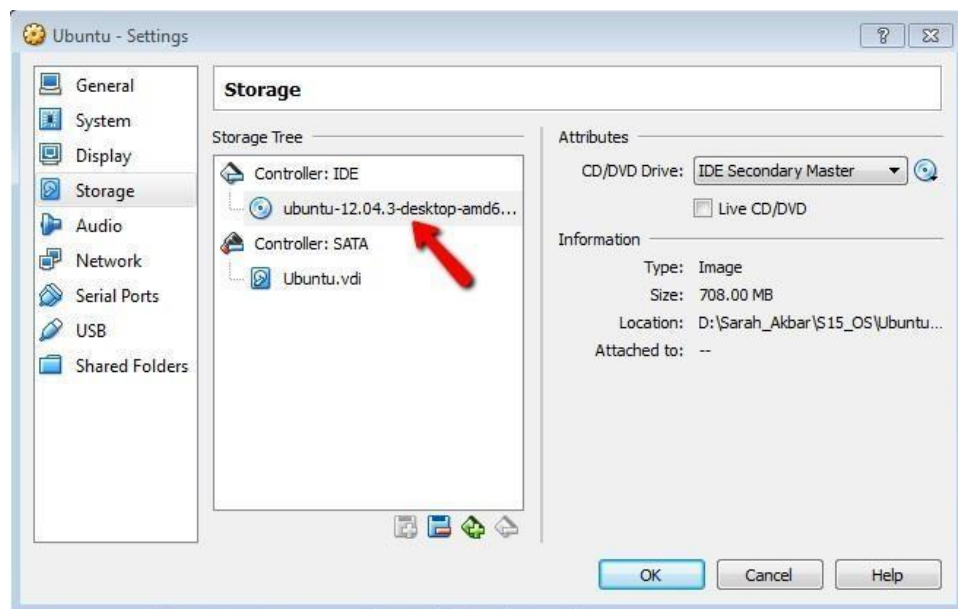


9.6 By clicking on “Choose a virtual CD/DVD disk file... An “Open Window” will appear.

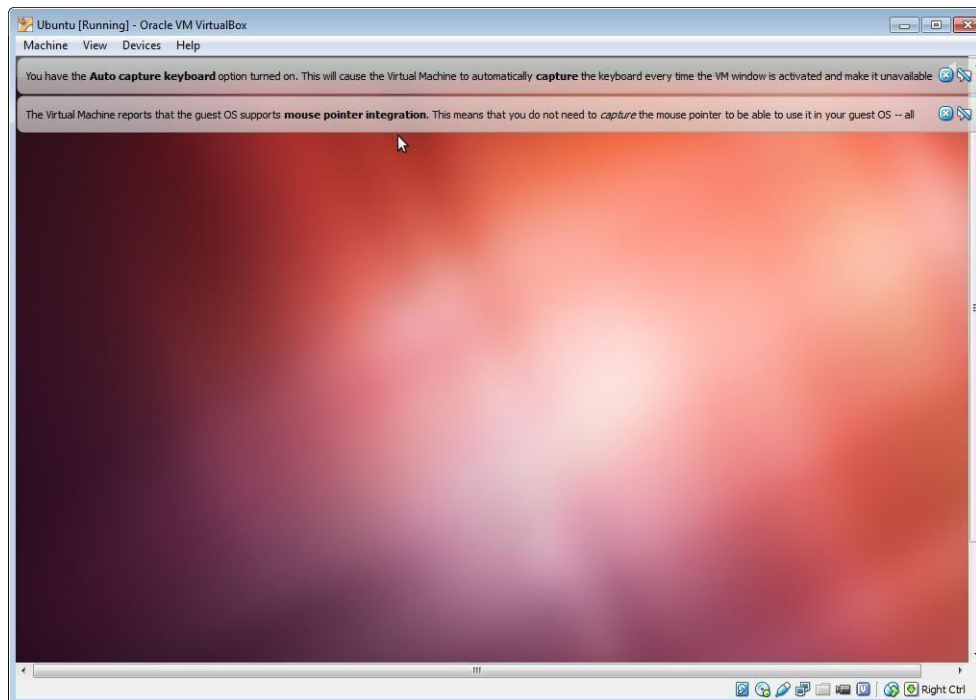
Select .iso(Downloaded From Ubuntu website) image and click “Open” button.



9.7 Now it's added here. Press “OK” button.



Step 10: Click “Start” button and following window will appear.

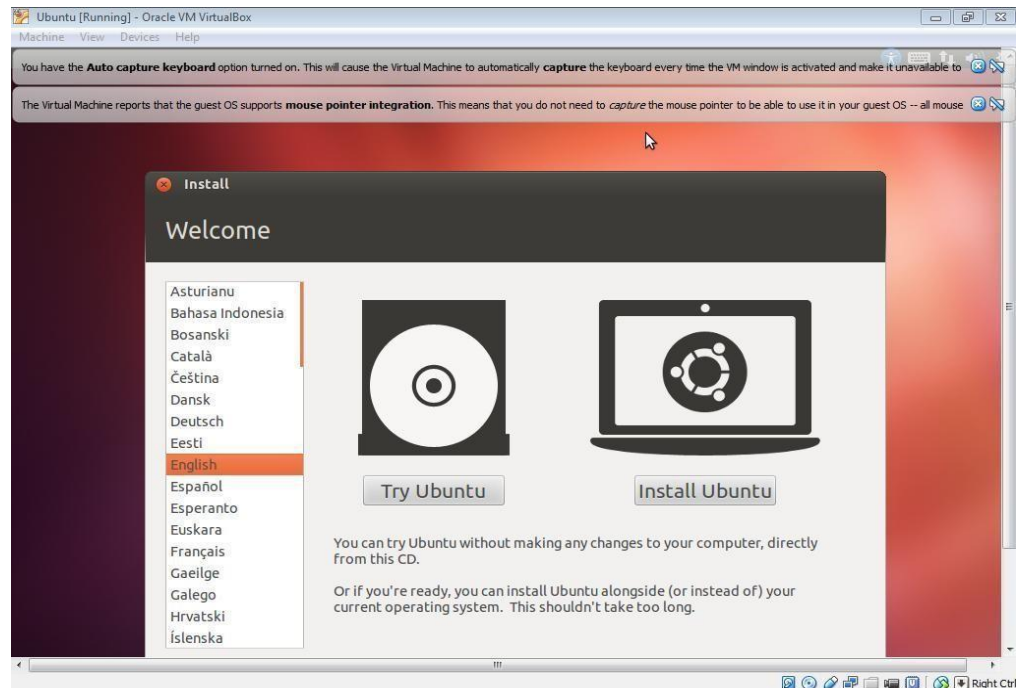


Installing Ubuntu

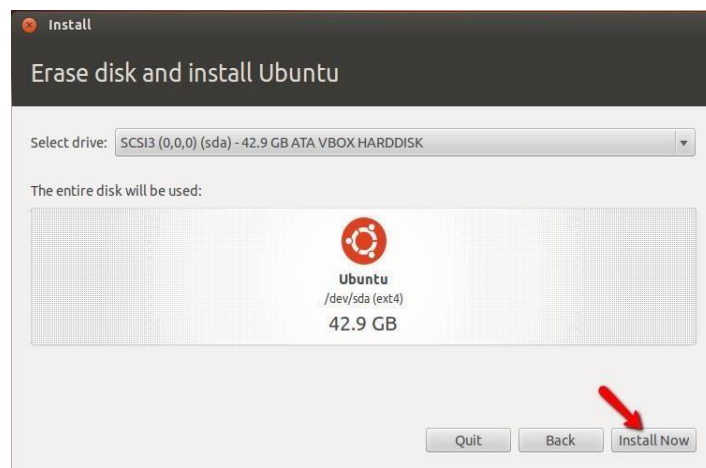
Step1: Now you will start installing Ubuntu from ISO image just added in virtual machine named “Ubuntu”

When following screen will appear

Step2: Select “Install Ubuntu” button.



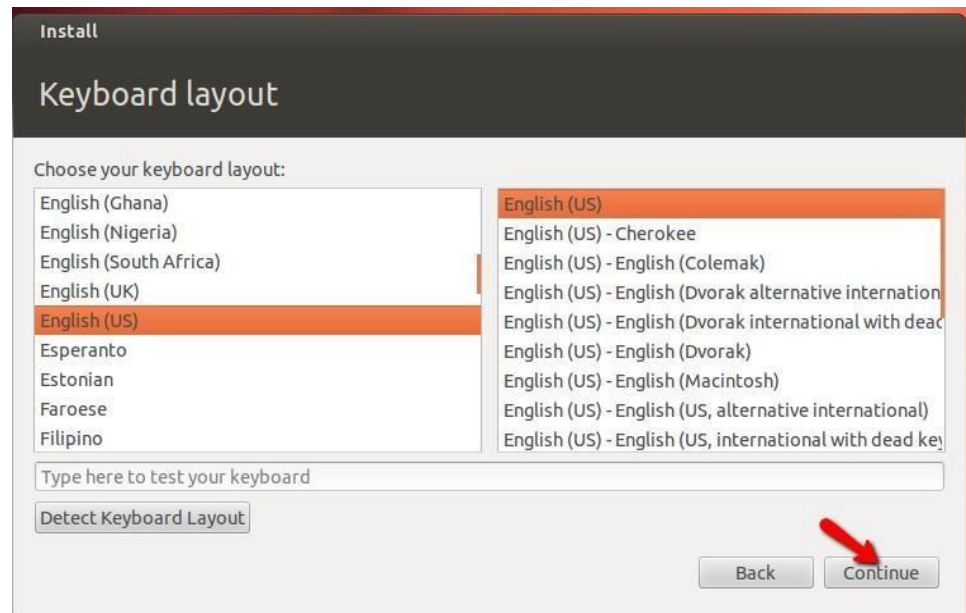
Step3: Click “Install now”



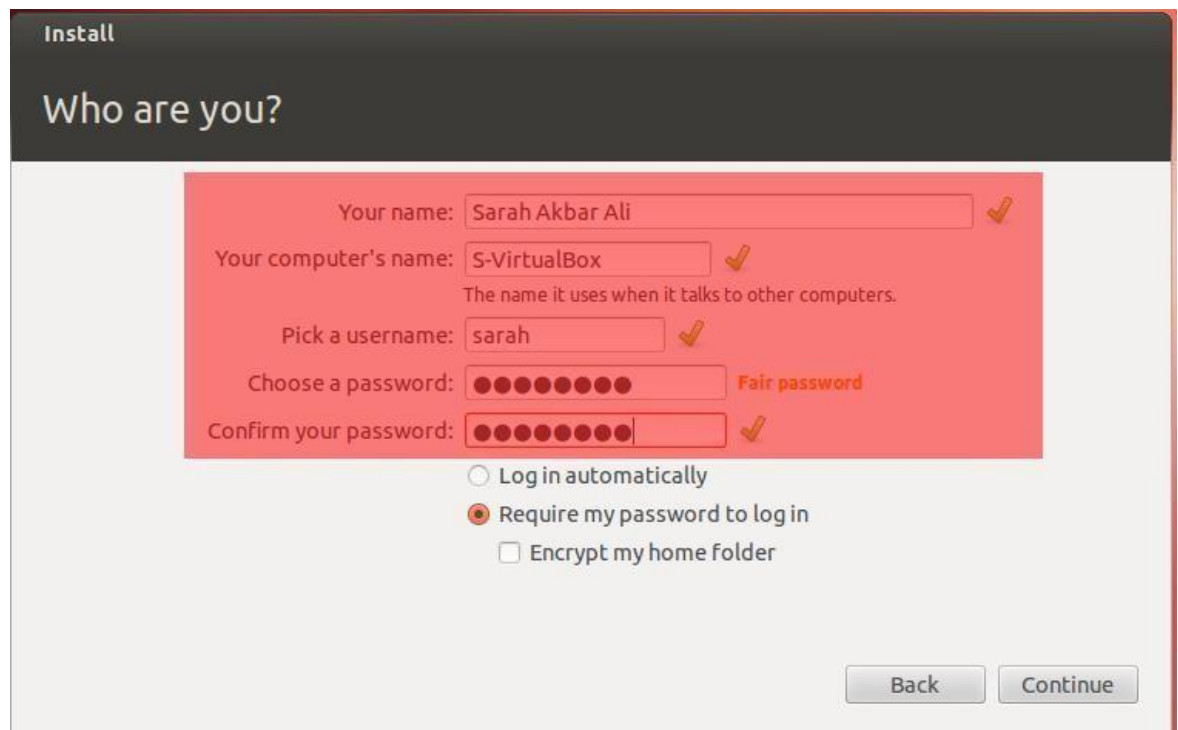
Step4:



Step5: Choose Language



Step6: Give username and password information.



Step7: Following welcome window will appear

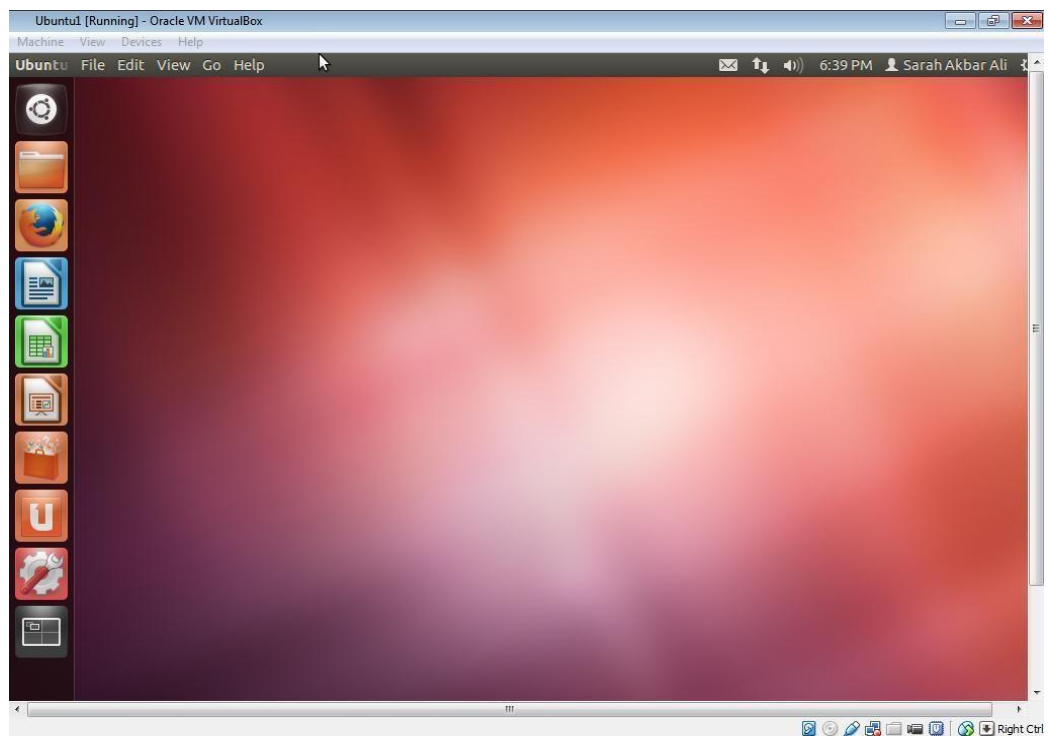


Step8: Restart system.



Step 9: Press “Enter”. Login into Ubuntu

Step10: Desktop of Ubuntu displayed.





Storage link to iso

Operating System

Lab No 2

Overview of Ubuntu Directories and Basic Shell Commands

CLO: 1

Rubrics for Lab:

Task	0	1	2	3
As mentioned in the Task-1	Student could not create the file or even don't know what to do	First task achieved.	Two tasks achieved	All the Tasks achieved and presented properly

Topic to be covered

- Overview of Ubuntu Directories
- Basic Shell Command on Linux for working on Terminal
 - a. Helping Manual in Ubuntu using command
 - b. Listing Files
 - c. Listing Hidden Files
 - d. Creating & Viewing Files

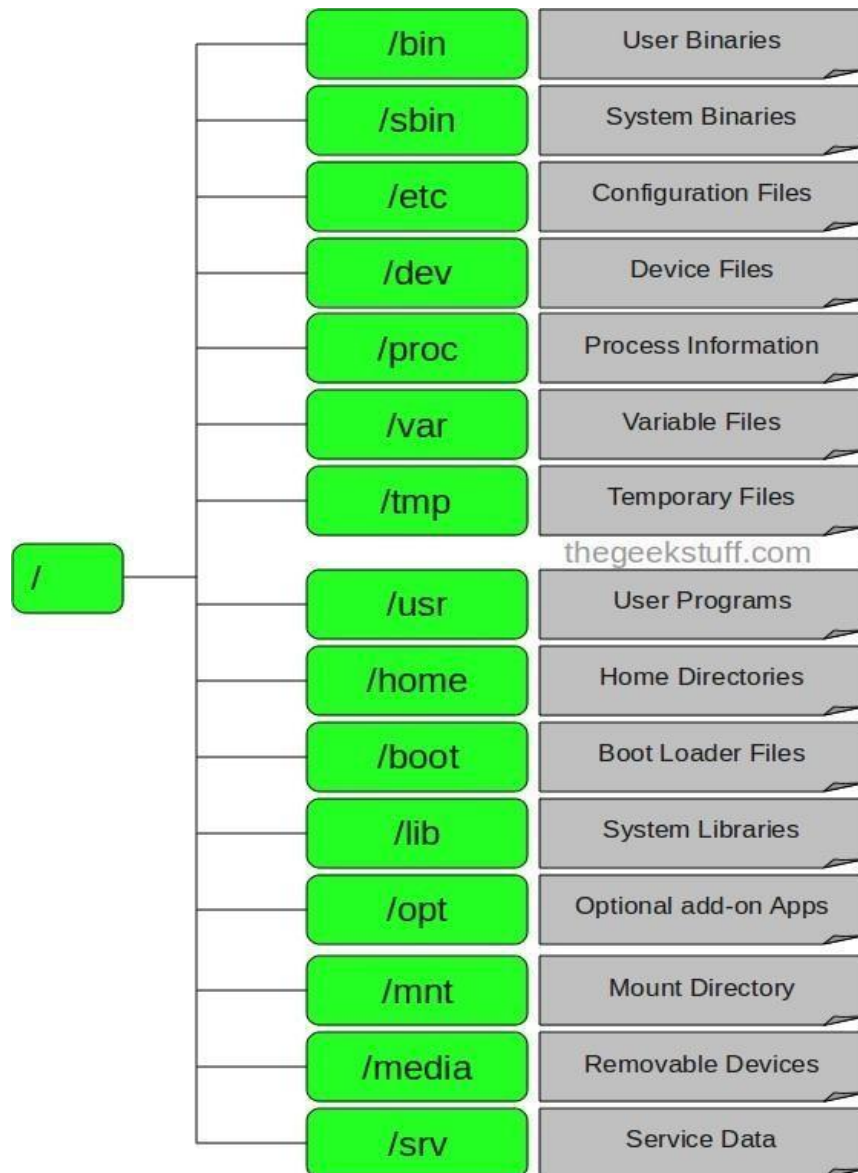
- e. Combining files
- f. Page wise content view
- g. Deleting Files
- h. Moving & Re-naming Files
- i. Copying Files
- j. Searching Contents of a file
 - i. Simple Search using less
 - ii. grep
- Directory Manipulations
 - a. Removing Directories
 - b. Renaming Directories
- Other Useful Command
 - a. Clear
 - b. History
 - c. Sudo

Objectives

- Students are able to understand Linux directories
- Students are able to understand and use different Linux Shell commands.

