Preface

This lab manual is designed for a one-semester course in Operating System. The prerequisite
for students using this manual is a one-semester course in Algorithm and Data structure.
Before this course, the students also learnt C programming language.

This manual is designed to focus students learning over the modern operating system concepts. We used Ubuntu and example Operating system.

This Lab manual covers Linux introduction, BASH basics and advance scripting. System call and all the other algorithm visualization is done in C programming language.

Table of Contents

Lab No 01.a

Ubuntu Installation

Objective: Is to understand the process of Linux installation.

Scope: Linux distribution Ubuntu 18.04 installation.

Task:

Step 1) Download Ubuntu 18.04 LTS ISO File

Please make sure you have the latest version of Ubuntu 18.04 LTS, If not, please download the ISO file from the link here

https://www.ubuntu.com/download/desktop

Since Ubuntu 18.04 LTS only comes in a 64-bit edition, so you can install it on a system that supports 64-bit architecture.

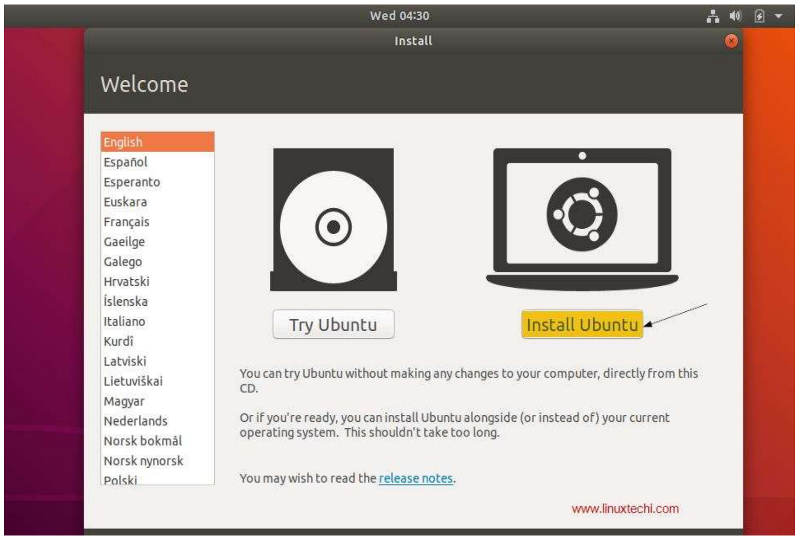
Step 2) Create a Bootable Disk

Once the ISO file is downloaded then next step is to burn the downloaded ISO image into the USB/DVD or flash drive to boot the computer from that drive.

Also make sure you change the boot sequence so that system boots using the bootable CD/DVD or flash drive.

Step 3) Boot from USB/DVD or Flash Drive

Once the system is booted using the bootable disk, you can see the following screen presented before you with options including "**Try Ubuntu**" and "**Install Ubuntu**" as shown in the image below,



You can try Ubuntu without making any changes to your computer, directly from this CD.

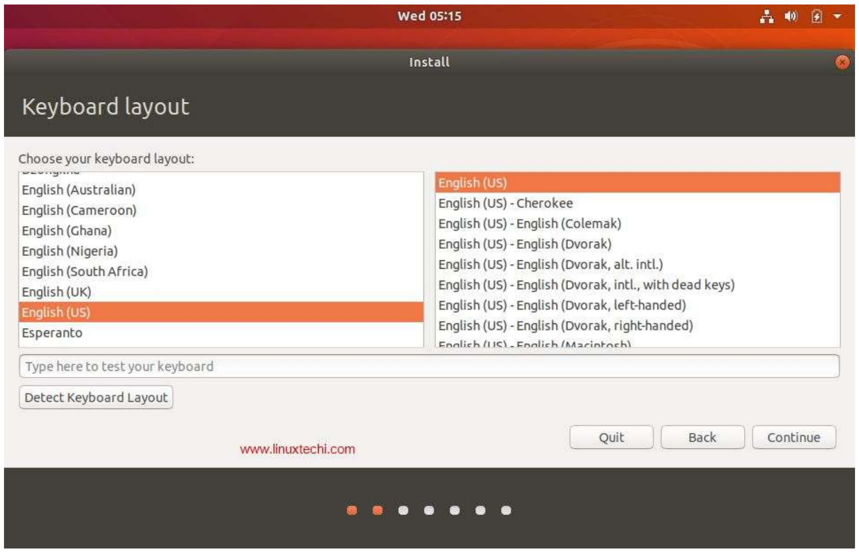
Or if you're ready, you can install Ubuntu alongside (or instead of) your current operating system. This shouldn't take too long.

You may wish to read the release notes.

Even though when you click "Try Ubuntu" you can have a sneak peek into the 18.04 LTS without installing it in your system, our goal here is to install Ubuntu 18.04 LTS in your system. So click "**Install Ubuntu**" to continue with the installation process.