Overview of Ubuntu Directories

We will learn the Linux file system structures and understand the meaning of individual high-level directories.



1. / – Root

Every single file and directory starts from the root directory. Only root user has write privilege under this directory. Please note that **/root** is root user's home directory, which is not same as **/**.

2. /bin – User Binaries

Contains binary executables. Common Linux commands you need to use in single-user modes are located under this directory. Commands used by all the users of the system are located here.

For example: ps, ls, ping, grep, and cp.

3. /sbin – System Binaries

Just like **/bin**, **/sbin** also contain binary executables. But, the Linux commands located under this directory are used typically by system administrator, for system maintenance purpose.

For example: iptables, reboot, fdisk, ifconfig, swapon

4. /etc – Configuration Files

Contains configuration files required by all programs. This also contains startup and shutdown shell scripts used to start/stop individual programs.

For example: /etc/resolv.conf, /etc/logrotate.conf

5. /dev – Device Files

Contains device files. These include terminal devices, usb, or any device attached to the system.

For example: /dev/tty1, /dev/usbmon0

6. /proc – Process Information

Contains information about system process. This is a pseudo filesystem contains information about running process. For example: **/proc/ {PID}** directory contains information about the process with that particular PID. This is a virtual filesystem with text information about system resources.

For example: /proc/uptime

7. /var – Variable Files var stands for variable files.

Content of the files that are expected to grow can be found under this directory

For example: log files (**/var/log**); packages and database files (**/var/lib**); emails (**/var/mail**); print queues (**/var/spool**); **lock files** (**/var/lock**); temp files needed across reboots (**/var/tmp**);

8. /tmp – Temporary Files

Directory that contains temporary files created by system and users. Files under this directory are deleted when system is rebooted.

9. /usr – User Programs

Contains binaries, libraries, documentation, and source-code for second level programs. **/usr/bin** contains binary files for user programs. If you can't find a user binary under **/bin**, look under **/usr/bin**.

For example: at, awk, cc, less, scp

/usr/sbin contains binary files for system administrators. If you can't find a system binary under **/sbin**, look under **/usr/sbin**. For example: atd, cron, sshd, useradd, userdel **/usr/lib** contains libraries for **/usr/bin** and **/usr/sbin**

/usr/local contains users programs that you install from source. For example, when you install apache from source, it goes under **/usr/local/apache2**

10. /home – Home Directories

Home directories for all users to store their personal files.

For example: **/home/Sherjeel**, **/home/Waqar**

11. /boot – Boot Loader Files

Contains boot loader related files. Kernel **initrd**, **vmlinux**, **grub** files are located under **/boot**

For example: **initrd.img-2.6.32-24-generic**, **vmlinux-2.6.32-24-generic**

12. /lib – System Libraries

Contains library files that supports the binaries located under /bin and /sbin. Library filenames are either ld* or lib*.so.*

For example: ld-2.11.1.so, libncurses.so.5.7

13. /opt – Optional add-on Applications opt stands for optional.

Contains add-on applications from individual vendors. Add-on applications should be installed under either /opt/ or /opt/ subdirectory.

14. /mnt – Mount Directory

Temporary mount directory where sysadmins can mount filesystems.

15. /media – Removable Media Devices

Temporary mount directory for removable devices.

For examples, /media/cdrom for CD-ROM; /media/floppy for floppy drives; /media/cdrecorder for CD writer.

16. /srv – Service Data srv stands for service.

Contains server specific services related data.

For example, /srv/cvs contains CVS (Concurrent version system) related data.

Source: <http://www.thegeekstuff.com/2010/09/linux-file-system-structure/>

Basic Command on Linux for working on Terminal

Launch a terminal by pressing `ctrl + t` or press windows and type terminal

a. Helping Manual in Ubuntu using command

The 'Man' command

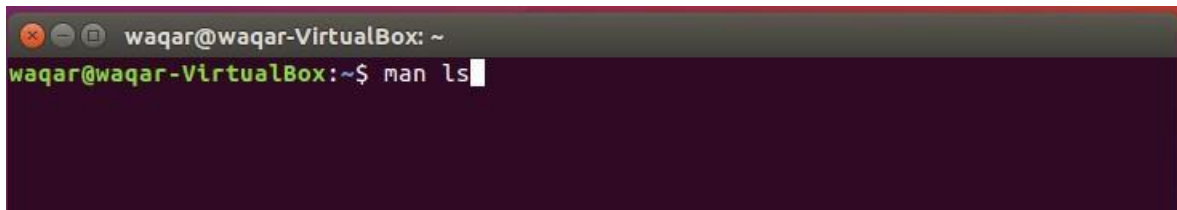
Man stands for manual which is a reference book of a Linux operating system. It is similar to HELP file found in popular software.

To get help on any command that you do not understand, you can type

man

The terminal would open the manual page for that command.

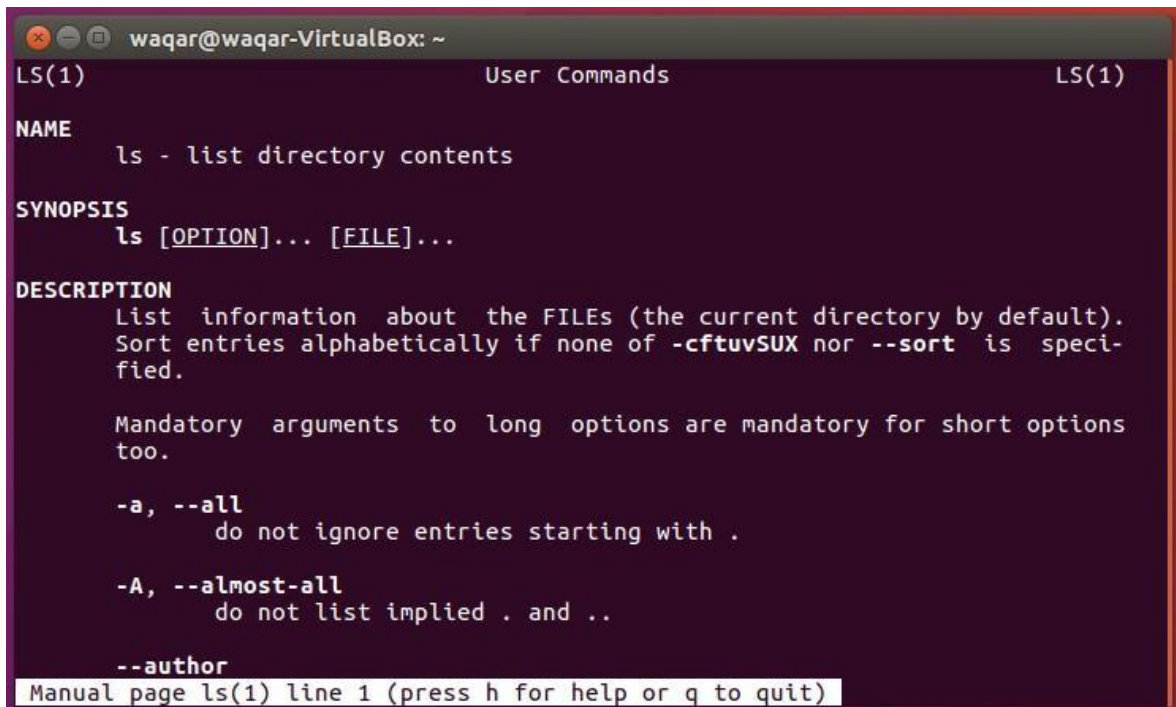
For an example, if we type *man ls* and hit enter; terminal would give us information on **ls**

A screenshot of a terminal window. The title bar at the top reads 'waqar@waqar-VirtualBox: ~'. The terminal text shows the prompt 'waqar@waqar-VirtualBox:~\$' followed by the command 'man ls' and a cursor. The background of the terminal is dark purple.

command.

Figure 1: man command for ls

When you type this command and hit Enter you will get the detailed manual for **ls** command as show in figure below.



```
waqar@waqar-VirtualBox: ~
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 speci-
    fied.

    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

Manual page ls(1) line 1 (press h for help or q to quit)
```

b. Listing Files

When you first login, your current working directory is your home directory. Your home directory has the same name as your user-name, for example, **Waqar**, and it is where your

Figure 2: Detail manual for ls command

personal files and subdirectories are saved.

To find out what is in your home directory, type

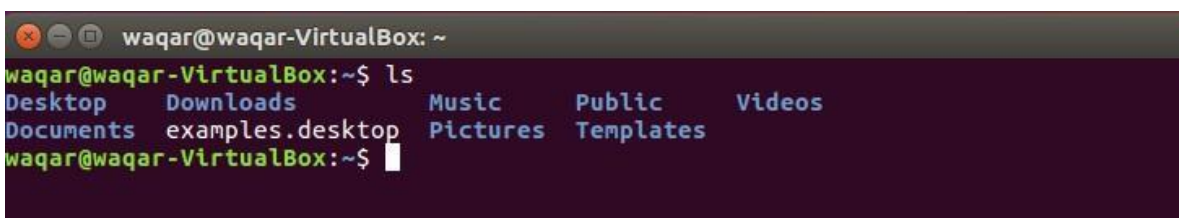
ls

The **ls** command (lowercase L and lowercase S) lists the contents of your current working directory.



```
waqar@waqar-VirtualBox: ~
waqar@waqar-VirtualBox:~$ ls
```

Figure 3: ls Command



```
waqar@waqar-VirtualBox: ~
waqar@waqar-VirtualBox:~$ ls
Desktop  Downloads  Music      Public     Videos
Documents examples.desktop Pictures    Templates
```

Figure 4: ls command execution

There may be no files visible in your home directory, in which case, the LINUX prompt

will be returned. Alternatively, there may already be some files inserted by the System Administrator when your account was created.

ls does not, in fact, cause all the files in your home directory to be listed, but only those ones whose name does not begin with a dot (.) Files beginning with a dot (.) are known as hidden files and usually contain important program configuration information. They are hidden because you should not change them unless you are very familiar with LINUX!!!

Note:

- Directories are denoted in blue color.
- Files are denoted in white.
- You will find similar color schemes in different flavors of Linux.

Suppose, in your "Music" folder has following sub-directories and files

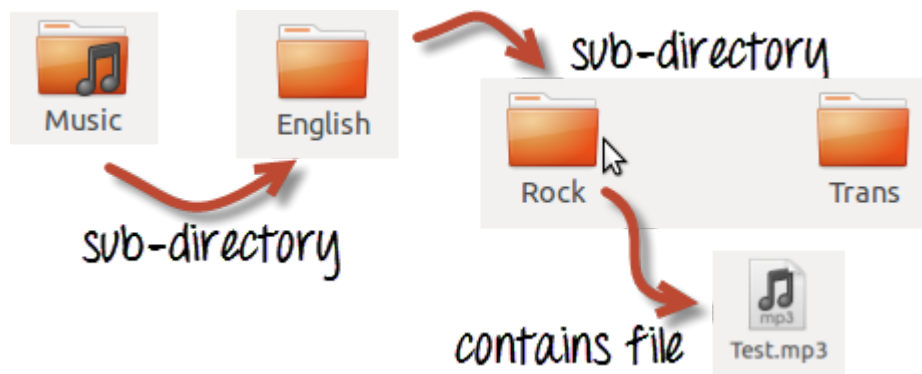


Figure 5: Sample Example

You can use '**ls -R**' to show all the files not only in directories but also subdirectories.

```
waqar@waqar-VirtualBox: ~
./Downloads/linux-4.13.11/virt/kvm:
arm          async_pf.o    coalesced_mmio.o  irqchip.c    kvm_main.c    vfio.h
async_pf.c   coalesced_mmio.c  eventfd.c         irqchip.o    kvm_main.o    vfio.o
async_pf.h   coalesced_mmio.h  eventfd.o         Kconfig      vfio.c

./Downloads/linux-4.13.11/virt/kvm/arm:
aarch32.c    arm.c        mmio.c         perf.c        psci.c        vgic
arch_timer.c hyp          mmu.c          pmu.c         trace.h

./Downloads/linux-4.13.11/virt/kvm/arm/hyp:
timer-sr.c   vgic-v2-sr.c  vgic-v3-sr.c

./Downloads/linux-4.13.11/virt/kvm/arm/vgic:
trace.h      vgic.h        vgic-its.c      vgic-mmio.h    vgic-v2.c
vgic.c       vgic-init.c   vgic-kvm-device.c  vgic-mmio-v2.c  vgic-v3.c
vgic-debug.c vgic-irqfd.c  vgic-mmio.c      vgic-mmio-v3.c

./Downloads/linux-4.13.11/virt/lib:
built-in.o   irqbypass.ko  irqbypass.mod.o  Kconfig        modules.builtin
irqbypass.c  irqbypass.mod.c  irqbypass.o      Makefile       modules.order
```

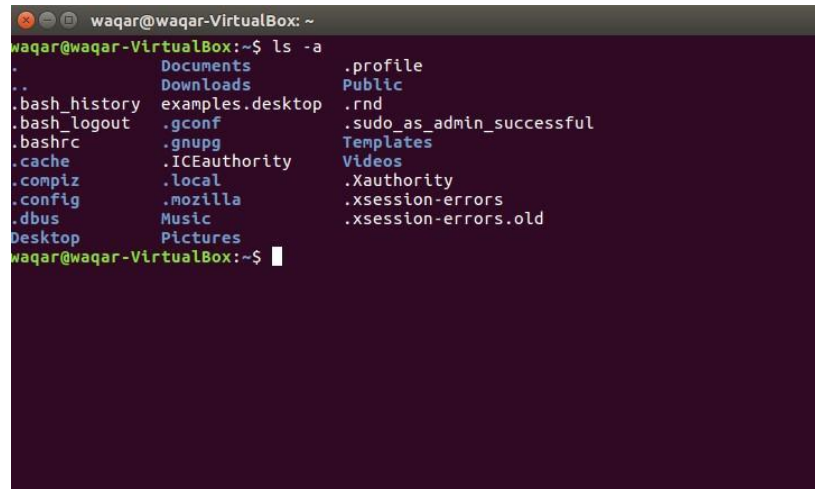
Figure 6: ls -r execution

c. Listing Hidden Files

To list all files in your home directory including those whose names begin with a dot type following command.

ls -a

As you can see, **ls -a** lists files that are normally hidden.



```
waqar@waqar-VirtualBox: ~  
waqar@waqar-VirtualBox:~$ ls -a  
.  
..  
.bash_history  
.bash_logout  
.bashrc  
.cache  
.compiz  
.config  
.dbus  
Desktop  
Documents  
Downloads  
examples.desktop  
.gconf  
.gnupg  
.ICEauthority  
.local  
.mozilla  
Music  
Pictures  
.profile  
Public  
.rnd  
.sudo_as_admin_successful  
Templates  
Videos  
.Xauthority  
.xsession-errors  
.xsession-errors.old  
waqar@waqar-VirtualBox:~$
```

Figure 7: `ls -R` command execution

ls is an example of a command which can take options: “**-a**” is an example of an option. The options change the behavior of the command. There are online manual pages that tell you which options a particular command can take, and how each option modifies the behavior of the command. (See later in this tutorial).

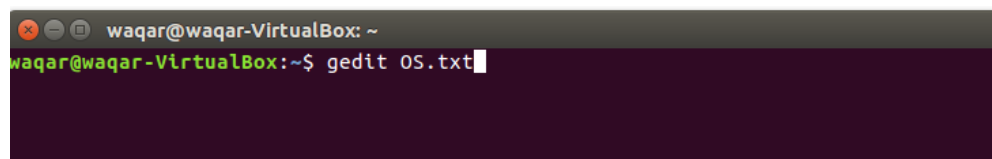
d. Creating & Viewing Files

You can use `gedit`, `cat` and `touch` command for creating a file.

1. Gedit

If you want to create a file and open in text editor then type following command

gedit yourfilename.extension
gedit OS.txt



```
waqar@waqar-VirtualBox: ~  
waqar@waqar-VirtualBox:~$ gedit OS.txt
```

Figure 8: File creating using `gedit`

2. Cat

If you want to create file and input in it using command line then you can use `cat`

command.

```
cat > yourfilename.extension  
cat > os.txt
```

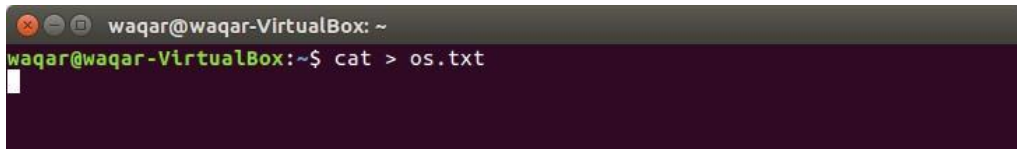
A terminal window with a dark purple background. The title bar shows 'waqar@waqar-VirtualBox: ~'. The prompt is 'waqar@waqar-VirtualBox:~\$' and the command 'cat > os.txt' has been entered. A white cursor is visible on the line following the command.

Figure 9: File creation using cat

After this what text you enter will be written in file and you need to press Ctrl+d to end the input in file.

3. Touch

If you want to create an empty file and don't want it to open in any editor or terminal then you can use touch command.

```
touch yourfilename.extension  
touch Os.txt
```

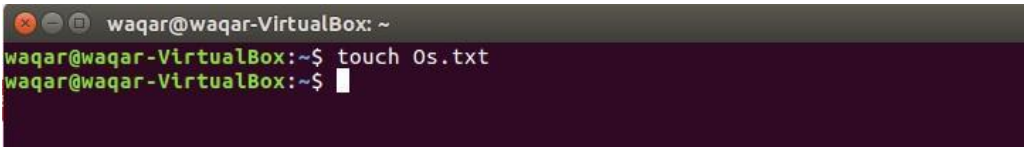
A terminal window with a dark purple background. The title bar shows 'waqar@waqar-VirtualBox: ~'. The prompt is 'waqar@waqar-VirtualBox:~\$' and the command 'touch Os.txt' has been entered. A white cursor is visible on the line following the command.

Figure 10: File creation using touch

To see the content of your file you can use **cat** command as well

```
cat yourfilename. extension  
cat OS.txt
```

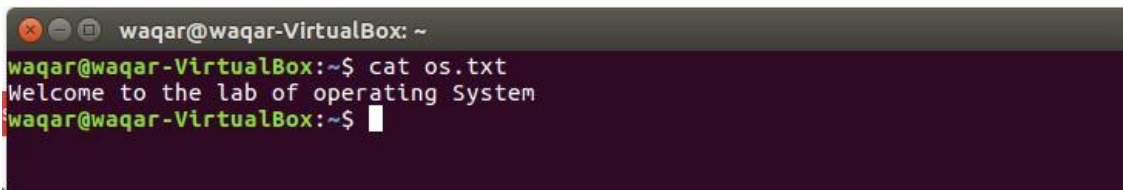
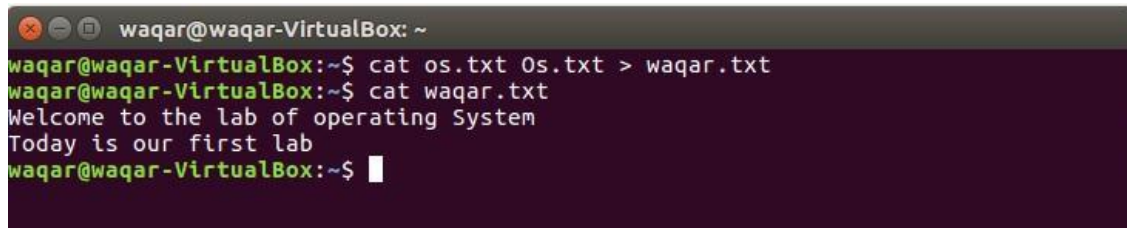
A terminal window with a dark purple background. The title bar shows 'waqar@waqar-VirtualBox: ~'. The prompt is 'waqar@waqar-VirtualBox:~\$' and the command 'cat os.txt' has been entered. The output 'Welcome to the lab of operating System' is displayed on the next line. A white cursor is visible on the line following the output.

Figure 11: Data display of file using cat

e. Combining Files

if you want to combine different files then you can also use cat command

```
cat yourfilename.extension yourfilename.extension > newfilename.extension  
cat OS.txt os.txt > waqar.txt
```

A screenshot of a terminal window titled 'waqar@waqar-VirtualBox: ~'. The terminal shows the following commands and output: 'cat os.txt Os.txt > waqar.txt', 'cat waqar.txt', and the output 'Welcome to the lab of operating System' followed by 'Today is our first lab'. The prompt 'waqar@waqar-VirtualBox:~\$' is visible at the end of the last line.

```
waqar@waqar-VirtualBox: ~
waqar@waqar-VirtualBox:~$ cat os.txt Os.txt > waqar.txt
waqar@waqar-VirtualBox:~$ cat waqar.txt
Welcome to the lab of operating System
Today is our first lab
waqar@waqar-VirtualBox:~$
```

Figure 12: Combining a file

f. Page wise content view

Let us assume that there is a big file or having data up to 30 Mb in the form of text. Than the size of terminal is small to display the data of file. So, we can display data page wise by using **less** command.

less yourfilename.extension
less OS.txt

Task1

You have to create 2 different text files. One with your name and the second one with your registration no. Copy and paste some dummy data after that do the following task

- 1) Display the data of both files
- 2) Display data page wise of both files
- 3) Combine both files in new file

You have to submit screenshots of all tasks.

g. Deleting Files

The 'rm' command removes files from the system without confirmation.

rm yourfilename.extension
rm os.txt

to check if file is delete or not type ls and see there will be no file with name os.txt

h. Moving & Renaming Files

mv file1 new_file_location is the command which moves a file to new destination.

What we are going to do now, is to take a file stored in an open access area of the file system and use the **mv** command to move it to your Desktop.

mv filename newfilename will rename the file.

i. Copying Files

cp *file1 file2* is the command which makes a copy of file1 in the current working

directory and calls it file2

What we are going to do now, is to take a file stored in an open access area of the file system and use the **cp** command to copy it to your Desktop.

j. Searching Contents of a file

i. Simple search using less

Using **less**, you can search through a text file for a keyword (pattern). For example, to search through os.txt for the word 'science', type

less os.txt

then, still in **less**, type a forward slash [/] followed by the word to search

/science

As you can see, **less** finds and highlights the keyword. Type [n] to search for the next occurrence of the word.

ii. Grep

grep is one of many standard LINUX utilities. It searches files for specified words or patterns. First clear the screen using clear command, then type

grep science os.txt

As you can see, **grep** has printed out each line containing the word **science**.

Or has it ????

Try typing

grep Science os.txt

The **grep** command is case sensitive; it distinguishes between Science and science.

To ignore upper/lower case distinctions, use the -i option, i.e. type

grep -i science os.txt

To search for a phrase or pattern, you must enclose it in single quotes (the apostrophe symbol). For example, to search for spinning top, type

grep

-i 'spinning top' os.txt

Some of the other options of grep are:

-v display those lines that do NOT match

-n precede each matching line with the line number

-c print only the total count of matched lines

Try some of them and see the different results. Don't forget, you can use more than one option at a time. For example, the number of lines without the words science or Science is

grep -ivc science os.txt

k. wc (word count) of File

A handy little utility is the **wc** command, short for word count. To do a word count on `os.txt`, type

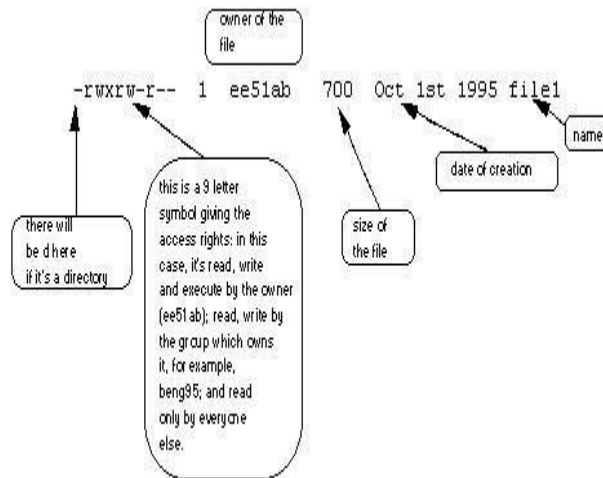
```
wc -w os.txt
```

To find out how many lines the file has, type

```
wc -l os.txt
```

I. ls -l (l for long listing!)

You will see that you now get lots of details about the contents of your directory, similar to the example below.



Each file (and directory) has associated access rights, which may be found by typing **ls -l**. Also, **ls -lg** gives additional information as to which group owns the file (`beng95` in the following example):

```
-rwxrw-r-- 1 ee51ab beng95 2450 Sept29 11:52 file1
```

In the left-hand column is a 10 symbol string consisting of the symbols `d`, `r`, `w`, `x`, `-`, and, occasionally, `s` or `S`. If `d` is present, it will be at the left hand end of the string, and indicates a directory: otherwise `-` will be the starting symbol of the string.

The 9 remaining symbols indicate the permissions, or access rights, and are taken as three groups of 3.

- The left group of 3 gives the file permissions for the user that owns the file (or directory) (`ee51ab` in the above example);
- the middle group gives the permissions for the group of people to whom the file (or directory) belongs (`eebeng95` in the above example);
- the rightmost group gives the permissions for all others.

The symbols `r`, `w`, etc., have slightly different meanings depending on whether they refer to a simple file or to a directory.

Access rights on files.

- `r` (or `-`), indicates read permission (or otherwise), that is, the presence or absence of

permission to read and copy the file

- w (or -), indicates write permission (or otherwise), that is, the permission (or otherwise) to change a file
- x (or -), indicates execution permission (or otherwise), that is, the permission to execute a file, where appropriate

Access rights on directories.

- r allows users to list files in the directory;
- w means that users may delete files from the directory or move files into it;
- x means the right to access files in the directory. This implies that you may read files in the directory provided you have read permission on the individual files.

So, in order to read a file, you must have execute permission on the directory containing that file, and hence on any directory containing that directory as a subdirectory, and so on, up the tree.

Some examples

- rwxrwxrwx: a file that everyone can read, write and execute (and delete).
- rwxr-xr-x: a file that only the owner can read and write - no-one else
- rw-----: can read or write and no-one has execution rights (e.g. your mailbox file).

m. Changing access rights

chmod (changing a file mode)

Only the owner of a file can use chmod to change the permissions of a file.

The options of chmod are as follows

Symbol	Meaning
u	user
g	group
o	other
a	all
r	read

- w write (and delete)
- x execute (and access directory)
- + add permission
- take away permission

For example, to remove read write and execute permissions on the file os.txt for the group and others, type

chmod go-rwx os.txt

This will leave the other permissions unaffected.

To give read and write permissions on the file os.txt to all,

chmod a+rw os.txt

Task2

Try changing access permissions on the file os.txt

Use **ls -l** to check that the permissions have changed.

Other useful LINUX commands

Additional Links:

<https://www.guru99.com/must-know-linux-commands.html>

<http://www.mediacollege.com/linux/command/linux-command.html>

<https://fossbytes.com/a-z-list-linux-command-line-reference/>

<https://www-uxsup.csx.cam.ac.uk/pub/doc/suse/suse9.0/userguide-9.0/ch24s04.html>

Operating System

L

Lab No 3 Compilation and Makefile

CLO: 1

Rubrics for Lab:

Task	0	1	2	3
As mentioned in the Task-1	Students don't know how to compile a program	Student do compile the program and know what to do	Student achieved the first two tasks but could not do the third	Student achieved all the given task

Topic to be covered

How to compile and run a c program on LINUX using gcc compiler

- C Code
- Compilation
- execution

Compile and execution multiple source files

- Create Multiple Files
- Compilation
- execution

Error and Warning Flags during compilation

Makefile

Objectives

Students are able to write, compile and execute their C code on linux.

Students are able to know how Error and Warning Flags are helpful during compilation of code.

Students are able to understand how Makefile help us to build code.

How to compile and run a c program on LINUX using gcc compiler

Step 1. Open up a terminal. ...

Step 2. Use a text editor of your choice to create the C source code.

For example

gedit

yourfil

And enter the C source code below:

ename

.c

gedit

hello.c

```
#include <stdio.h>
int main()
{
    printf ("Hello World\n");
}
```

Figure 13: C Code for hello world

Save the program and close the editor window

Step 3. Compile the program

Type the command on terminal

gcc yourfilename.c

gcc hello.c

This command will invoke the GNU C compiler to compile the file hello.c and result to an executable called **a.out**

Step 4. Execute the program.

Type the command

This should result in the output

`./a.out`

Hello World

Practice Task 1:

Now write a code that takes input string from a user “Hello this is my second program” and prints it on terminal. (*Figure out the command to take input from user*).

As we see that linux give the file name a.out by default and if we want to change the output file name then we need to use -o flag with gcc

gcc hello.c -o

newhello.out

-o switch

In this case, we use the -o option to specify a different output file for the executable, 'newhello.out'.

Now Run the compilation Command with -c Switch

gcc -c hello.c

Now check the output filename of this command there will be a file with name hello.o.

Now try to execute that file

./hello.o

Practice Task 2:

What will happen? Go to man page of gcc and read what -c flag does? Explain it in your own words

Compile and execution of multiple source files

Create a file functions.h and declare in it the following function,

void print_hello ();

Now create a file main.c and write following code in it/

```

#include <stdio.h>
#include "functions.h"
int main()
{
    print_hello ();
    return 0;
}

```

Figure 14: main.c

Now, create a functions.c and write the following code in it.

```

#include <stdio.h>
#include "functions.h"
void print_hello()
{
    printf("Hello World!");
}

```

Figure 15: hello.c

To compile and execute multiple source files with gcc, use the following command:

gcc main.c functions.c

./a.out

Warning and Error Flags

Warnings are diagnostic messages that report constructions that are not inherently erroneous but that are risky or suggest there may have been an error.

-Werror

Make all warnings into errors.

-Wall

This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros. This also enables some language-specific warnings.

Write a C program with

“test.c”.

```
int main()
{
    int * ptr= malloc(sizeof(int));
}
```

Lab Task:

Q1-Compile it and answer if there is any warning?

Q2-Then, if there is any warning convert it into an error.

Then,

Compile “test.c” with command,

gcc test.c -Wall -Werror

Q3-Explain, what happened now and if there is any warning and error then solve it.

Makefile

Makefile are special format files that together with the *make* utility will help you to automatically build and manage your projects. For this session you will need to create these files:

- main.c
- hello.c
- factorial.c
- functions.h



hello.c functions.h factorial.c main.c

Create a new directory and placing all the files in there. Main.c , function.h and hello.c is same as above and factorial.c calculate factorial

Build Process

1. Compiler takes the source files and outputs object files
2. Linker takes the object files and creates an executable

Compiling by hand

The trivial way to compile the files and obtain an executable, is by running the command:

```
gcc main.c hello.c factorial.c -o hello
```

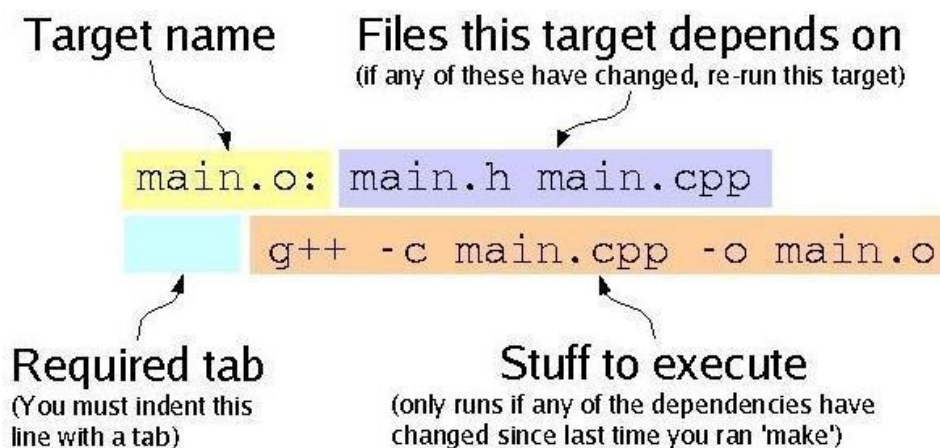
The basic Makefile

Create a file using gedit with name “Makefile”.

The basic Makefile is composed of:

```
target: dependencies
```

```
[tab] system command
```



Important Note: On this first example we see that our target is called *all*. This is the default target for makefiles. The *make* utility will execute this target if no other one is specified. We also see that there are no dependencies for target *all*, so *make* safely executes the system commands specified.

Finally, make compiles the program according to the command line we gave it.

Using dependency:

Sometimes is useful to use different targets. This is because if you modify a single file in your project, you don't have to recompile everything, only what you modified.

Here is an example:

all: hello.out

hello.out: main.o factorial.o hello.o

gcc main.o factorial.o hello.o -o hello.out

main.o: main.c

gcc -c main.c

factorial.o: factorial.c

gcc -c factorial.c

hello.o: hello.c

gcc -c hello.c

clean:

rm -rfv *.o *.out

Now we see that the target *all* has only dependencies, but no system commands. In order for *make* to execute correctly, it has to meet all the dependencies of the called target (in this case *all*).

Each of the dependencies are searched through all the targets available and executed if found. In this example we see a target called *clean*. It is useful to have such target if you want to have a fast way to get rid of all the object files and executables. _____

Using Variables and Comments

You can also use variables when writing Makefiles. It comes in handy in situations where you want to change the compiler, or the compiler options.

```
# I am a comment, and I want to say that the variable CC will be
# the compiler to use.
CC=gcc

# Hey!, I am comment number 2. I want to say that CFLAGS will be the
# options I'll pass to the compiler.
CFLAGS=-c -Wall

all: hello

hello: main.o factorial.o hello.o
    $(CC) main.o factorial.o hello.o -o hello

main.o: main.c
    $(CC) $(CFLAGS) main.c

factorial.o: factorial.c
    $(CC) $(CFLAGS) factorial.c

hello.o: hello.c
    $(CC) $(CFLAGS) hello.c

clean: rm *o hello
```

As you can see, variables can be very useful sometimes. To use them, just assign a value to a variable before you start to write your targets. After that, you can just use them with the dereference operator \$(VAR).

If you understand this example, you could adapt it to your own personal projects changing only 2 lines, no matter how many additional files you have.

Operating System

Lab No 4

Process Creation and Execution (fork() & exec())

CLO: 2

Rubrics for Lab:

Task	0	1	2	3
Understanding the Fork() and exec system call	Students don't know how Fork() and exec works	Students have knowledge of Fork() and exec() but couldn't answer the question given in the tasks	Student partial answered the question given in task	Students have complete knowledge of Fork() Commands and answer the question properly
As mentioned in task-2	Student has no knowledge of Command line argument.	Student has little knowledge of Command line argument and is not able to write full C code.	Students perform the task partially	Student has complete knowledge of Command line argument and performed the full task

Topic to be covered

Process Creation and execution

1. fork()
 - a. What is fork ()?
 - b. How fork () work?
 - c. Sample Program (Dry Run as well)
 - d. Graded Task
2. exec()
 - a. What is exec ()?
 - b. Sample Program (Dry Run as well)
3. Command Line Argument
 - a. Parsing command line arguments
 - b. Sample program
 - c. Graded Task

Objectives

After this lab students will be able to create and execute its own process in operating system.

They will be able to deal with command line arguments in operating system.

They will be able to deal with Xterm terminal.

What is fork ()?

System call **fork ()** is used to create processes. It takes no arguments and returns a process ID.

How fork () works?

The purpose of fork () is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork () system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork ():

If fork () returns a negative value, the creation of a child process was unsuccessful.

fork () returns a zero to the newly created child process.

fork () returns a positive value, the process ID of the child process, to the parent. The returned process ID is of type pid_t defined in sys/types.h. Normally, the process ID is an integer. Moreover, a process can use function getpid () to retrieve the process ID assigned to this process.

Therefore, after the system call to fork (), a simple test can tell which process is the

child. Please note that UNIX will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.

Sample Program

Processes are created with the `fork ()` system call (so the operation of creating a new process is sometimes called forking a process). The child process created by `fork` is a copy of the original parent process, except that it has its own process ID.

1. To display information about currently running processes use the `ps` command:

ps

2. `ps`tree is a small, command line program that displays the processes (i.e., executing instances of programs) on the system in the form of a tree diagram.

*ps*tree

Example 1:

Let us take an example to make the above points clear.

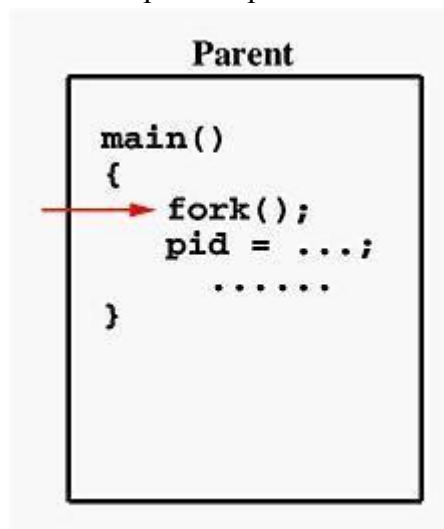
```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>

#define MAX_COUNT 200
#define BUF_SIZE 100

void main(void)
{
    pid_t pid;
    int i;
    char buf[BUF_SIZE];

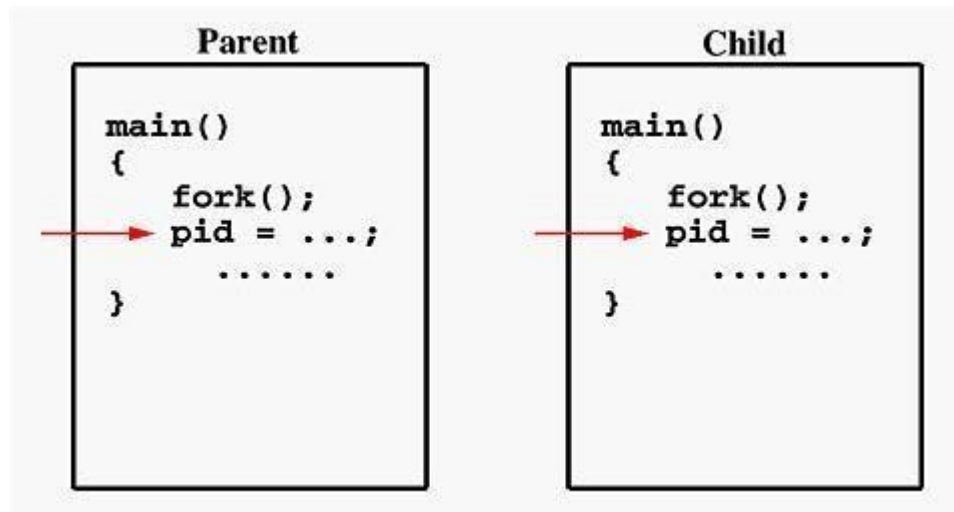
    fork();
    pid = getpid();
    for (i = 1; i <= MAX_COUNT; i++) {
        sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
        write(1, buf, strlen(buf));
    }
}
```

Suppose the above program executes up to the point of the call to `fork ()`.