Step 4) Choose your Keyboard layout

Choose your favorite keyboard layout and click "Continue". By default English (US) keyboard is selected and if you want to change, you can change here and click "Continue",


Wed 05:15
Install
Keyboard layout
English (Australian)
English (Cameroon)
English (Ghana)
English (Nigeria)
English (South Africa)
English (UK)
English (US)
Esperanto
English (US)
English (US) - Cherokee
English (US) - English (Colemak)
English (US) - English (Dvorak)
English (US) - English (Dvorak, alt. intl.)
English (US) - English (Dvorak, intl., with dead keys)
English (US) - English (Dvorak, left-handed)
English (US) - English (Dvorak, right-handed)
English (US) - English (Macintosh)
Type here to test your keyboard
Detect Keyboard Layout
Quit
Back
Continue
www.linuxtechi.com

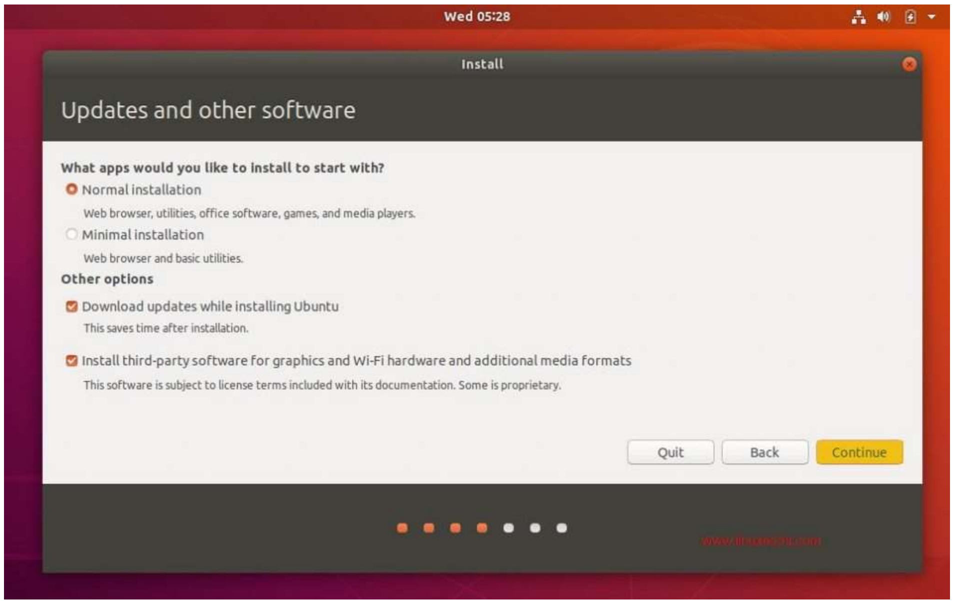
Choose your keyboard layout:

Step 5) Preparing to Install Ubuntu and other Software

In the next screen, you'll be provided following beneath options including:

Type of Installation:

- Normal Installation or Minimal installation, If you want a minimal installation then select second option otherwise go for the Normal Installation. In my case I am doing Normal Installation
- Download Updates While Installing Ubuntu (select this option if your system has internet connectivity during installation)
- Install third party software for graphics and Wi-Fi hardware, MP3 and additional media formats Select this option if your system has internet connectivity)



Updates and other software

What apps would you like to install to start with?

[✓ Normal installation]
Web browser, utilities, office software, games, and media players.

Minimal installation
Web browser and basic utilities.

Other options

[✓ Download updates while installing Ubuntu]
This saves time after installation.

[✓ Install third-party software for graphics and Wi-Fi hardware and additional media formats]
This software is subject to license terms included with its documentation. Some is proprietary.

click on "**Continue**" to proceed with installation

Step 6) Select the appropriate Installation Type

Next the installer presents you with the following installation options including:

- Erase Disk and Install Ubuntu
- Encrypt the new Ubuntu installation for security
- Use LVM with the new Ubuntu installation
- Something Else

Where,

Erase Disk and Install Ubuntu – Choose this option if your system is going to have only Ubuntu and erasing anything other than that is not a problem. This ensures a fresh copy of Ubuntu 18.04 LTS is installed in your system.

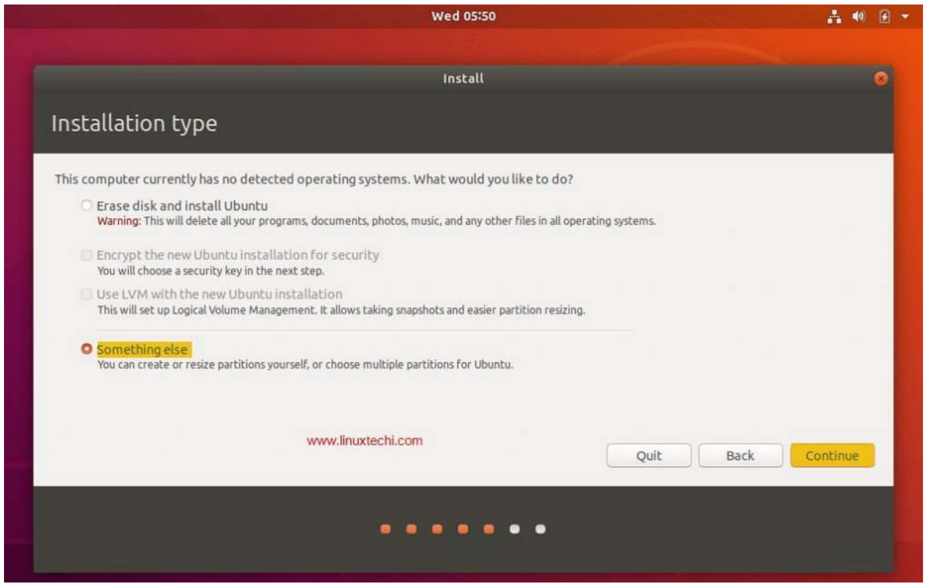
Encrypt the new Ubuntu installation for security – Choose this option if you are looking for extended security for your disks as your disks will be completely encrypted. If you are beginner, then it is better not to worry about this option.