If the call to **fork ()** is executed successfully, UNIX will

- Make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the **fork ()** call. In this case, both processes will start their execution at the assignment statement as shown below:



Both processes start their execution right after the system call **fork ()**. Since both processes have identical but separate address spaces, those variables initialized **before** the **fork ()** call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space. Other address spaces created by **fork ()** calls will not be affected even though they have identical variable names.

What is the reason of using **write** rather than **printf**? It is because **printf ()** is "buffered," meaning **printf ()** will group the output of a process together. While buffering the output for the parent process, the child may also use **printf** to print out some information, which will also be buffered. As a result, since the output will not be send to screen immediately, you may not get the right order of the expected result. Worse, the output from the two processes may be mixed in strange ways. To overcome this problem, you may consider to use the "unbuffered" **write**.

If you run this program, you might see the following on the screen:

```
.....
This line is from PID 3456, value 13
This line is from PID 3456, value 14
.....
This line is from PID 3456, value 20
This line is from PID 4617, value 100
This line is from PID 4617, value 101
.....
```

This line is from PID 3456, value 21
This line is from PID 3456, value 22

.....

Process ID 3456 may be the one assigned to the parent or the child. Due to the fact that these processes are run concurrently, their output lines are intermixed in a rather unpredictable way. Moreover, the order of these lines are determined by the CPU scheduler. Hence, if you run this program again, you may get a totally different result

Exercise 1 (Dry Run):

Task1:

```
#include <stdio.h> //standard c library for input output
#include <unistd.h> /* contains fork prototype */
int main()
{
    int pid;
    printf("Hello World!\n");
    printf("I am the parent process and pid is : %d.\n",getpid());
    printf("Here i am before use of forking\n");
    pid = fork(); //new process is created
    printf("Here I am just after forking\n"); //Child process will start execution from this line
    if (pid == 0)
        printf("I am the child process and pid is :%d.\n",getpid());
    else
        printf("I am the parent process and pid is: %d.\n",getpid());
}
```

Q1: Dry Run the above program and explain the output?

Task1 Graded (On Paper):

```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main()
{
    printf("Here I am just before first forking statement\n");
    fork();
    printf("Here I am just after first forking statement\n");
    fork();
    printf("Here I am just after second forking statement\n");
    printf("\t\tHello World from process %d!\n", getpid());
    return 0;
}
```

Q2: Tell how many processes are created in above source code and draw a process tree for given source code and dry run it.

Exec:

What is exec?

Forking provides a way for an existing process to start a new one, but what about the case where the new process is not part of the same program as parent process? This is the case in the shell; when a user starts a command it needs to run in a new process, but it is unrelated to the shell. This is where the exec system call comes into play. exec will replace the contents of the currently running process with the information from a program binary. The following code replaces the child process with the binary created for hello.c

Sample Program

```
#include <stdio.h>
int main()
{
    printf("Hello World\n");
}
```

Step 1: Create a file “hello.c” and type following source code

Step 2: Compile and make an executable file named hello.out

Step 3: Create another file “parent.c”. Type following code.

```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
int main(void)
{
    int pid;
    printf("I am the parent process and pid is : %d.\n",getpid());
    printf("Here i am before use of forking\n");
    pid = fork(); //new process is created
    printf("Here I am just after forking\n"); //Child process will start execution from this line
    if (pid == 0)
    {
        printf("I am the child process and pid is :%d.\n",getpid());
        printf("I am loading „hello” process\n");
        execv("hello.out", "hello.out", NULL);
    }
    else
    {
        printf("I am the parent process and pid is: %d.\n",getpid());
    }
}
```

Exercise 2:

Q1: Compile the above program? Do you see any error? If Yes! Explain the error in code?

Q2: Resolve the error in the code and handover the new file.

Parsing command line arguments

Introduction

Command-line parameters are passed to a program at run-time by the operating system when the program is requested by another program, such as a command interpreter ("shell") like cmd.exe on Windows or bash on Linux and OS X. The user types a command and the shell calls the operating system to run the program. Exactly how this is done is beyond the scope of this article (on Windows, look up Create Process; on UNIX

and UNIX-like

systems look up `fork(3)` and `exec(3)` in the manual).

The uses for command-line parameters are various, but the main two are:

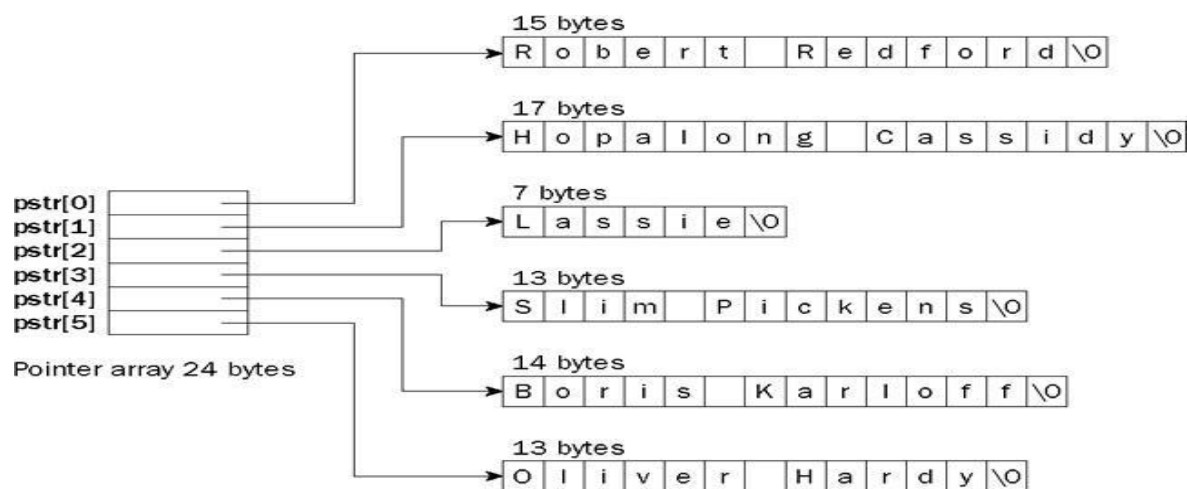
1. Modifying program behavior - command-line parameters can be used to tell a program how you expect it to behave; for example, some programs have a `-q` (quiet) option to tell them not to output as much text.
2. Having a program run without user interaction - this is especially useful for programs that are called from scripts or other programs.

The command-line

Adding the ability to parse command-line parameters to a program is very easy. Every C and C++ program has a main function. In a program without the capability to parse its command-line, main is usually defined like this:

```
int main()
```

To see the command-line we must add two parameters to main which are, by convention, named `argc` (argument count) and `argv` (argument vector [here, vector refers to an array, not a C++ or Euclidean vector]). `argc` has the type `int` and `argv` usually has the type `char**` or `char* []` (see below). main now looks like this:



Total Memory is 103 bytes

```
Int main (int argc , char * argv[]) // char ** argv
```

`argc` tells you how many command-line arguments there were. It is always at least 1,

because the first string in argv (argv[0]) is the command used to invoke the program. argv contains the actual command-line arguments as an array of strings, the first of which (as we have already discovered) is the program's name.

Earlier it was mentioned that argc contains the number of arguments passed to the program. This is useful as it can tell us when the user hasn't passed the correct number of arguments, and we can then inform the user of how to run our program (Error check).

Exercise 3:

Q1: Write a program that prints out all the parameters that are passed to it via command line (Hint: argc)

For instance *./program.out hello Pakistan*

Should print out put:

Number of parameters 2

Hello

Pakistan

Graded task 2

Write another program that takes two integers as command line parameters and prints the sum of the two parameters. (Hint: The parameters are treated as strings and not integers.)

Operating System Lab No 5

Inter Process Communication (PIPES)

CLO: 2

Rubrics for Lab:

Task	0	1	2	3
As mentioned in the task	Student don't know about the concept of pipes and could not write the code	Student know the concept of pipe and do code some of the portion of the task	Student performed the task partially	Student performed the task completely

Topics to be Covered

- Inter process Communication
- Pipes in C
- Pipes Syntax
 - Read
 - Write
- Communication between multiple files

Objectives

Students are able to understand the Inter process Communication IPC's and able to do communication between processes.

Inter-process Communication

As name suggests, sending messages or useful data between two processes. Shortly say IPC. There are different mechanisms available for IPC. This lab will focus on Process pipes mechanism.

Process Pipes

Piping is a process where the output of one process is made the input of another. LINUX users have seen examples of this from the UNIX command line using pipe sign '|'. For your experience just do a following quick activity.

Exercise 1:

Step1: Following command will simply show you the contents of directories and subdirectory's contents of current working directory. Try it out.

find .

Step2: Above command will show you whole output in one go. If you want to examine its output like file contents. Send output of Step2 to another command "more". You are already familiar with it in lab2.

Step3: Try following command.

find. | more

Pipe sign in above command is serving as "IPC pipe" taking output of find and sending it to more as input

Pipes in C

We will now see how we do this from C programs. We will have two forked processes and will communicate between them.

We must first open a pipe. UNIX allows two ways of opening a pipe.

i. Formatted Piping - popen()

ii. Low level Piping –

pipe() We will use second one in this lab.

Note: Learn more about both in chapter 12, Beginning LINUX Programming by WROX – 2nd Edition or Google Pipe ()

Step 1:

man pipe

read how pipe work, get the necessary information before code.

pipe()

It's a low-level function to create a pipe. This provides a means of passing data between two programs, without any overhead. It also gives us more control over the reading and writing of data.

The pipe function has the prototype

```
#include <unistd.h>
int pipe(int file_descriptor[2]);
```

pipe() is passed (a pointer to) an array of two integer file descriptors. It fills the array with two new file descriptors

pipe() returns a zero on success. On failure, it returns -1 and sets errno to indicate the reason for failure. See man pages for error descriptions.

The two file descriptors returned are connected in a special way. Any data written to

a first in, first out basis, usually abbreviated to FIFO. This means that if you write the bytes 1, 2, 3 to `file_descriptor[1]`, reading from `file_descriptor[0]` will produce 1, 2, 3. This is different from a stack, which operates on a last in, first out basis, usually abbreviated to LIFO.

Important Note: It's important to realize that these are file descriptors, not file streams, so we must use the lower-level read and write calls to access the data, rather than `fread` and `fwrite`.

Let's do following lab activity to understand it practically!

Exercise 2

Step1: create a file with name "pipe1.c". Type in the following code.

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

int main (int argc , char ** argv)
{
    int data_processed=0;
    int file_descriptor[2];
    const char some_data[] = "123";
    char buffer[BUFSIZ + 1];
    memset(buffer, '\0', sizeof(buffer));

    if(pipe(file_descriptor)==0){
        data_processed = write(); // see man for write parameter
        printf("Wrote %d bytes:", data_processed);
        data_processed = read(); // see man for read parameter
        printf("Read %d, byte %s", data_processed, buffer);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

Exercise:

1. Compile and run it.
2. Reverse the order of the `write()` and `read()` and run the program again.
3. Briefly describe what happened with the reversed order and why the program behaved that way.
4. Modify `pipe1.c` in the following ways:
 - a. Dynamically allocate buffer so it's exactly the right size for `some_data`.
 - b. Copy the string from `some_data` into buffer.

c. Modify the read so its third argument is the exact size of buffer (rather than the

Graded Task : Multiple pipes across a fork/exec

In this task, you will write a program that has two-way communication between parent and child. Write a c program in which parent receive array of integer from command line argument and pass it to child using pipe, child need to sort that array (using bubble sort) and return that array to parent using pipe and parent need to display that sorted array.

Operating System Lab No 6

POSIX THREADS

CLO: 3

Rubrics for Lab:

Task	0	1	2	3
As mentioned in the task	Student don't know about the pthread library	Student know the concept of thread and could not use the pthread library properly	Student performed the given task partially	Student performed the full task by using the proper threading concept

Topics to be Covered

Threads Overview

- What is thread?
- What are Pthread?
- Why Pthreads?

Designing threaded program?

Compiling Threaded Program?

- Sub routine Groups

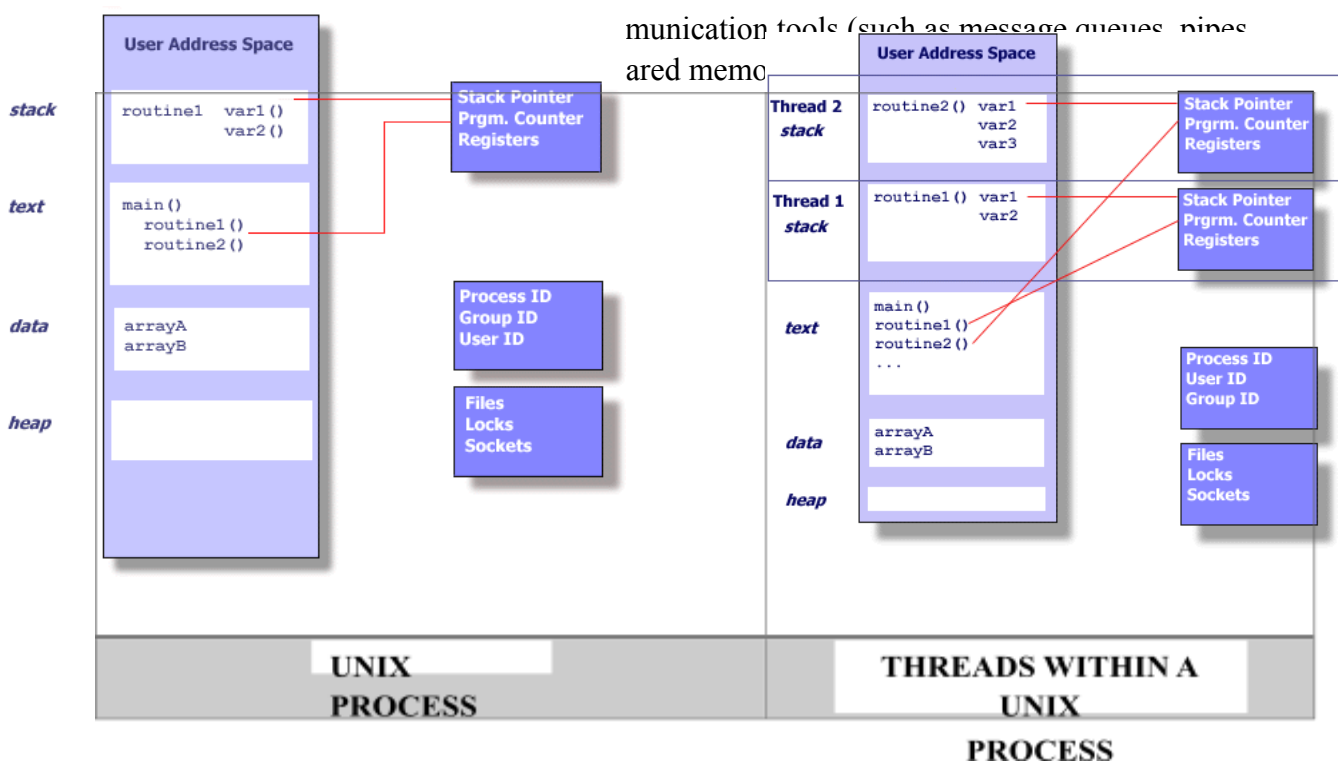
Thread Management?

- Creating and Terminating Thread?
- Passing Arguments to Threads?

Threads Overview?

What is a Thread?

- Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system. But what does this mean?
- To the software developer, the concept of a "procedure" that runs independently from its main program may best describe a thread.
- To go one step further, imagine a main program (a.out) that contains a number of procedures. Then imagine all of these procedures being able to be scheduled to run simultaneously and/or independently by the operating system. That would describe a "multi-threaded" program.
- How is this accomplished?
- Before understanding a thread, one first needs to understand a UNIX process. A process is created by the operating system, and requires a fair amount of "overhead". Processes contain information about program resources and program execution state, including:
 - o Process ID, process group ID, user ID, and group ID
 - o Environment
 - o Working directory.
 - o Program instructions
 - o Registers
 - o Stack
 - o Heap
 - o File descriptors
 - o Signal actions
 - o Shared libraries



- Threads use and exist within these process resources, yet are able to be scheduled by the operating system and run as independent entities largely because they duplicate only the bare essential resources that enable them to exist as executable code.
- This independent flow of control is accomplished because a thread maintains its own:
 - o Stack pointer
 - o Registers
 - o Scheduling properties (such as policy or priority)
 - o Set of pending and blocked signals
 - o Thread specific data.
- So, in summary, in the UNIX environment a thread:
 - o Exists within a process and uses the process resources
 - o Has its own independent flow of control as long as its parent process exists and the OS supports it
 - o Duplicates only the essential resources it needs to be independently schedulable
 - o May share the process resources with other threads that act equally independently (and dependently)
 - o Dies if the parent process dies - or something similar
 - o Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.
- Because threads within the same process share resources:
 - o Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
 - o Two pointers having the same value point to the same data.
 - o Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

What are Pthreads?

- Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.
- In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required.
 - o For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995).
 - o Implementations adhering to this standard are referred to as POSIX threads, or Pthreads.
 - o Most hardware vendors now offer Pthreads in addition to their proprietary API's.
- The POSIX standard has continued to evolve and undergo revisions, including the Pthreads specification.
- Some useful links:

- o standards.ieee.org/findstds/standard/1003.1-2008.html
- o www.opengroup.org/austin/papers/posix_faq.html
- o www.unix.org/version3/ieee_std.html
- Pthreads are defined as a set of C language programming types and procedure calls, implemented with a **pthread.h** header/include file and a thread library
 - though this library may be part of another library, such as **libc**, in some implementations.

Why Pthreads?

► Light

Weight:

- When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.
- For example, the following table compares timing results for the **fork()** subroutine and the **pthread_create()** subroutine. Timings reflect 50,000 process/thread creations, were performed with the **time** utility, and units are in seconds, no optimization flags.

Note: don't expect the system and user times to add up to real time, because these are SMP systems with multiple CPUs/cores working on the problem at the same time. At best, these are approximations run on local machines, past and present.

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

► Efficient Communications/Data Exchange:

- The primary motivation for considering the use of Pthreads in a high performance computing environment is to achieve optimum performance. In

particular, if an application is using MPI for on-node communications, there is a potential that performance could be improved by using Pthreads instead.