Use LVM with the new Ubuntu installation – Choose this option if you want to use LVM based file systems.

Something Else – Choose this option if you are advanced user and you want to manually create your own partitions and want to install Ubuntu along with existing OS (May be Windows or other Linux Flavor)

In this article, we will be creating our custom partitions on a hard disk of 40 GB and the following partitions are to be created:

/boot	1 GB (ext4 files system)
/home	18 GB (ext4 file system)
/	12 GB (ext4 file system)
/var	6 GB (ext4 file system)
Swap	2 GB


Wed 05:50

Install

Installation type

This computer currently has no detected operating systems. What would you like to do?

Erase disk and install Ubuntu
Warning: This will delete all your programs, documents, photos, music, and any other files in all operating systems.

Encrypt the new Ubuntu installation for security
You will choose a security key in the next step.

Use LVM with the new Ubuntu installation
This will set up Logical Volume Management. It allows taking snapshots and easier partition resizing.

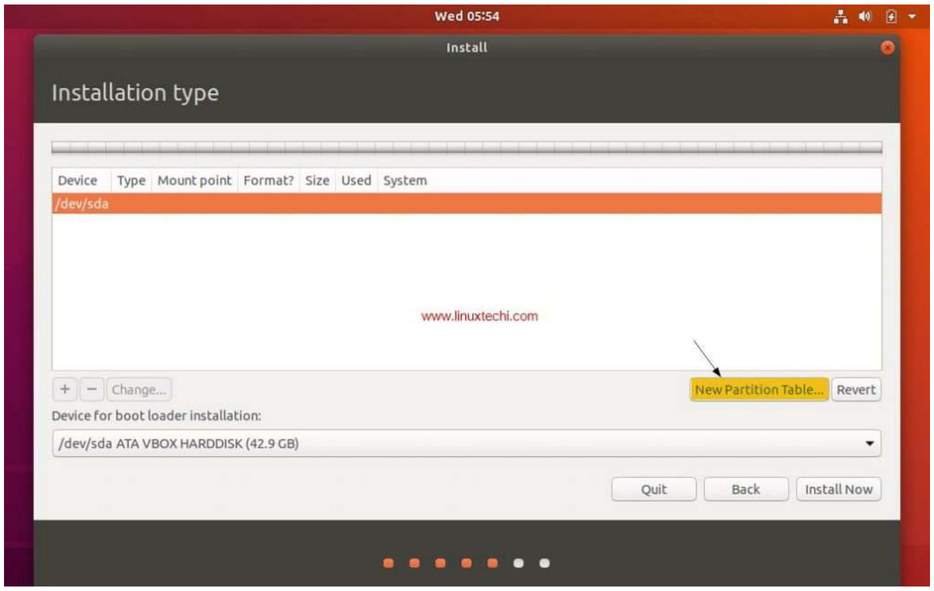
Something else
You can create or resize partitions yourself, or choose multiple partitions for Ubuntu.

www.linuxtechi.com

Now, Choose “**Something Else**” and Click on continue

You can see the available disk size for Ubuntu in the next window as shown below:

Now in order to create your own partitions, click on “**New Partition Table**”


Wed 05:54

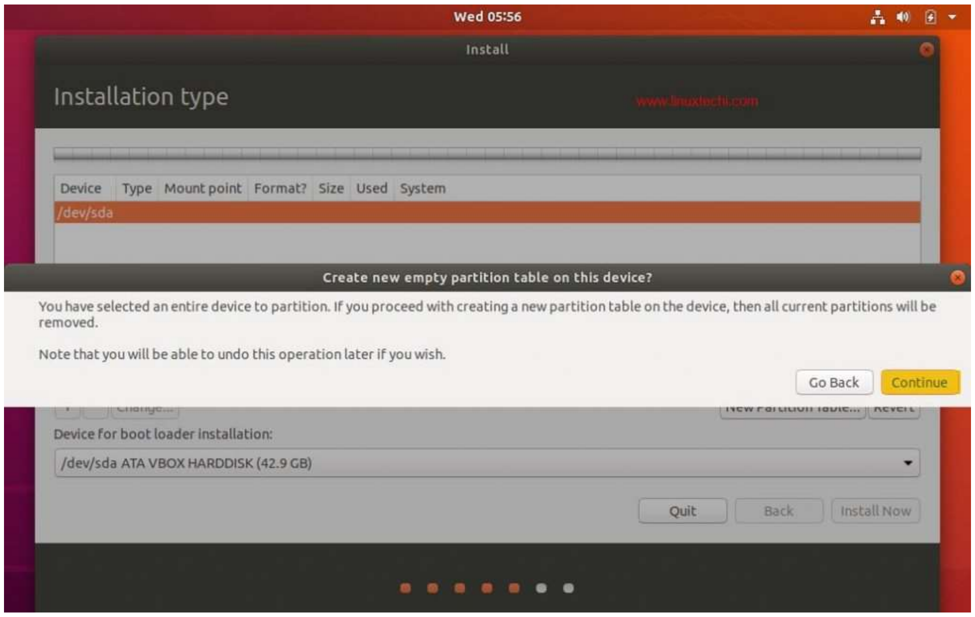
Install

Installation type

Device	Type	Mount point	Format?	Size	Used	System
/dev/sda						

www.linuxtechi.com

Device for boot loader installation:
/dev/sda ATA VBOX HARDDISK (42.9 GB)