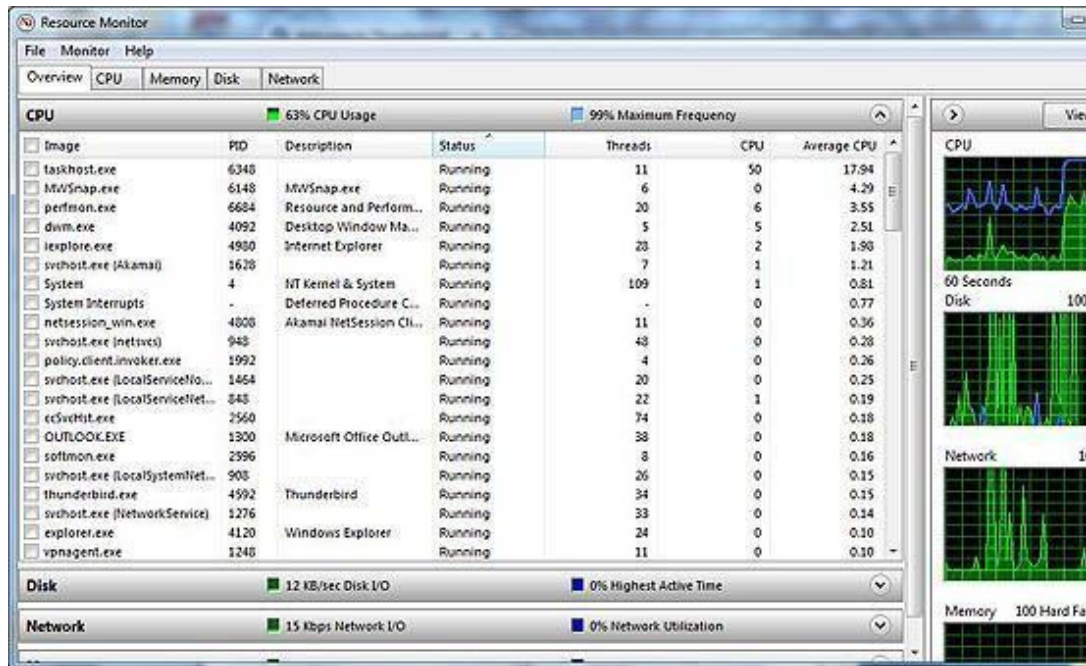
- MPI libraries usually implement on-node task communication via shared memory, which involves at least one memory copy operation (process to process).
- For Pthreads there is no intermediate memory copy required because threads share the same address space within a single process. There is no data transfer, per se. It can be as efficient as simply passing a pointer.
- In the worst case scenario, Pthread communications become more of a cache- to-CPU or memory-to-CPU bandwidth issue. These speeds are much higher than MPI shared memory communications.
- For example: some local comparisons, past and present, are shown below:

Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
Intel 2.6 GHz Xeon E5-2670	4.5	51.2
Intel 2.8 GHz Xeon 5660	5.6	32
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

► Other Common Reasons:

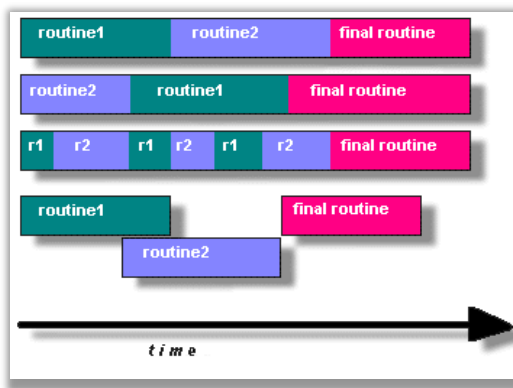
- Threaded applications offer potential performance gains and practical advantages over non-threaded applications in several other ways:
 - o Overlapping CPU work with I/O: For example, a program may have sections where it is performing a long I/O operation. While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.
 - o Priority/real-time scheduling: tasks which are more important can be scheduled to supersede or interrupt lower priority tasks.
 - o Asynchronous event handling: tasks which service events of indeterminate frequency and duration can be interleaved. For example, a web server can both transfer data from previous requests and manage the arrival of new requests.

- A perfect example is the typical web browser, where many interleaved tasks can be happening at the same time, and where tasks can vary in priority.
- Another good example is a modern operating system, which makes extensive use of threads. A screenshot of the MS Windows OS and applications using threads is shown below.



Parallel Programming:

- On modern, multi-core machines, pthreads are ideally suited for parallel programming, and whatever applies to parallel programming in general, applies to parallel pthreads programs.
- There are many considerations for designing parallel programs, such as:
 - o What type of parallel programming model to use?
 - o Problem partitioning
 - o Load balancing
 - o Communications
 - o Data dependencies
 - o Synchronization and race conditions
 - o Memory issues
 - o I/O issues
 - o Program complexity
 - o Programmer effort/costs/time



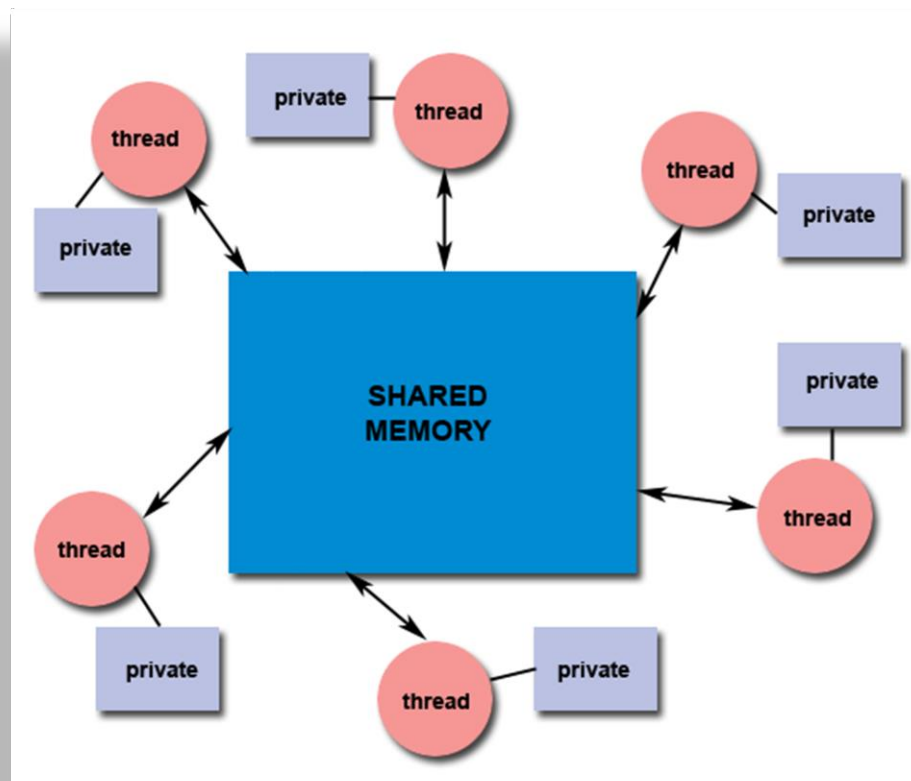
topics is beyond the scope of this tutorial, however interested obtain a quick overview in the [Introduction to Parallel](#) trial.

ugh, in order for a program to take Pthreads, it must be able to be discrete, independent tasks which concurrently. For example, if routine1 can be interchanged, interleaved &/or overlapped in real time, they are candidates for threading.

- Programs having the following characteristics may be well suited for pthreads:
 - o Work that can be executed, or data that can be operated on, by multiple tasks simultaneously:
 - o Block for potentially long I/O waits
 - o Use many CPU cycles in some places but not others
 - o Must respond to asynchronous events
 - o Some work is more important than other work (priority interrupts)
- Several common models for threaded programs exist:
 - o **Manager/worker:** a single thread, the *manager* assigns work to other threads, the *workers*. Typically, the manager handles all input and parcels out work to the other tasks. At least two forms of the manager/worker model are common: static worker pool and dynamic worker pool.
 - o **Pipeline:** a task is broken into a series of suboperations, each of which is handled in series, but concurrently, by a different thread. An automobile assembly line best describes this model.
 - o **Peer:** similar to the manager/worker model, but after the main thread creates other threads, it participates in the work.

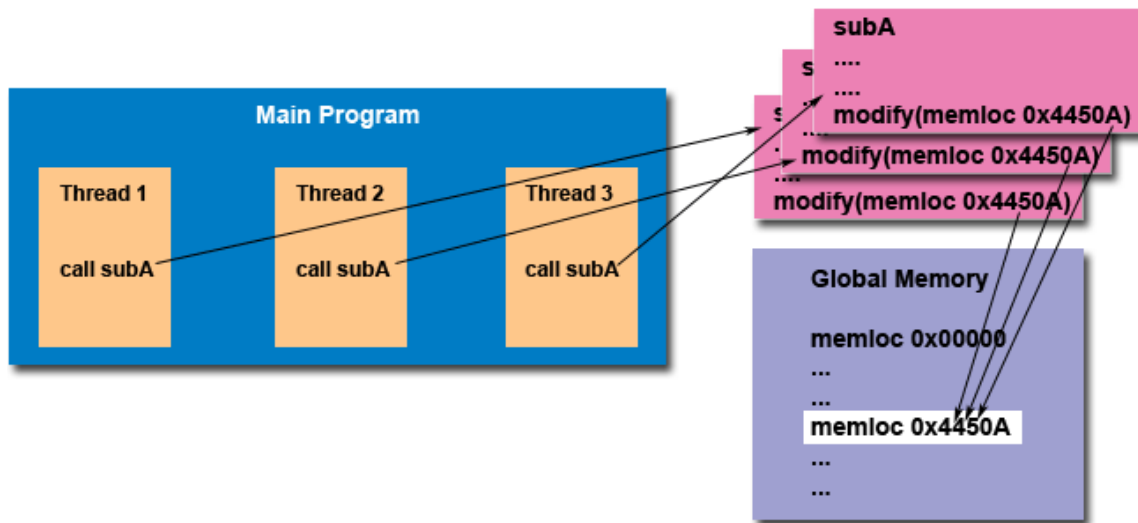
► Shared Memory Model:

- All threads have access to the same global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access (protecting) globally shared data.



► Thread-safeness:

- Thread-safeness: in a nutshell, refers an application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions.
- For example, suppose that your application creates several threads, each of which makes a call to the same library routine:
 - o This library routine accesses/modifies a global structure or location in memory.
 - o As each thread calls this routine it is possible that they may try to modify this global structure/memory location at the same time.
 - o If the routine does not employ some sort of synchronization constructs to prevent data corruption, then it is not thread-safe.



- The implication to users of external library routines is that if you aren't 100% certain the routine is thread-safe, then you take your chances with problems that could arise.
- Recommendation: Be careful if your application uses libraries or other objects that don't explicitly guarantee thread-safeness. When in doubt, assume that they are not thread-safe until proven otherwise. This can be done by "serializing" the calls to the uncertain routine, etc.

► Thread Limits:

- Although the Pthreads API is an ANSI/IEEE standard, implementations can, and usually do, vary in ways not specified by the standard.
- Because of this, a program that runs fine on one platform, may fail or produce wrong results on another platform.
- For example, the maximum number of threads permitted, and the default thread stack size are two important limits to consider when designing your program.
- Several thread limits are discussed in more detail later in this tutorial.

Pthread API

- The original Pthreads API was defined in the ANSI/IEEE POSIX 1003.1 - 1995 standard. The POSIX standard has continued to evolve and undergo revisions, including the Pthreads specification.
- Copies of the standard can be purchased from IEEE or downloaded for free from other sites online.
- The subroutines which comprise the Pthreads API can be informally grouped into four major groups:
 1. **Thread management:** Routines that work directly on threads - creating, detaching, joining, etc. They also include functions to set/query thread attributes (joinable, scheduling etc.)

2. **Mutexes:** Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.
 3. **Condition variables:** Routines that address communications between threads that share a mutex. Based upon programmer specified conditions. This group includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.
 4. **Synchronization:** Routines that manage read/write locks and barriers.
- Naming conventions: All identifiers in the threads library begin with **pthread_**. Some examples are shown below.

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers

- The concept of opaque objects pervades the design of the API. The basic calls work to create or modify opaque objects - the opaque objects can be modified by calls to attribute functions, which deal with opaque attributes.
- The Pthreads API contains around 100 subroutines. This tutorial will focus on a subset of these - specifically, those which are most likely to be immediately useful to the beginning Pthreads programmer.
- *For portability, the **pthread.h** header file should be included in each source file using the Pthreads library.*
- The current POSIX standard is defined only for the C language. Fortran programmers can use wrappers around C function calls. Some Fortran compilers may provide a Fortran pthreads API.

- A number of excellent books about Pthreads are available. Several of these are listed in the [References](#) section of this tutorial.

Compiling Threaded Program

Several examples of compile commands used for pthreads codes are listed in the table below.

Compiler / Platform	Compiler Command	Description
INTEL Linux	icc -pthread	C
	icpc -pthread	C++
PGI Linux	pgcc -lpthread	C
	pgCC -lpthread	C++
GNU Linux, Blue Gene	gcc -pthread	GNU C
	g++ -pthread	GNU C++
IBM Blue Gene	bgxlc_r / bgcc_r	C (ANSI / non-ANSI)
	bgxlc_r, bgxlc++_r	C++

Thread Management

Creating and Terminating Threads

▶ Routines:

```
pthread_create
(thread,attr,start_routine,arg) pthread_exit
(status)
pthread_cancel (thread)
pthread_attr_init (attr)
pthread_attr_destroy
(attr)
```

▶ Creating Threads:

- Initially, your **main()** program comprises a single, default thread. All other threads must be explicitly created by the programmer.
- **pthread_create** creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.
- **pthread_create** arguments:

pthread_create(thread, attr, start_routine, arg);

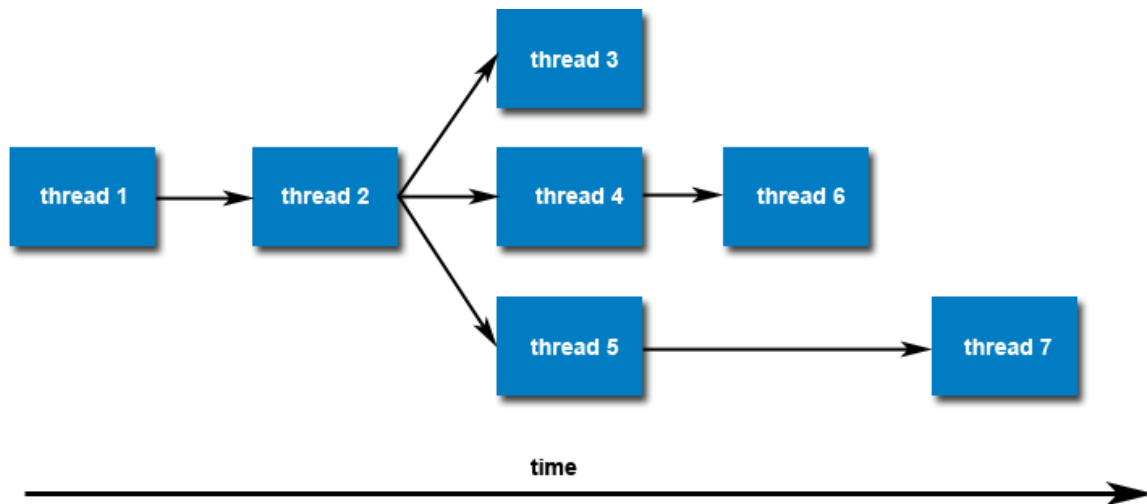
- o **thread**: An opaque, unique identifier for the new thread returned by the subroutine.
- o **attr**: An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.

- o **start_routine**: the C routine that the thread will execute once it is created.
- o **arg**: A single argument that may be passed to *start_routine*. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

For example: You need to create a thread, which runs a function hello world with no argument. Command will be as follow

```
pthread_create(thread_id, NULL, Hello_World, NULL);
```

- The maximum number of threads that may be created by a process is implementation dependent. Programs that attempt to exceed the limit can fail or produce wrong results.
- Querying and setting your implementation's thread limit - Linux example shown. Demonstrates querying the default (soft) limits and then setting the maximum number of processes (including threads) to the hard limit. Then verifying that the limit has been overridden.
- Once created, threads are peers, and may create other threads. There is no implied hierarchy or dependency between threads.



► Thread Attributes:

- By default, a thread is created with certain attributes. Some of these attributes can be changed by the programmer via the thread attribute object.
- **pthread_attr_init** and **pthread_attr_destroy** are used to initialize/destroy the thread attribute object.
- Other routines are then used to query/set specific attributes in the thread attribute object. Attributes include:
 - o Detached or joinable state
 - o Scheduling inheritance

- o Scheduling policy
- o Scheduling parameters
- o Scheduling contention scope
- o Stack size
- o Stack address
- o Stack guard (overflow) size
- Some of these attributes will be discussed later.



Question: After a thread has been created, how do you know
a) When it will be scheduled to run by the operating

Thread Binding and Scheduling:

system?

b) Which processor/core it will run on?

- The Pthreads API provides several routines that may be used to specify how threads are scheduled for execution. For example, threads can be scheduled to run FIFO (first-in first-out), RR (round-robin) or OTHER (operating system determines). It also provides the ability to set a thread's scheduling priority value.
- These topics are not covered here, however a good overview of "how things work" under Linux can be found in the **sched_setscheduler** man page.
- The Pthreads API does not provide routines for binding threads to specific cpus/cores. However, local implementations may include this functionality - such as providing the non-standard **pthread_setaffinity_np** routine. Note that "_np" in the name stands for "non-portable".
- Also, the local operating system may provide a way to do this. For example, Linux provides the **sched_setaffinity** routine.

▶ Terminating Threads & pthread_exit():

- There are several ways in which a thread may be terminated:
 - o The thread returns normally from its starting routine. Its work is done.
 - o The thread makes a call to the **pthread_exit** subroutine - whether its work is done or not.
 - o The thread is canceled by another thread via the **pthread_cancel** routine.

- o The entire process is terminated due to making a call to either the **exec()** or **exit()**

- o If `main()` finishes first, without calling **`pthread_exit`** explicitly itself
- The **`pthread_exit()`** routine allows the programmer to specify an optional termination *status* parameter. This optional parameter is typically returned to threads "joining" the terminated thread (covered later).
- In subroutines that execute to completion normally, you can often dispense with calling **`pthread_exit()`** - unless, of course, you want to pass the optional status code back.
- Cleanup: the **`pthread_exit()`** routine does not close files; any files opened inside the thread will remain open after the thread is terminated.
- **Discussion on calling `pthread_exit()` from `main()`:**
 - o There is a definite problem if `main()` finishes before the threads it spawned if you don't call **`pthread_exit()`** explicitly. All of the threads it created will terminate because `main()` is done and no longer exists to support the threads.
 - o By having `main()` explicitly call **`pthread_exit()`** as the last thing it does, `main()` will block and be kept alive to support the threads it created until they are done.

Example: Pthread Creation and Termination

- This simple example code creates 5 threads with the **`pthread_create()`** routine. Each thread prints a "Hello World!" message, and then terminates with a call to **`pthread_exit()`**.

Pthread Creation and Termination Example

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS    5

void *PrintHello(void *threadid)
{
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0; t<NUM_THREADS; t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc){
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }

    /* Last thing that main() should do */
    pthread_exit(NULL);
}
```

Passing Arguments to Threads

- The **pthread_create()** routine permits the programmer to pass one argument to the thread start routine. For cases where multiple arguments must be passed, this limitation is easily overcome by creating a structure which contains all of the arguments, and then passing a pointer to that structure in the **pthread_create()** routine.
- All arguments must be passed by reference and cast to (void *).



Question: How can you safely pass data to newly created threads, given their non-deterministic start-up and scheduling?