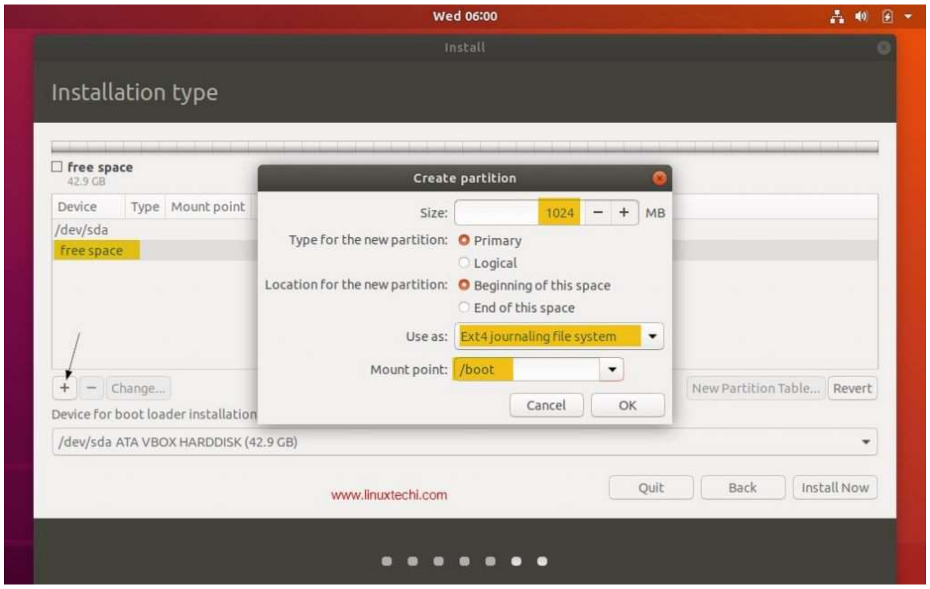


You have selected an entire device to partition. If you proceed with creating a new partition table on the device, then all current partitions will be removed.

Note that you will be able to undo this operation later if you wish.

Click on Continue

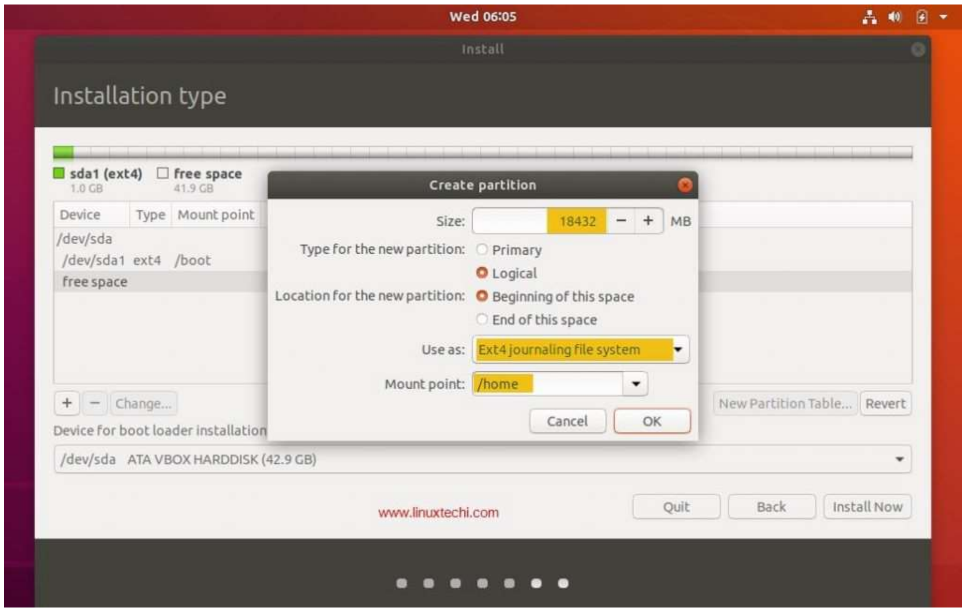
Create /boot partition of size 1GB, Select the free space and then Click on the "+" symbol to create a new partition



Size: 1024 MB
Type for the new partition: [✓ Primary] Logical
Location for the new partition: [✓ Beginning of this space] End of this space
Use as: Ext4 journaling file system
Mount point: /boot

Click on "OK"

Let's create /home partition of size 18 GB,



🟩 sda1 (ext4) 1.0 GB ☐ free space 41.9 GB

Device	Type	Mount point
/dev/sda		
/dev/sda1	ext4	/boot
free space		

Create partition

Size: 18432 MB

Type for the new partition: Primary [✓ Logical]