```

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS      8

char *messages[NUM_THREADS];

struct thread_data
{
    int  thread_id;
    int  sum;
    char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
    int taskid, sum;
    char *hello_msg;
    struct thread_data *my_data;

    sleep(1);
    my_data = (struct thread_data *) threadarg;
    taskid = my_data->thread_id;
    sum = my_data->sum;
    hello_msg = my_data->message;
    printf("Thread %d: %s Sum=%d\n", taskid, hello_msg, sum);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int *taskids[NUM_THREADS];
    int rc, t, sum;

```

```

sum=0;
messages[0] = "English: Hello World!";
messages[1] = "French: Bonjour, le monde!";
messages[2] = "Spanish: Hola al mundo";
messages[3] = "Klingon: Nuq neH!";
messages[4] = "German: Guten Tag, Welt!";
messages[5] = "Russian: Zdravstvuyte, mir!";
messages[6] = "Japan: Sekai e konnichiwa!";
messages[7] = "Latin: Orbis, te saluto!";

for(t=0;t<NUM_THREADS;t++) {
    sum = sum + t;
    thread_data_array[t].thread_id = t;
    thread_data_array[t].sum = sum;
    thread_data_array[t].message = messages[t];
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *)
        &thread_data_array[t]);
    if (rc) {
        printf("ERROR; return code from pthread_create() is %d\n", rc);
        exit(-1);
    }
}
pthread_exit(NULL);
}

```

TASK TO SUBMIT

You have to create a calculator where the user enters two numbers and one command. These three inputs are to be taken using command line argument. The command is the task associated with the number where (A=addition, S=subtraction, M=multiply). For example

./threads.out 3 4 M

Multiplication result is 12.

Implement this scenario using pthreads where each task performed is linked with one thread i.e.

Thread 1: Addition

Thread 2: Subtraction

Thread 3: multiplication

Write the code where only one thread will execute depending upon the task associated with the command.

Operating System Lab No 7

Producer-Consumer problem

CLO: 3,6

Rubrics for Lab:

Task	0	1	2	3
Understanding the concept of producer consumer-problem solution	Student is not able to create and manage semaphore variables.	Student can create variable but could not implement the produce consumer problem	• Student Write the proper code of produce consumer problem but couldn't run and present	• Student completely implement the concept of producer consumer problem solution and presented properly

Aim

To synchronize producer and consumer processes using semaphore.

Semaphores

- ☐ A semaphore is a counter used to synchronize access to a shared data amongst multiple processes.
- ☐ To obtain a shared resource, the process should:
 - o Test the semaphore that controls the resource.
 - o If value is positive, it gains access and decrements value of semaphore.
 - o If value is zero, the process goes to sleep and awakes when value is > 0 .
- ☐ When a process relinquishes resource, it increments the value of semaphore by 1.

Producer-Consumer problem

- ☐ A producer process produces information to be consumed by a consumer process
- ☐ A producer can produce one item while the consumer is consuming another one.
- ☐ With bounded-buffer size, consumer must wait if buffer is empty, whereas producer must wait if buffer is full.
- ☐ The buffer can be implemented using any IPC facility.

Algorithm

1. Create a shared memory segment *BUFSIZE* of size 1 and attach it.
2. Obtain semaphore id for variables *empty*, *mutex* and *full* using *semget* function.
3. Create semaphore for *empty*, *mutex* and *full* as follows:
 - a. Declare *semun*, a union of specific commands.

- b. The initial values are: 1 for mutex, N for empty and 0 for full
 - c. Use semctl function with SETVAL command
- 4. Create a child process using fork system call.

- a. Make the parent process to be the *producer*
 - b. Make the child process to be the *consumer*
- 5. The *producer* produces 5 items as follows:
 - a. Call *wait* operation on semaphores *empty* and *mutex* using semop function.
 - b. Gain access to buffer and produce data for consumption
 - c. Call *signal* operation on semaphores *mutex* and *full* using semop function.
- 6. The *consumer* consumes 5 items as follows:
 - a. Call *wait* operation on semaphores *full* and *mutex* using semop function.
 - b. Gain access to buffer and consume the available data.
 - c. Call *signal* operation on semaphores *mutex* and *empty* using semop function.
- 7. Remove shared memory from the system using shmctl with IPC_RMID argument
- 8. Stop

Program

```
/* Producer-Consumer problem using semaphore – pcsem.c */ #include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h> #include
<sys/shm.h> #include
<sys/sem.h>

#define N 5
#define BUFSIZE 1
#define PERMS 0666

int *buffer;
int nextp = 0, nextc = 0;
int mutex, full, empty;          /* semaphore variables */

void producer()
{
    int data; if(nextp
    == N)
        nextp = 0;
    printf("Enter data for producer to produce : "); scanf("%d", (buffer + nextp));
    nextp++;
}

void consumer()
{
    int g; if(nextc
    == N)
        nextc = 0;
    g = *(buffer + nextc++); printf("\nConsumer consumes
    data %d", g);
}

void sem_op(int id, int value)
{
    struct sembuf op; int v;
    op.sem_num = 0; op.sem_op =
    value; op.sem_flg =
    SEM_UNDO;
    if((v = semop(id, &op, 1)) < 0)
        printf("\nError executing semop instruction");
}
```

```

void sem_create(int semid, int initval)
{
    int semval;
    union semun
    {
        int val;
        struct semid_ds *buf;
        unsigned short *array;
    } s;

    s.val = initval;
    if((semval = semctl(semid, 0, SETVAL, s)) < 0) printf("\nError in executing
        semctl");
}

void sem_wait(int id)
{
    int value = -1; sem_op(id,
        value);
}

void sem_signal(int id)
{
    int value = 1; sem_op(id,
        value);
}

main()
{
    int shmid, i; pid_t pid;

    if((shmid = shmget(1000, BUFSIZE, IPC_CREAT|PERMS)) < 0)
    {
        printf("\nUnable to create shared memory"); return;
    }
    if((buffer = (int*)shmat(shmid, (char*)0, 0)) == (int*)-1)
    {
        printf("\nShared memory allocation error\n"); exit(1);
    }

    if((mutex = semget(IPC_PRIVATE, 1, PERMS|IPC_CREAT)) == -1)
    {
        printf("\nCan't create mutex semaphore"); exit(1);
    }
}

```

```

if((empty = semget(IPC_PRIVATE, 1, PERMS|IPC_CREAT)) == -1)
{
    printf("\nCan't create empty semaphore"); exit(1);
}
if((full = semget(IPC_PRIVATE, 1, PERMS|IPC_CREAT)) == -1)
{
    printf("\nCan't create full semaphore"); exit(1);
}

sem_create(mutex,      1);
sem_create(empty,      N);
sem_create(full, 0);

if((pid = fork()) < 0)
{
    printf("\nError in process creation"); exit(1);
}
else if(pid > 0)
{
    for(i=0; i<N; i++)
    {
        sem_wait(empty);
        sem_wait(mutex); producer();
        sem_signal(mutex);
        sem_signal(full);
    }
}
else if(pid == 0)
{
    for(i=0; i<N; i++)
    {
        sem_wait(full); sem_wait(mutex);
        consumer(); sem_signal(mutex);
        sem_signal(empty);
    }
    printf("\n");
}
}

```

Output

```
$ gcc pcsem.c
```

```
$ ./a.out
```

```
Enter data for producer to produce : 5
```

```
Enter data for producer to produce : 8 Consumer  
consumes data 5
```

```
Enter data for producer to produce : 4 Consumer  
consumes data 8
```

```
Enter data for producer to produce : 2 Consumer  
consumes data 4
```

```
Enter data for producer to produce : 9 Consumer  
consumes data 2
```

```
Consumer consumes data 9
```

Result

Thus synchronization between producer and consumer process for access to a shared memory segment is implemented.

Operating System Lab No 8

BANKERS ALGORITHM

CLO: 3,6

AIM

To implement deadlock avoidance by using Banker's Algorithm.

ALGORITHM

1. Start the program.
2. Get the values of resources and processes.
3. Get the avail value.
4. After allocation find the need value.
5. Check whether its possible to allocate.
6. If it is possible then the system is in safe state.
7. Else system is not in safety state.
8. If the new request comes then check that the system is in safety or not if we allow the request.
9. Stop.

PROGRAM

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

main()
{
    int r[1][10], av[1][10];
    int all[10][10], max[10][10], ne[10][10], w[10], safe[10]; int i=0, j=0, k=0,
    l=0, np=0, nr=0, count=0, cnt=0;

    clrscr();
    printf("enter the number of processes in a system"); scanf("%d", &np);
    printf("enter the number of resources in a system"); scanf("%d",&nr);
    for(i=1; i<=nr; i++)
    {
        printf("Enter no. of instances of resource R%d ",i); scanf("%d",
        &r[0][i]); av[0][i] = r[0][i];
    }

    for(i=1; i<=np; i++) for(j=1;
        j<=nr; j++)
        all[i][j] = ne[i][j] = max[i][j] = w[i]=0;
```

```

printf("Enter the allocation matrix"); for(i=1; i<=np; i++)
{
    for(j=1; j<=nr; j++)
    {
        scanf("%d", &all[i][j]);
        av[0][j] = av[0][j] - all[i][j];
    }
}

printf("Enter the maximum matrix");
for(i=1; i<=np; i++)
{
    for(j=1; j<=nr; j++)
    {
        scanf("%d",&max[i][j]);
    }
}

for(i=1; i<=np; i++)
{
    for(j=1; j<=nr; j++)
    {
        ne[i][j] = max[i][j] - all[i][j];
    }
}

for(i=1; i<=np; i++)
{
    printf("process P%d", i); for(j=1;
j<=nr; j++)
    {
        printf("\n allocated %d\t",all[i][j]); printf("maximum %d\t",max[i][j]);
        printf("need %d\t",ne[i][j]);
    }
    printf("\n_____ \n");
}

printf("\nAvailability "); for(i=1; i<=nr;
i++)
    printf("R%d %d\t", i, av[0][i]); printf("\n
_____"); printf("\n
safe sequence");

```



```

for(count=1; count<=np; count++)
{
    for(i=1; i<=np; i++)
    {
        Cnt = 0;
        for(j=1; j<=nr; j++)
        {
            if(ne[i][j] <= av[0][j] && w[i]==0) cnt++;
        }
        if(cnt == nr)
        {
            k++;
            safe[k] = i; for(l=1; l<=nr;
            l++)
                av[0][l] = av[0][l] + all[i][l]; printf("\n P%d
            ",safe[k]); printf("\t Availability "); for(l=1;
            l<=nr; l++)
                printf("R%d %d\t", l, av[0][l]); w[i]=1;
        }
    }
}
getch();
}

```

Output

enter the number of processes in a system 3 enter the
number of resources in a system 3

enter no. of instances of resource R1 10 enter no. of
instances of resource R2 7 enter no. of instances of
resource R3 7

Enter the allocation matrix 3 2 1
1 1 2
4 1 2

Enter the maximum matrix 4 4 4
3 4 5
5 2 4

process P1

<hr/>			
process P2			
allocated	1	maximum	3
allocated	1	maximum	4
allocated	2	maximum	5
<hr/>			
process P3			
<hr/>			
Availability R1 2			
safe sequence			
<hr/>			
R2 3			
R3 2			

Result

Thus bankers algorithm for dead lock avoidance was executed successfully.

Operating System Lab No 9 Scheduling

CLO-4

Rubrics for Lab:

Task	0	1	2	3
Implementing the SJF algorithm	Student has no knowledge of FCFS and SJF algorithms.	Student has implemented the FCFS but could not implement the SJF	Student has coded the SJF but could not compile it properly	Student performed the SJF properly and submit the complete result

a) FCFS Scheduling

Aim

To schedule snapshot of processes queued according to FCFS scheduling.

Process Scheduling

- ☐ CPU scheduling is used in multiprogrammed operating systems.
- ☐ By switching CPU among processes, efficiency of the system can be improved.
- ☐ Some scheduling algorithms are FCFS, SJF, Priority, Round-Robin, etc.
- ☐ Gantt chart provides a way of visualizing CPU scheduling and enables to understand better.

First Come First Serve (FCFS)

- ☐ Process that comes first is processed first
- ☐ FCFS scheduling is non-preemptive
- ☐ Not efficient as it results in long average waiting time.
- ☐ Can result in starvation, if processes at beginning of the queue have long bursts.

Algorithm

1. Define an array of structure *process* with members *pid*, *btime*, *wtime* & *ttime*.
2. Get length of the ready queue, i.e., number of process (say *n*)
3. Obtain *btime* for each process.
4. The *wtime* for first process is 0.
5. Compute *wtime* and *ttime* for each process as:
 - a. $wtime_{i+1} = wtime_i + btime_i$
 - b. $ttime_i = wtime_i + btime_i$
6. Compute average waiting time *awat* and average turnaround time *atur*
7. Display the *btime*, *ttime* and *wtime* for each process.
8. Display GANTT chart for the above scheduling
9. Display *awat* time and *atur*
10. Stop

Program

/* FCFS Scheduling - fcfs.c

***/ #include <stdio.h>**

struct process

```
{  
    int pid; int  
    btime; int  
    wtime; int  
    ttime;  
} p[10];
```

main()

```
{  
    int i,j,k,n,ttur,twat; float  
    awat,atur;  
  
    printf("Enter no. of process : "); scanf("%d",  
    &n);  
    for(i=0; i<n; i++)  
    {  
        printf("Burst time for process P%d (in ms) : ",(i+1)); scanf("%d",  
        &p[i].btime);  
        p[i].pid = i+1;  
    }  
  
    p[0].wtime = 0;  
    for(i=0; i<n; i++)  
    {  
        p[i+1].wtime = p[i].wtime + p[i].btime; p[i].ttime =  
        p[i].wtime + p[i].btime;  
    }  
    ttur = twat = 0;  
    for(i=0; i<n; i++)  
    {  
        ttur += p[i].ttime; twat  
        += p[i].wtime;  
    }  
    awat = (float)twat / n; atur  
    = (float)ttur / n;  
  
    printf("\n          FCFS  
    Scheduling\n\n"); for(i=0; i<28; i++)  
        printf("-");  
    printf("\nProcess B-Time T-Time W-Time\n"); for(i=0;  
    i<28; i++)  
        printf("-");
```

```

for(i=0; i<n; i++)
    printf("\n P%d\t%4d\t%3d\t%2d",
           p[i].pid,p[i].btime,p[i].ttime,p[i].wtime);
printf("\n"); for(i=0;
i<28; i++)
    printf("-");

printf("\n\nAverage waiting time          : %5.2fms", awat);
printf("\nAverage turn around time : %5.2fms\n", atur);

printf("\n\nGANTT Chart\n");
printf("-");
for(i=0; i<(p[n-1].ttime + 2*n); i++) printf("-");
printf("\n");
printf("|"); for(i=0;
i<n; i++)
{
    k = p[i].btime/2; for(j=0;
j<k; j++)
    printf(" ");
    printf("P%d",p[i].pid); for(j=k+1;
j<p[i].btime; j++)
    printf(" ");
    printf("|");
}
printf("\n");
printf("-");
for(i=0; i<(p[n-1].ttime + 2*n); i++) printf("-");
printf("\n");
printf("0"); for(i=0; i<n;
i++)
{
    for(j=0; j<p[i].btime; j++) printf("
");
    printf("%2d",p[i].ttime);
}
}

```

Output

Enter no. of process : 4

Burst time for process P1 (in ms) : 10 Burst time
for process P2 (in ms) : 4 Burst time for process P3
(in ms) : 11 Burst time for process P4 (in ms) : 6

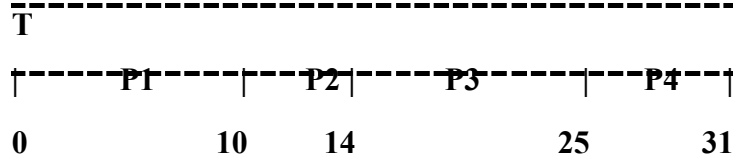
FCFS Scheduling

Process B-Time T-Time W-Time			

allocated 3	maximum need		
	4	1	
allocated 2	maximum need		
	4	2	
allocated 1	maximum need		
	4	3	
allocated 4	maximum 5		need 1
allocated 1	maximum 2		need 1
allocated 2	maximum 4		need 2
P3	Avail abilit y R1 6	R2 4	R3 4
P1	Avail abilit y R1 9	R2 6	R3 5
P2	Ava ilab ility R1 10	R2 7	R3 7
P1	10	10	0
P2	4	14	10
P3	11	25	14
P4	6	31	25

Average waiting time :
12.25ms Average turn around time :
20.00ms

GANT Chart



Result

Thus waiting time & turnaround time for processes based on FCFS scheduling was computed and the average waiting time was determined.

b) SJF Scheduling

Aim

To schedule snapshot of processes queued according to SJF scheduling.

Shortest Job First (SJF)

- ☐ Process that requires smallest burst time is processed first.
- ☐ SJF can be preemptive or non-preemptive
- ☐ When two processes require same amount of CPU utilization, FCFS is used to break the tie.
- ☐ Generally efficient as it results in minimal average waiting time.
- ☐ Can result in starvation, since long critical processes may not be processed.

Algorithm

1. Define an array of structure *process* with members *pid*, *btime*, *wtime* & *ttime*.
2. Get length of the ready queue, i.e., number of process (say *n*)
3. Obtain *btime* for each process.
4. *Sort* the processes according to their *btime* in ascending order.
 - a. If two process have same *btime*, then FCFS is used to resolve the tie.
5. The *wtime* for first process is 0.
6. Compute *wtime* and *ttime* for each process as:
 - a. $wtime_{i+1} = wtime_i + btime_i$
 - b. $ttime_i = wtime_i + btime_i$
7. Compute average waiting time *awat* and average turn around time *atur*.
8. Display *btime*, *ttime* and *wtime* for each process.
9. Display GANTT chart for the above scheduling
10. Display *awat* and *atur*
11. Stop

Program

```
/* SJF Scheduling – sjf.c */ #include
<stdio.h>
struct process
{
    int pid; int
    btime; int
    wtime; int
    ttime;
} p[10], temp;

main()
{
    int i,j,k,n,ttur,twat; float
    awat,atur;

    printf("Enter no. of process : "); scanf("%d",
    &n);
    for(i=0; i<n; i++)
    {
        printf("Burst time for process P%d (in ms) : ",(i+1)); scanf("%d",
        &p[i].btime);
        p[i].pid = i+1;
    }

    for(i=0; i<n-1; i++)
    {
        for(j=i+1; j<n; j++)
        {
            if((p[i].btime > p[j].btime) ||
                (p[i].btime == p[j].btime && p[i].pid > p[j].pid))
            {
                temp = p[i]; p[i]
                = p[j]; p[j] =
                temp;
            }
        }
    }
    p[0].wtime = 0;
    for(i=0; i<n; i++)
    {
        p[i+1].wtime = p[i].wtime + p[i].btime; p[i].ttime =
        p[i].wtime + p[i].btime;
    }
    ttur = twat = 0;
```

```

for(i=0; i<n; i++)
{
    ttur += p[i].ttime; twat
    += p[i].wtime;
}
awat = (float)twat / n; atur
= (float)ttur / n;

printf("\n          SJF Scheduling\n\n");
for(i=0; i<28; i++)
    printf("-");
printf("\nProcess B-Time T-Time W-Time\n"); for(i=0;
i<28; i++)
    printf("-");
for(i=0; i<n; i++)
    printf("\n  P%-4d\t%-4d\t%-3d\t%-2d",
        p[i].pid,p[i].btime,p[i].ttime,p[i].wtime);
printf("\n"); for(i=0;
i<28; i++)
    printf("-");
printf("\n\nAverage waiting time          : %5.2fms", awat);
printf("\nAverage turn around time : %5.2fms\n", atur);

printf("\n\nGANTT Chart\n");
printf("-");
for(i=0; i<(p[n-1].ttime + 2*n); i++) printf("-");
printf("\n|"); for(i=0;
i<n; i++)
{
    k = p[i].btime/2; for(j=0;
j<k; j++)
    printf(" ");
    printf("P%d",p[i].pid); for(j=k+1;
j<p[i].btime; j++)
    printf(" ");
    printf("|");
}
printf("\n-");
for(i=0; i<(p[n-1].ttime + 2*n); i++) printf("-");
printf("\n0"); for(i=0;
i<n; i++)
{
    for(j=0; j<p[i].btime; j++) printf("
");
    printf("%2d",p[i].ttime);
}
}

```

Output

Enter no. of process : 5
Burst time for process P1 (in ms) : 10
Burst time for process P2 (in ms) : 6
Burst time for process P3 (in ms) : 5
Burst time for process P4 (in ms) : 6
Burst time for process P5 (in ms) : 9

SJF Scheduling

Process	B-Time	T-Time	W-Time
P3	5	5	0
P2	6	11	5
P4	6	17	11
P5	9	26	17
P1	10	36	26

Average waiting time : 11.80ms

Average turn around time : 19.00ms

GANT Chart

T									
	P3		P2		P4		P5		P1
0	5	11	17	26	30				

Result

Thus waiting time & turnaround time for processes based on SJF scheduling was computed and the average waiting time was determined.