Location for the new partition: [✓ Beginning of this space] End of this space

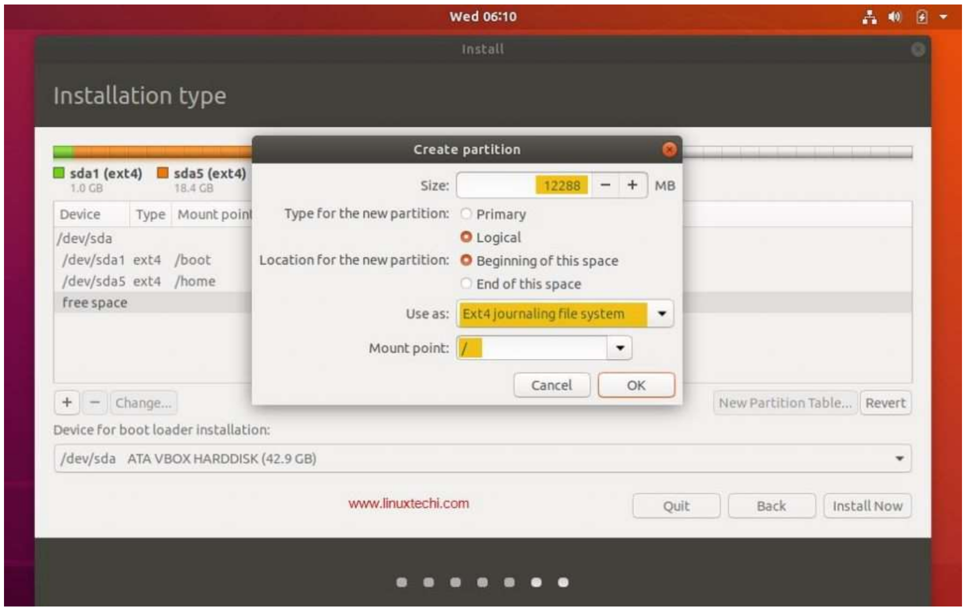
Use as: Ext4 journaling file system

Mount point: /home

Device for boot loader installation:

/dev/sda ATA VBOX HARDDISK (42.9 GB)

In the same way create / & /var file system of size 12 GB & 6 GB respectively



🟩 sda1 (ext4) 1.0 GB 🟧 sda5 (ext4) 18.4 GB

Device	Type	Mount point
/dev/sda		
/dev/sda1	ext4	/boot
/dev/sda5	ext4	/home
free space		

Create partition

Size: 12288 MB

Type for the new partition: Primary [✓ Logical]

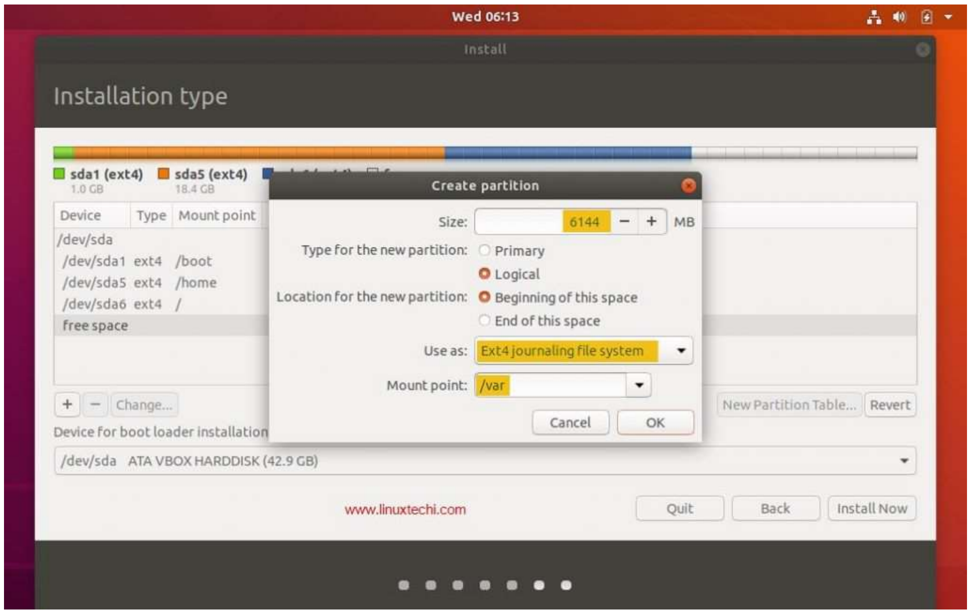
Location for the new partition: [✓ Beginning of this space] End of this space

Use as: Ext4 journaling file system

Mount point: /

Device for boot loader installation:

/dev/sda ATA VBOX HARDDISK (42.9 GB)



Device	Type	Mount point
/dev/sda		
/dev/sda1	ext4	/boot
/dev/sda5	ext4	/home
/dev/sda6	ext4	/
free space		

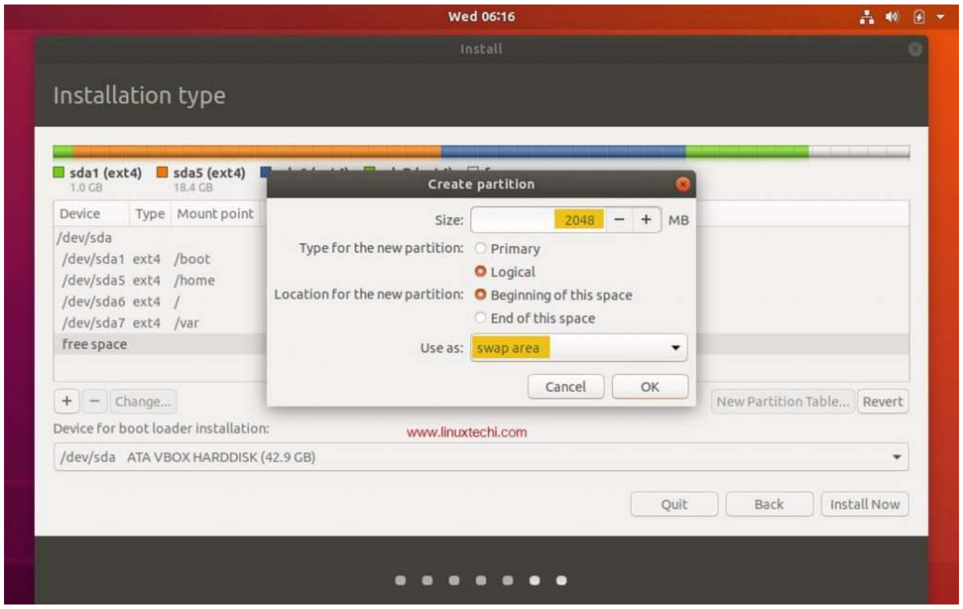
Create partition

Size: 6144 MB
Type for the new partition: Primary [✓ Logical]
Location for the new partition: [✓ Beginning of this space] End of this space
Use as: Ext4 journaling file system
Mount point: /var

Cancel OK

Device for boot loader installation: /dev/sda ATA VBOX HARDDISK (42.9 GB)

Now create last partition as swap of size 2 GB,



Device	Type	Mount point
/dev/sda		
/dev/sda1	ext4	/boot
/dev/sda5	ext4	/home
/dev/sda6	ext4	/
/dev/sda7	ext4	/var
free space		

Create partition

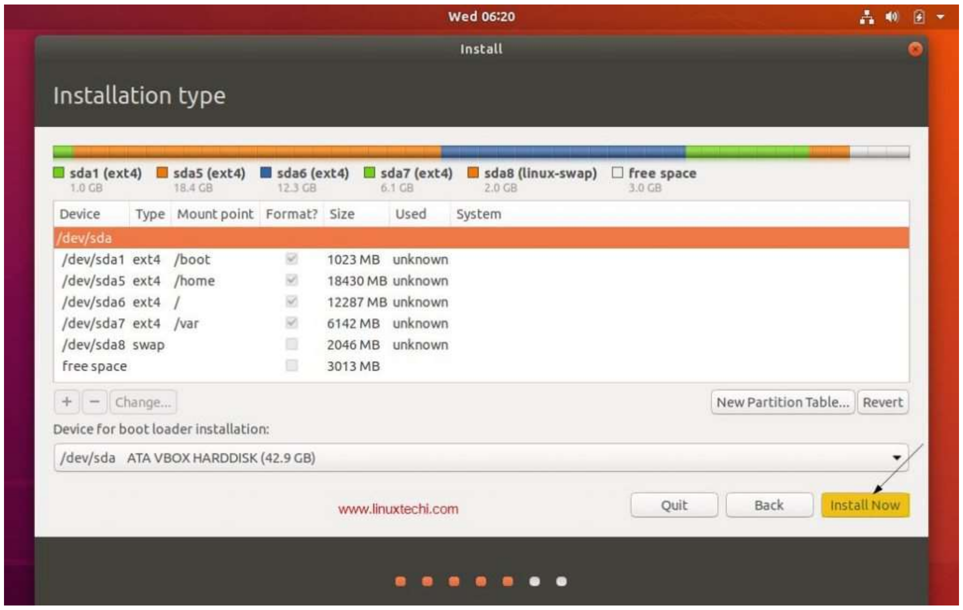
Size: 2048 MB
Type for the new partition: Primary [✓ Logical]
Location for the new partition: [✓ Beginning of this space] End of this space
Use as: swap area

Cancel OK

Device for boot loader installation: /dev/sda ATA VBOX HARDDISK (42.9 GB)

Click on OK

Once you are done with the partition creation task , then click on “**Install Now**” option to proceed with the installation