Operating System Lab No 10 Scheduling-II

CLO: 4

Rubrics for Lab:

Task	0	1	2	3
Implementing the Priority scheduling algorithm	Student don't know the concept of Priority and Round robin algorithm	Student has basic knowledge of the concept used on round robin and Priority	Student implemented the Priority algorithm but could not present the result properly	Implemented the both algorithms properly and presented the result

a) Priority Scheduling

Aim

To schedule snapshot of processes queued according to Priority scheduling.

Priority

- ☐ Process that has higher priority is processed first.
- ☐ Priority can be preemptive or non-preemptive
- ☐ When two processes have same priority, FCFS is used to break the tie.
- ☐ Can result in starvation, since low priority processes may not be processed.

Algorithm

1. Define an array of structure *process* with members *pid*, *btime*, *pri*, *wtime* &

ttime.

2. Get length of the ready queue, i.e., number of process (say n)
3. Obtain *btime* and *pri* for each process.
4. Sort the processes according to their *pri* in ascending order.
 - a. If two process have same *pri*, then FCFS is used to resolve the tie.
5. The *wtime* for first process is 0.
6. Compute *wtime* and *ttime* for each process as:
 - a. $wtime_{i+1} = wtime_i + btime_i$
 - b. $ttime_i = wtime_i + btime_i$
7. Compute average waiting time *awat* and average turn around time *atur*
8. Display the *btime*, *pri*, *ttime* and *wtime* for each process.
9. Display GANTT chart for the above scheduling

10. Display *awat* and *atur*
11. Stop

Program

```
/* Priority Scheduling          - pri.c */

#include <stdio.h>

struct process
{
    int pid; int
    btime; int pri;
    int wtime; int
    ttime;
} p[10], temp;

main()
{
    int i,j,k,n,ttur,twat; float
    awat,atur;

    printf("Enter no. of process : "); scanf("%d", &n);
    for(i=0; i<n; i++)
    {
        printf("Burst time for process P%d (in ms) : ", (i+1)); scanf("%d", &p[i].btime);
        printf("Priority for process P%d : ", (i+1)); scanf("%d", &p[i].pri);
        p[i].pid = i+1;
    }

    for(i=0; i<n-1; i++)
    {
        for(j=i+1; j<n; j++)
        {
            if((p[i].pri > p[j].pri) ||
                (p[i].pri == p[j].pri && p[i].pid > p[j].pid) )
            {
                temp = p[i]; p[i]
                = p[j]; p[j] =
                temp;
            }
        }
    }
    p[0].wttime = 0;
    for(i=0; i<n; i++)
    {
        p[i+1].wttime = p[i].wttime + p[i].btime; p[i].ttime =
        p[i].wttime + p[i].btime;
    }
}
```



```

ttur = twat = 0;
for(i=0; i<n; i++)
{
    ttur += p[i].ttime; twat
    += p[i].wtime;
}
awat = (float)twat / n; atur
= (float)ttur / n;

printf("\n\t Priority Scheduling\n\n"); for(i=0; i<38; i++)
    printf("-");
printf("\nProcess B-Time Priority T-Time
W-Time\n"); for(i=0; i<38; i++)
    printf("-"); for (i=0;
i<n; i++)
    printf("\n          P%-4d\t%-4d\t%-3d\t%-4d\t%-4d",
        p[i].pid,p[i].btime,p[i].pri,p[i].ttime,p[i].wtime)
        ;
printf("\n"); for(i=0;
i<38; i++)
    printf("-");

printf("\n\nAverage waiting time          : %5.2fms",
awat); printf("\nAverage turn around time : %5.2fms\n", atur);

printf("\n\nGANTT Chart\n"); printf("-
");
for(i=0; i<(p[n-1].ttime + 2*n); i++) printf("-");
printf("\n|"); for(i=0; i<n;
i++)
{
    k = p[i].btime/2;
    for(j=0; j<k; j++)
        printf(" ");
    printf("P%d",p[i].pid); for(j=k+1;
j<p[i].btime; j++)
        printf(" ");
    printf("|");
}
printf("\n-");
for(i=0; i<(p[n-1].ttime + 2*n); i++) printf("-");
printf("\n0"); for(i=0; i<n;
i++)
{
    for(j=0; j<p[i].btime; j++) printf(" ");
    printf("%2d",p[i].ttime);
}
}

```

Output

Enter no. of process : 5

Burst time for process P1 (in ms) : 10

Priority for process P1 : 3

Burst time for process P2 (in ms) : 7

Priority for process P2 : 1

Burst time for process P3 (in ms) : 6

Priority for process P3 : 3

Burst time for process P4 (in ms) : 13

Priority for process P4 : 4

Burst time for process P5 (in ms) : 5

Priority for process P5 : 2

Priority Scheduling

Process	B-Time	Priority	T-Time	W-Time
P2	7	1	7	0
P5	5	2	12	7
P1	10	3	22	12
P3	6	3	28	22
P4	13	4	41	28

Average waiting time :

13.80ms Average turn around time :

22.00ms

GANTT Chart

	P2		P5		P1		P3		P4	
0		7		12		22		28		41

Result

Thus waiting time & turnaround time for processes based on Priority scheduling was computed and the average waiting time was determined.

b) Round Robin Scheduling

Aim

To schedule snapshot of processes queued according to Round robin scheduling.

Round Robin

- ☐ All processes are processed one by one as they have arrived, but in rounds.
- ☐ Each process cannot take more than the time slice per round.
- ☐ Round robin is a fair preemptive scheduling algorithm.
- ☐ A process that is yet to complete in a round is preempted after the time slice and put at the end of the queue.
- ☐ When a process is completely processed, it is removed from the queue.

Algorithm

1. Get length of the ready queue, i.e., number of process (say n)
2. Obtain *Burst* time B_i for each processes P_i .
3. Get the *time slice* per round, say TS
4. Determine the number of rounds for each process.
5. The wait time for first process is 0.
6. If $B_i > TS$ then process takes more than one round. Therefore turnaround and waiting time should include the time spent for other remaining processes in the same round.
7. Calculate *average* waiting time and turn around time
8. Display the GANTT chart that includes
 - a. order in which the processes were processed in progression of rounds
 - b. Turnaround time T_i for each process in progression of rounds.
9. Display the *burst* time, *turnaround* time and *wait* time for each process (in order of rounds they were processed).
10. Display *average* wait time and turnaround time
11. Stop

Program

```
/* Round robin scheduling          - rr.c */

#include <stdio.h>

main()
{
    int i,x=-1,k[10],m=0,n,t,s=0;
    int a[50],temp,b[50],p[10],bur[10],bur1[10]; int
    wat[10],tur[10],ttur=0,twat=0,j=0; float
    awat,atur;

    printf("Enter no. of process : "); scanf("%d", &n);
    for(i=0; i<n; i++)
    {
        printf("Burst time for process P%d : ", (i+1)); scanf("%d", &bur[i]);
        bur1[i] = bur[i];
    }
    printf("Enter the time slice (in ms) : "); scanf("%d", &t);

    for(i=0; i<n; i++)
    {
        b[i] = bur[i] / t;
        if((bur[i]%t) != 0)
            b[i] += 1;
        m += b[i];
    }

    printf("\n\t\tRound Robin Scheduling\n");

    printf("\nGANTT Chart\n"); for(i=0; i<m;
    i++)
        printf("_____");
    printf("\n");

    a[0] = 0;
    while(j < m)
    {
        if(x == n-1) x
            = 0;
        else
            x++;
        if(bur[x] >= t)
        {
            bur[x] -= t;
            a[j+1] = a[j] + t;
```

```

        if(b[x] == 1)
        {
            p[s] = x;
            k[s] = a[j+1]; s++;
        }
        j++;
        b[x] -= 1;
        printf("      P%d    |", x+1);
    }
    else if(bur[x] != 0)
    {
        a[j+1] = a[j] + bur[x]; bur[x] =
        0; if(b[x] == 1)
        {
            p[s] = x;
            k[s] = a[j+1]; s++;
        } j++;

        b[x] -= 1;
        printf("      P%d    |", x+1);
    }
}

printf("\n");
for(i=0; i<m; i++)
)
    printf("_____");
printf("\n");

for(j=0; j<=m; j++)
    printf("%d\t", a[j]);

for(i=0; i<n; i++)
{
    for(j=i+1; j<n; j++)
    {
        if(p[i] > p[j])
        {
            temp = p[i]; p[i]
            = p[j]; p[j] =
            temp;

            temp = k[i]; k[i]
            = k[j]; k[j] =
            temp;
        }
    }
}
}

```

```

for(i=0; i<n; i++)
{
    wat[i] = k[i] - bur1[i]; tur[i] = k[i];
}
for(i=0; i<n; i++)
{
    ttur += tur[i]; twat
    += wat[i];
}

printf("\n\n"); for(i=0;
i<30; i++)
    printf("-"); printf("\nProcess\tBurst\tTrnd\tWait\n");
for(i=0; i<30; i++)
    printf("-"); for (i=0;
i<n; i++)
    printf("\nP%-4d\t%-4d\t%-4d\t%-4d", p[i]+1, bur1[i], tur[i], wat[i]);
printf("\n"); for(i=0; i<30;
i++)
    printf("-");

awat = (float)twat / n; atur
= (float)ttur / n;
printf("\n\nAverage waiting time          : %.2f ms",
awat); printf("\nAverage turn around time : %.2f ms\n", atur);
}

```

Output

Enter no. of process : 5

Burst time for process P1 : 10 Burst time for

process P2 : 29 Burst time for process P3 : 3

Burst time for process P4 : 7 Burst time for

process P5 : 12 Enter the time slice (in ms) :

10

Round Robin Scheduling

GANTT Chart

P1		P2		P3		P4		P5		P2		P5		P2	
0	10	20	23	30	40	50	52	61							

Process	Burst	Trnd	Wait
P1	10	10	0
P2	29	61	32
P3	3	23	20
P4	7	30	23
P5	12	52	40

Average waiting time : 23.00

ms Average turn around time : 35.20 ms

Result

Thus waiting time and turnaround time for processes based on Round robin scheduling was computed and the average waiting time was determined.

Operating System Lab No 11

Paging

CLO:1,6

Rubrics for Lab:

Task	0	1	2	3
Implement and understand the FIFO and LRU page replacement algorithm	Student has no knowledge of the page replacement	Student has basic knowledge of the page replace but don't know how to implement	<ul style="list-style-type: none"> Student performed one of the page replacement algorithm properly 	Student performed the both page replacement algorithms properly.

Paging

a) Paging Technique

Aim To determine physical address of a given page using page table.

Algorithm

1. Get process size
2. Compute no. of pages available and display it
3. Get relative address
4. Determine the corresponding page
5. Display page table
6. Display the physical address

Program

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

main()
{
    int size, m, n, pgno, pagetable[3]={5,6,7}, i, j, frameno; double m1;
    int ra=0, ofs;

    printf("Enter process size (in KB of max 12KB):"); scanf("%d", &size);
    m1 = size / 4; n =
    ceil(m1);
    printf("Total No. of pages: %d", n); printf("\nEnter relative address
    (in hexa) \n"); scanf("%d", &ra);

    pgno = ra / 1000; ofs = ra
    % 1000;
    printf("page no=%d\n", pgno);
    printf("page table"); for(i=0;i<n;i++)
        printf("\n %d [%d]", i, pagetable[i]); frameno =
    pagetable[pgno];
    printf("\nPhysical address: %d%d", frameno, ofs);
}
```

Output

Enter process size (in KB of max 12KB):12 Total No. of
pages: 3

Enter relative address (in hexa): 2643

page no=2 page

table

0 [5]

1 [6]

2 [7]

Physical address : 7643

Result

Thus physical address for the given logical address is determining using Paging technique.

b) FIFO Page Replacement

Aim: To implement demand paging for a reference string using FIFO method.

FIFO

- ☐ Page replacement is based on when the page was brought into memory.
- ☐ When a page should be replaced, the oldest one is chosen.
- ☐ Generally, implemented using a FIFO queue.
- ☐ Simple to implement, but not efficient.
- ☐ Results in more page faults.
- ☐ The page-fault may increase, even if frame size is increased (Belady's anomaly)

Algorithm

1. Get length of the reference string, say l .
2. Get reference string and store it in an array, say rs .
3. Get number of frames, say nf .
4. Initialize $frame$ array upto length nf to -1.
5. Initialize position of the oldest page, say j to 0.
6. Initialize no. of page faults, say $count$ to 0.
7. For each page in reference string in the given order, examine:
 - a. Check whether page exist in the $frame$ array
 - b. If it does not exist then
 - i. Replace page in position j .
 - ii. Compute page replacement position as $(j+1)$ modulus nf .
 - iii. Increment $count$ by 1.
 - iv. Display pages in $frame$ array.
8. Print $count$.
9. Stop

Program

```
#include <stdio.h> main()
{
    int i,j,l,rs[50],frame[10],nf,k,avail,count=0;

    printf("Enter length of ref. string : "); scanf("%d", &l);
    printf("Enter reference string :\n"); for(i=1; i<=l; i++)
        scanf("%d", &rs[i]);
    printf("Enter number of frames : "); scanf("%d", &nf);
    for(i=0; i<nf; i++) frame[i] = -1;
    j = 0;
    printf("\nRef. str      Page frames"); for(i=1; i<=l; i++)
```

```

{
    printf("\n%4d\t", rs[i]); avail = 0;
    for(k=0; k<nf; k++) if(frame[k]
        == rs[i])
        avail = 1;
    if(avail == 0)
    {
        frame[j] = rs[i]; j =
        (j+1) % nf;
        count++;
        for(k=0; k<nf; k++)
            printf("%4d", frame[k]);
    }
}
printf("\n\nTotal no. of page faults : %d\n",count);
}

```

Output

Enter length of ref. string : 20 Enter reference
string :
1 2 3 4 2 1 5 6 2 1 2 3 7 6 3
Enter number of frames : 5 Ref. str

Page frames

Total no. of page faults : 10
Result

Thus page replacement was implemented using FIFO algorithm.

c)LRU Page Replacement

Aim

To implement demand paging for a reference string using LRU method.

LRU

- ☐ Pages used in the recent past are used as an approximation of future usage.
- ☐ The page that has not been used for a longer period of time is replaced.
- ☐ LRU is efficient but not optimal.
- ☐ Implementation of LRU requires hardware support, such as counters/stack.

Algorithm

1. Get length of the reference string, say *len*.
2. Get reference string and store it in an array, say *rs*.
3. Get number of frames, say *nf*.
4. Create *access* array to store counter that indicates a measure of recent usage.
5. Create a function *arrmin* that returns position of minimum of the given array.
6. Initialize *frame* array upto length *nf* to -1.
7. Initialize position of the page replacement, say *j* to 0.
8. Initialize *freq* to 0 to track page frequency
9. Initialize no. of page faults, say *count* to 0.
10. For each page in reference string in the given order, examine:
 - a. Check whether page exist in the *frame* array.
 - b. If page exist in memory then
 - i. Store incremented *freq* for that page position in *access* array.

c. If
page
does
not
exist
in
memory
then

- i. Check for any empty frames.
 - ii. If there is an empty frame,
 - ☐ Assign that frame to the page
 - ☐ Store incremented *freq* for that page position in *access* array.
 - ☐ Increment *count*.
 - iii. If there is no free frame then
 - ☐ Determine page to be replaced using *arrmin* function.
 - ☐ Store incremented *freq* for that page position in *access* array.
 - ☐ Increment *count*.
 - iv. Display pages in *frame* array.

11. Print *count*.
12. Sto

p Program

```
/* LRU page replacement - lru.c */ #include
```

```
<stdio.h>
```

```
int arrmin(int[], int);
```

```

main()
{
    int i,j,len,rs[50],frame[10],nf,k,avail,count=0; int access[10],
    freq=0, dm;

    printf("Length of Reference string : "); scanf("%d", &len);
    printf("Enter reference string :\n"); for(i=1; i<=len; i++)
        scanf("%d", &rs[i]); printf("Enter no.
    of frames : "); scanf("%d", &nf);

    for(i=0; i<nf;
        i++) frame[i]
        = -1;
    j = 0;

    printf("\nRef. str          Page
    frames"); for(i=1; i<=len; i++)
    {
        printf("\n%4d\t", rs[i]); avail = 0;
        for(k=0; k<nf; k++)
        {
            if(frame[k] == rs[i])
            {
                avail = 1; access[k] =
                ++freq; break;
            }
        }
        if(avail == 0)
        {
            dm = 0;
            for(k=0; k<nf; k++)
            {
                if(frame[k] == -1)
                {
                    dm = 1;
                    break;
                }
            }
            if(dm == 1)
            {
                frame[k] = rs[i]; access[k] =
                ++freq; count++;
            }

            else
            {
                j =
                arrmi
                n(acce
                ss,
                nf);
                frame

```

```
[j] = ++fre
rs[i]; q;
access count
[j] = ++;

    for(k=0; k<nf; k++)
        printf("%4d", frame[k]);
    }
}
```



```

        printf("\n\nTotal no. of page faults : %d\n", count);
    }

int arrmin(int a[], int n)
{
    int i, min = a[0]; for(i=1;
        i<n; i++) if (min > a[i])
            min = a[i];
    for(i=0; i<n; i++)
        if (min == a[i])
            return i;
}

```

Output

Length of Reference string : 15 Enter reference
string :

Total no. of page faults : 8

Result

Thus page replacement was implemented using LRU algorithm

Operating System

Lab No 12

Memory Management Technique

CLO: 6

Rubrics for Lab:

Task	0	1	2	3
Implementation and understanding of MVT and MFT	Student don't know the concept of mft and mvt	Student know the concept of MFT and MVT but don't know the implementation structure	Student implemented one management technique.	Student implemented the both techniques properly
Implementation and understanding the Best fit, worst fit and First fit contiguous memory techniques	Student don't have idea about the Best fit, Worst fit and First fit	Student know the concept but don't know the implementation structure	Student well awarded about the concept and implemented one of the three techniques	Student implemented the best fit, first fit and worst fit and presented the result properly

OBJECTIVE

Memory Management Techniques

Write a C program to simulate the MVT and MFT memory management techniques

DESCRIPTION

MFT (Multiprogramming with a Fixed number of Tasks) is one of the old memory management techniques in which the memory is partitioned into fixed size partitions and each job is assigned to a partition. The memory assigned to a partition does not change. MVT (Multiprogramming with a Variable number of Tasks) is the memory management technique in which each job gets just the amount of memory it needs. That is, the partitioning of memory is dynamic and changes as jobs enter and leave the system. MVT is a more "efficient" use of resources. MFT suffers with the problem of internal fragmentation and MVT suffers with external fragmentation.

PROGRAM

MFT MEMORY MANAGEMENT TECHNIQUE

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