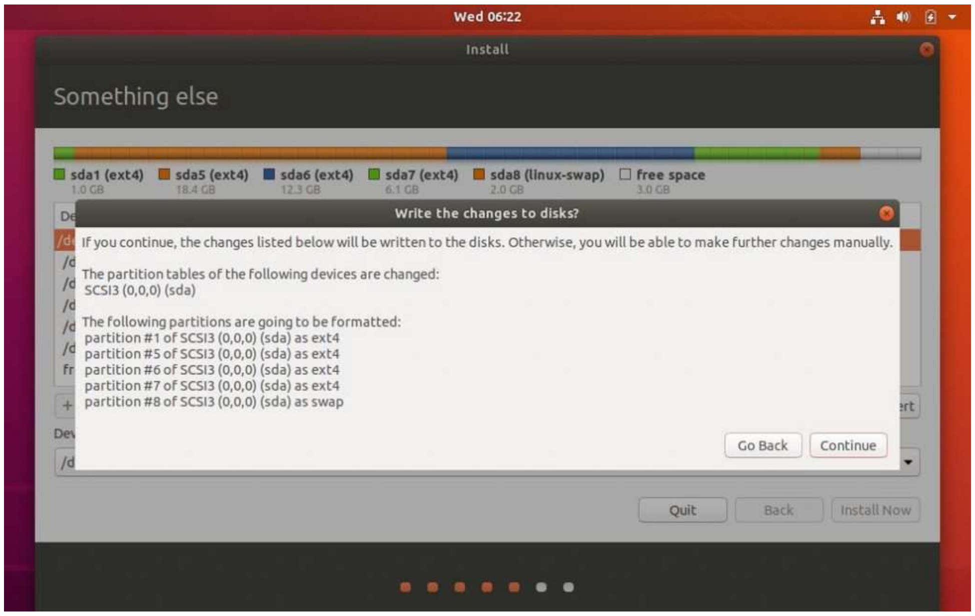
Installation type

🟩 sda1 (ext4) 1.0 GB 🟧 sda5 (ext4) 18.4 GB 🟦 sda6 (ext4) 12.3 GB 🟩 sda7 (ext4) 6.1 GB 🟧 sda8 (linux-swap) 2.0 GB ☐ free space 3.0 GB

Device	Type	Mount point	Format?	Size	Used	System
/dev/sda						
/dev/sda1	ext4	/boot	✓	1023 MB	unknown	
/dev/sda5	ext4	/home	✓	18430 MB	unknown	
/dev/sda6	ext4	/	✓	12287 MB	unknown	
/dev/sda7	ext4	/var	✓	6142 MB	unknown	
/dev/sda8	swap		☐	2046 MB	unknown	
free space			☐	3013 MB		

Device for boot loader installation:

/dev/sda ATA VBOX HARDDISK (42.9 GB)



Something else

Write the changes to disks?

If you continue, the changes listed below will be written to the disks. Otherwise, you will be able to make further changes manually.

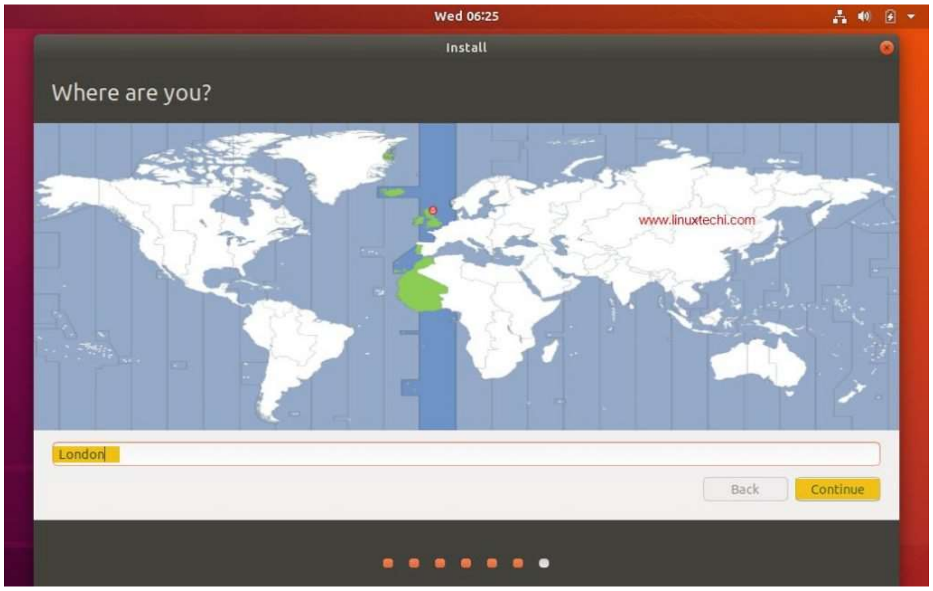
The partition tables of the following devices are changed:
SCSI3 (0,0,0) (sda)

The following partitions are going to be formatted:
partition #1 of SCSI3 (0,0,0) (sda) as ext4
partition #5 of SCSI3 (0,0,0) (sda) as ext4
partition #6 of SCSI3 (0,0,0) (sda) as ext4
partition #7 of SCSI3 (0,0,0) (sda) as ext4
partition #8 of SCSI3 (0,0,0) (sda) as swap

Now click on “Continue” to write all the changes to the disks

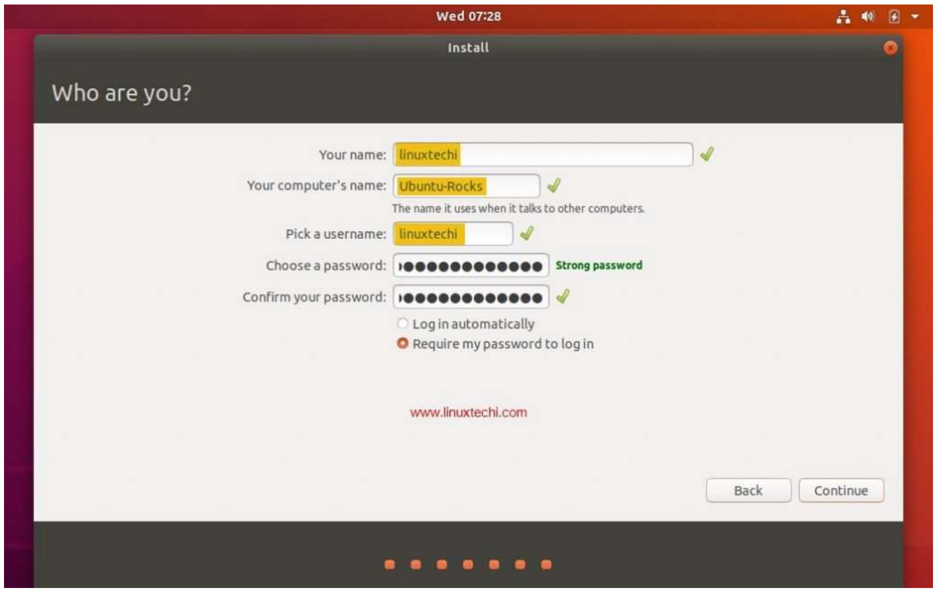
Step 7) Select Your Time zone

Choose your favorite time zone and then click on “Continue”