```
main()
{
    int ms, bs, nob, ef,n, mp[10],tif=0; int
    i,p=0;
```

```
clrscr();
printf("Enter the total memory available (in Bytes) -- ");
scanf("%d",&ms);
printf("Enter the block size (in Bytes) -- ");
scanf("%d", &bs);
nob=ms/bs; ef=ms -
nob*bs;
printf("\nEnter the number of processes -- ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
    printf("Enter memory required for process %d (in Bytes)--",i+1); scanf("%d",&mp[i]);
}

printf("\nNo. of Blocks available in memory -- %d",nob);
printf("\n\nPROCESS\tMEMORY REQUIRED\tALLOCATED\tINTERNAL FRAGMENTATION");
for(i=0;i<n && p<nob;i++)
{
    printf("\n %d\t\t%d",i+1,mp[i]);
    if(mp[i] > bs)
        printf("\t\tNO\t\t---");
    else
    {
        p
        r
        i
        n
        t
        f
        (
        "
        )
    }
}
```

```
f          -          i
+          m          ]
b          p          ;
s          [          p++;
```

```
printf("\nMemory is Full, Remaining Processes cannot be accomodated");
```

```
printf("\n\nTotal Internal Fragmentation is %d",tif);
printf("\nTotal External Fragmentation is %d",ef);
getch();
```

}

INPUT

Enter the total memory available (in Bytes) -- 1000
Enter the block size (in Bytes)-- 300
Enter the number of processes -- 5
Enter memory required for process 1 (in Bytes) -- 275
Enter memory required for process 2 (in Bytes) -- 400
Enter memory required for process 3 (in Bytes) -- 290
Enter memory required for process 4 (in Bytes) -- 293
Enter memory required for process 5 (in Bytes) -- 100

No. of Blocks available in memory -- 3

OUTPUT

PROCESS REQUIRED	MEMORY	ALLOCATED	INTERNAL FRAGMENTATION
1	275	YES	25
2	400	NO	----
3	290	YES	10
4	293	YES	7

Memory is Full, Remaining Processes cannot be accommodated

Total Internal Fragmentation is 42

Total External Fragmentation is 100

MVT MEMORY MANAGEMENT TECHNIQUE

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

```
#include<conio.h>
```

```
main()
```

```
{  
    int ms,mp[10],i, temp,n=0;  
    char ch = 'y';
```

```
    clrscr();  
    printf("\nEnter the total memory available (in Bytes)-- ");  
    scanf("%d",&ms);  
    temp=ms;  
    for(i=0;ch=='y';i++,n++)  
    {
```

```
        printf("\nEnter memory required for process %d (in Bytes) --  
        ",i+1); scanf("%d",&mp[i]);  
        if(mp[i]<=temp)  
        {
```

```
            printf("\nMe  
            mory is  
            allocated for  
            Process %d  
            ",i+1); temp =  
            temp - mp[i];
```

```
        }  
    }  
    else  
    {  
  
    }  
}
```

```
p  
r  
i  
n
```

t
f
(
"
\
n
M
e
m
o
r
y
i
s
F
u
l
l
"
)
;
b
r
e
a
k
;

```

        printf("\nDo you want to continue(y/n) -- ");
        scanf(" %c", &ch);
    }
    printf("\n\nTotal Memory Available -- %d", ms);

    printf("\n\n\tPROCESS\t\tMEMORY ALLOCATED ");
    for(i=0;i<n;i++)
        printf("\n \t%d\t\t%d", i+1, mp[i]);
    printf("\n\nTotal Memory Allocated is %d", ms-temp);
    printf("\nTotal External Fragmentation is %d", temp);
    getch();
}

```

INPUT

Enter the total memory available (in Bytes) -- 1000

Enter memory required for process 1 (in Bytes) -- 400

Memory is allocated for Process 1

Do you want to continue(y/n) -- y

Enter memory required for process 2 (in Bytes) -- 275

Memory is allocated for Process 2

Do you want to continue(y/n) -- y

Enter memory required for process 3 (in Bytes) -- 550

OUTPUT

Memory is Full

Total Memory Available -- 1000

PROCESS	MEMORY ALLOCATED
---------	------------------

1	400
---	-----

2	275
---	-----

Total Memory Allocated is 675

Total External Fragmentation is 325

(B) Contiguous memory allocation techniques

OBJECTIVE

*Write a C program to simulate the following contiguous memory allocation techniques

- a) Worst-fit b) Best-fit c) First-fit

DESCRIPTION

One of the simplest methods for memory allocation is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. In this multiple-partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process. The operating system keeps a table indicating which parts of memory are available and which are occupied. Finally, when a process arrives and needs memory, a memory section large enough for this process is provided. When it is time to load or swap a process into main memory, and if there is more than one free block of memory of sufficient size, then the operating system must decide which free block to allocate. Best-fit strategy chooses the block that is closest in size to the request. First-fit chooses the first available block that is large enough. Worst-fit chooses the largest available block.

PROGRAM

WORST-FIT

```
#include<stdio.h>
#include<conio.h>
#define max 25

void main()
{
    int frag[max],b[max],f[max],i,j,nb,nf,temp;
    static int bf[max],ff[max];
    clrscr();

    printf("\n\tMemory Management Scheme - First Fit");
    printf("\nEnter the number of blocks:");
    scanf("%d",&nb);
    printf("Enter the number of files:");
    scanf("%d",&nf);
    printf("\nEnter the size of the blocks:-\n");
    for(i=1;i<=nb;i++)
    {
        printf("Block %d:",i);
        scanf("%d",&b[i]);
    }
    printf("Enter the size of the files :-\n");
    for(i=1;i<=nf;i++)
    {
        printf("File %d:",i);
        scanf("%d",&f[i]);
    }
    for(i=1;i<=nf;i++)
    {
        for(j=1;j<=nb;j++)
        {
            if(bf[j]!=1)
            {
                temp=b[j]-f[i];
                if(temp>=0)
                {
                    ff[i]=j;
                    ;
                    break;
                }
            }
        }
    }
}
```

} }

```

frag[i]=temp; bf[ff[i]]=1;
    }
    printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");
    for(i=1;i<=nf;i++)
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
    getch();
}

```

INPUT

Enter the number of blocks: 3

Enter the number of files: 2

Enter the size of the blocks:-

Block 1: 5

Block 2: 2

Block 3: 7

Enter the size of the files:-

File 1: 1

File 2: 4

OUTPUT

File No	File Size	Block No	Block Size	Fragment
1	1	1	5	4
2	4	3	7	3

BEST-FIT

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

```
#include<conio
```

```
.h> #define
```

```
max 25
```

```
void main()
```

```
{
```

```
    int frag[max],b[max],f[max],i,j,nb,nf,temp,lowest=10000;
```

```
    static int bf[max],ff[max];
```

```
    clrscr();
```

```
    printf("\nEnter the number of blocks:");
```

```
    scanf("%d",&nb);
```

```
    printf("Enter the number of files:");
```

```
    scanf("%d",&nf);
```

```
    printf("\nEnter the size of the blocks:-\n");
```

```
    for(i=1;i<=nb;i++)
```

```
        printf("Block %d:",i);scanf("%d",&b[i]);
```

```
    printf("Enter the size of the files :-\n");
```

```
    for(i=1;i<=nf;i++)
```

```
    {
```

```
        printf("File %d:",i);
```

```
        scanf("%d",&f[i]);
```

```
    }
```

```
    for(i=1;i<=nf;i++)
```

```
    {
```

```
        for(j=1;j<=nb;j++)
```

```
        {
```

```
            if(bf[j]!=1)
```

```
            {
```

```
                temp=b[j]-f[i];
```

```
                if(temp>=0)
```

```
                    if(lowest>temp)
```

```
{  
    ff[i]=j;  
    lowest=temp;
```

```

    }
    }
    frag[i]=lowest;
    bf[ff[i]]=1;
    lowest=10000;
}
printf("\nFile No\tFile Size \tBlock No\tBlock
Size\tFragment"); for(i=1;i<=nf && ff[i]!=0;i++)
    printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
getch();
}

```

INPUT

Enter the number of blocks: 3

Enter the number of files: 2

Enter the size of the blocks:-

Block 1: 5

Block 2: 2

Block 3: 7

Enter the size of the files:-

File 1: 1

File 2: 4

OUTPUT

File No	File Size	Block No	Block Size	Fragment
1	1	2	2	1
2	4	1	5	1

FIRST-FIT

```

#include<stdio. h>
#include<conio
.h> #define
max 25

```

```

void main()
{
    int frag[max],b[max],f[max],i,j,nb,nf,temp,highest=0;
    static int bf[max],ff[max];
    clrscr();

    printf("\n\tMemory Management Scheme - Worst Fit");
    printf("\nEnter the number of blocks:"); scanf("%d",&nb);
    printf("Enter the number of files:");
    scanf("%d",&nf);
    printf("\nEnter the size of the blocks:-\n");
    for(i=1;i<=nb;i++)
    {
        printf("Block %d:",i);
        scanf("%d",&b[i]);
    }
    printf("Enter the size of the files :-\n");
    for(i=1;i<=nf;i++)
    {
        printf("File %d:",i);
        scanf("%d",&f[i]);
    }
    for(i=1;i<=nf;i++)

```

```
{  
    for(j=1;j<=nb;j++)
```

```

        {
            if(bf[j]!=1) //if bf[j] is not allocated
            {
                temp=b[j]-f[i];
                if(temp>=0)
                    if(highest<temp)
                    {
                        ff[i]=j;
                        highest=temp;
                    }
            }
        }
        frag[i]=highest;
        bf[ff[i]]=1;
        highest=0;
    }

    printf("\nFile_no:\tFile_size :\tBlock_no:\tBlock_size:\tFragement");
    for(i=1;i<=nf;i++)
        printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d",i,f[i],ff[i],b[ff[i]],frag[i]);
    getch();
}

```

INPUT

Enter the number of blocks: 3

Enter the number of files: 2

Enter the size of the blocks:-

Block 1: 5

Block 2: 2

Block 3: 7

Enter the size of the files:-

File 1: 1

File 2: 4

OUTPUT

File No	File Size	Block No	Block Size	Fragment
1	1	3	7	6
2	4	1	5	1

Operating System mLab No 13

CLO: 5,6

Rubrics for Lab:

File Organization

Task	0	1	2	3
Implementing the single level and 2-level directory	Student don't know about the concept of directories	Student know the concept but don't know the implementation techniques	Student implemented the single level directory on C	Student implemented the single level and two level directory and presented the result properly

File Organization

a) Single-Level Directory

Aim

To organize files in a single level directory structure, I.e., without sub-directories.

Algorithm

1. Get name of directory for the user to store all the files
2. Display menu

3. Accept choice
4. If choice =1 then
 - Accept filename without any collision
 - Store it in the directory
5. If choice =2 then
 - Accept filename
 - Remove filename from the directory array
6. If choice =3 then
 - Accept filename
 - Check for existence of file in the directory array
7. If choice =4 then
 - List all files in the directory array
8. If choice =5
 - then Stop

Program

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

```
@include <stdlib.h>
#include <conio.h>
```

```
struct
{
    char dname[10]; char
    fname[25][10]; int fcnt;
}dir;
```

```
main()
{
    int i, ch; char
    f[30]; clrscr();
    dir.fcnt = 0;
    printf("\nEnter name of directory -- "); scanf("%s",
    dir.dname);

    while(1)
    {
        printf("\n\n 1. Create File\t2. Delete File\t3. Search File \n4. Display Files\t5.
        Exit\nEnter your choice--"); scanf("%d",&ch);
```

```

switch(ch)
{
    case 1:
        printf("\n Enter the name of the file -- "); scanf("%s",
        dir.fname[dir.fcnt]); dir.fcnt++;
        break;

    case 2:
        printf("\n Enter the name of the file -- "); scanf("%s", f);
        for(i=0; i<dir.fcnt; i++)
        {
            if(strcmp(f, dir.fname[i]) == 0)
            {
                printf("File %s is deleted ",f); strcpy(dir.fname[i],
                dir.fname[dir.fcnt-1]); break;
            }
        }
        if(I == dir.fcnt)
            printf("File %s not found", f); else
            dir.fcnt--; break;

    case 3:
        printf("\n Enter the name of the file -- "); scanf("%s", f);
        for(i=0; i<dir.fcnt; i++)
        {
            if(strcmp(f, dir.fname[i]) == 0)
            {
                printf("File %s is found ", f); break;
            }
        }
        if(I == dir.fcnt)
            printf("File %s not found", f); break;

    case 4:
        if(dir.fcnt == 0)
            printf("\n Directory Empty"); else
        {
            printf("\n The Files are -- "); for(i=0;
            i<dir.fcnt; i++)
                printf("\t%s", dir.fname[i]);
        }
        break;
}

```

```

        default:
            exit(0);
    }
}
getch();
}

```

Output

Enter name of directory -- CSE

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit

Enter your choice -- 1

Enter the name of the file -- fcfs

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit

Enter your choice -- 1

Enter the name of the file -- sjf

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit

Enter your choice -- 1

Enter the name of the file -- lru

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit

Enter your choice -- 3

Enter the name of the file --

sjf File sjf is found

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit

Enter your choice -- 3

Enter the name of the file -- bank

File bank is not found

1. Create File 2. Delete File 3. Search File
4. Display Files 5. Exit

Enter your choice -- 4

The Files are -- fcfs sjf lru bank

1. Create File 2. Delete File 3. Search File

4. Display Files 5.

Exit Enter your choice

-- 2

Enter the name of the file -- lru

File lru is deleted

Result

Thus files were organized into a single level directory.

b) Two-Level Directory

Aim

To organize files as two-level directory with each user having his own user file directory (UFD).

Algorithm

1. Display menu
2. Accept choice
3. If choice =1 then
 Accept directory name
 Create an entry for that directory
4. If choice =2 then
 Get directory name
 If directory exist then accept filename without collision else report error
5. If choice =3 then
 Get directory name
 If directory exist then Get filename
 If file exist in that directory then delete entry else report error
6. If choice =4 then
 Get directory name
 If directory exist then Get filename
 If file exist in that directory then Display filename else report error
7. If choice =5 then Display files directory-wise
8. If choice =6 then

Stop Program

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

```
struct
{
    char dname[10], fname[10][10]; int fcnt;
}dir[10];
```

```
main()
{
    int i, ch, dcnt, k;
    char f[30], d[30];
    clrscr();    dcnt=0;
    while(1)
    {
        printf("\n\n 1. Create Directory\t 2. Create File\t 3.
Delete File");
        printf("\n 4. Search File \t \t 5. Display \t 6. Exit \n Enter your choice -- ");
```

```

scanf("%d", &ch); switch(ch)
{
    case 1:
        printf("\n Enter name of directory -- "); scanf("%s",
        dir[dcnt].dname); dir[dcnt].fcnt = 0;
        dcnt++;
        printf("Directory created"); break;

    case 2:
        printf("\n Enter name of the directory -- "); scanf("%s", d);
        for(i=0; i<dcnt; i++) if(strcmp(d,dir[i].dname)
        == 0)
        {
            printf("Enter name of the file -- "); scanf("%s",
            dir[i].fname[dir[i].fcnt]); dir[i].fcnt++;
            printf("File created"); break;
        }
        if(i == dcnt)
            printf("Directory %s not found",d); break;

    case 3:
        printf("\nEnter name of the directory -- "); scanf("%s", d);
        for(i=0; i<dcnt; i++)
        {
            if(strcmp(d,dir[i].dname) == 0)
            {
                printf("Enter name of the file -- "); scanf("%s", f);
                for(k=0; k<dir[i].fcnt; k++)
                {
                    if(strcmp(f, dir[i].fname[k]) == 0)
                    {
                        printf("File %s is deleted ", f); dir[i].fcnt--;
                        strcpy(dir[i].fname[k], dir[i].fname[dir[i].fcnt]);
                        goto jmp;
                    }
                }
                printf("File %s not found",f); goto jmp;
            }
        }
}

```

```
printf("Directory %s not found",d); jmp : break;
```

```
case 4:
```

```
printf("\nEnter name of the directory -- "); scanf("%s", d);
for(i=0; i<dcnt; i++)
{
    if(strcmp(d,dir[i].dname) == 0)
    {
        printf("Enter the name of the file -- "); scanf("%s", f);
        for(k=0; k<dir[i].fcnt; k++)
        {
            if(strcmp(f, dir[i].fname[k]) == 0)
            {
                printf("File %s is found ", f); goto jmp1;
            }
        }
        printf("File %s not found", f); goto jmp1;
    }
}
printf("Directory %s not found", d); jmp1: break;
```

```
case 5:
```

```
if(dcnt == 0)
    printf("\nNo Directory's "); else
{
    printf("\nDirectory\tFiles"); for(i=0;i<dcnt;i++)
    {
        printf("\n%s\t\t",dir[i].dname); for(k=0;k<dir[i].fcnt;k++)
        printf("\t%s",dir[i].fname[k]);
    }
}
break;
```

```
default:
```

```
    exit(0);
```

```
    }
```

```
}
```

```
getch();
```

```
}
```


Output

1. Create Directory 2. Create File 3. Delete File 4. Search File 5. Display 6. Exit Enter your choice -- 1

Enter name of directory -- CSE Directory created

1. Create Directory 2. Create File 3. Delete File 4. Search File 5. Display 6. Exit Enter your choice -- 1

Enter name of directory -- ECE Directory created

1. Create Directory 2. Create File 3. Delete File 4. Search File 5. Display 6. Exit Enter your choice -- 2

Enter name of the directory -- ECE Enter name of the file -- amruth File created

1. Create Directory 2. Create File 3. Delete File 4. Search File 5. Display 6. Exit Enter your choice -- 2

Enter name of the directory -- CSE Enter name of the file -- kowshik File created

1. Create Directory 2. Create File 3. Delete File 4. Search File 5. Display 6. Exit Enter your choice -- 2

Enter name of the directory -- CSE Enter name of the file -- pranesh File created

1. Create Directory 2. Create File 3. Delete File 4. Search File 5. Display 6. Exit Enter your choice -- 2

Enter name of the directory -- ECE Enter name of the file -- ajith
File created

1. Create Directory 2. Create File 3. Delete File 4. Search File 5. Display 6. Exit Enter your choice -- 5

Directory Files

CSE kowshik pranesh

ECE amruth ajith

1. Create Directory 2. Create File 3. Delete File 4. Search File 5. Display 6. Exit Enter your choice -- 3

Enter name of the directory -- ECE Enter name of the file -- ajith
File ajith is deleted

Result

Thus user files have been stored in their respective directories and retrieved easily