Step 8) Provide your User Credentials

In the next screen you will be prompted to provide your user credentials. In this screen provide your name, computer name, username and the password to login into Ubuntu 18.04 LTS

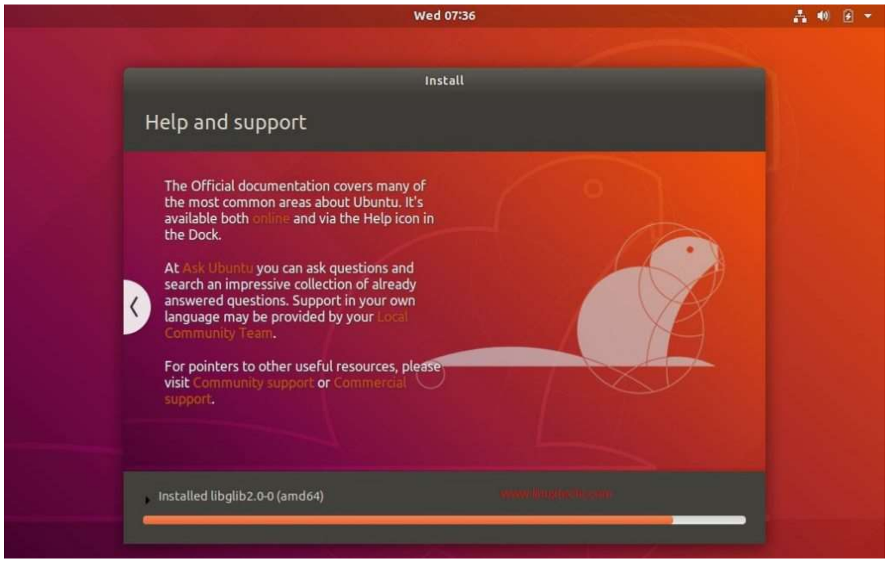


Your name: linuxtechi
Your computer's name: Ubuntu-Rocks
The name it uses when it talks to other computers.
Pick a username: linuxtechi
Choose a password: ●●●●●●●●●●●●● Strong password
Confirm your password: ●●●●●●●●●●●●●
Log in automatically
[✓ Require my password to log in]

Click “Continue” to begin the installation process.

Step 9) Start Installing Ubuntu 18.04 LTS

The installation of Ubuntu 18.04 LTS starts now and will take around 5-10 mins depending on the speed of your computer,


Wed 07:36

Install

Help and support

The Official documentation covers many of the most common areas about Ubuntu. It's available both online and via the Help icon in the Dock.

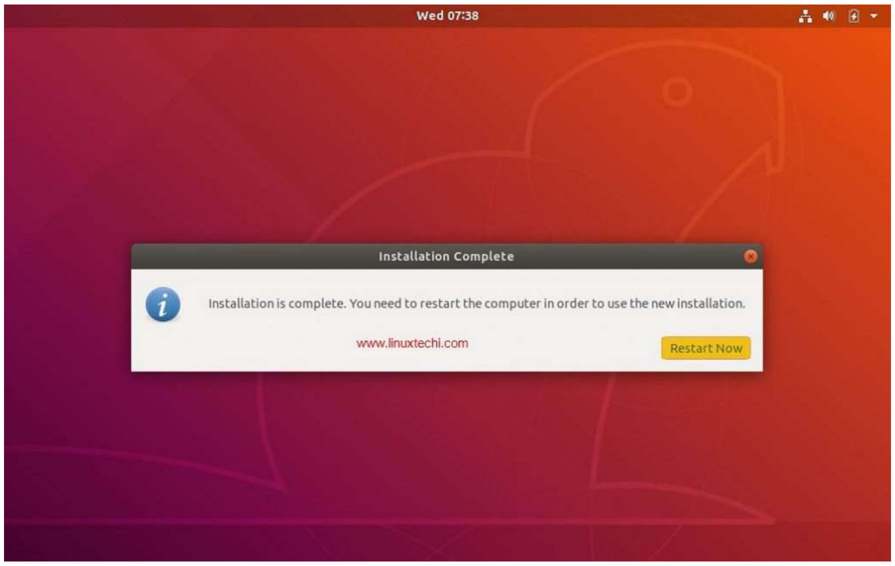
At Ask Ubuntu you can ask questions and search an impressive collection of already answered questions. Support in your own language may be provided by your Local Community Team.

For pointers to other useful resources, please visit Community support or Commercial support.

Installed libglib2.0-0 (amd64)

Step 10) Restart Your System

Once the installation is completed, remove the USB/DVD from the drive and Click “Restart Now” to restart your system.


Wed 07:38

Installation Complete

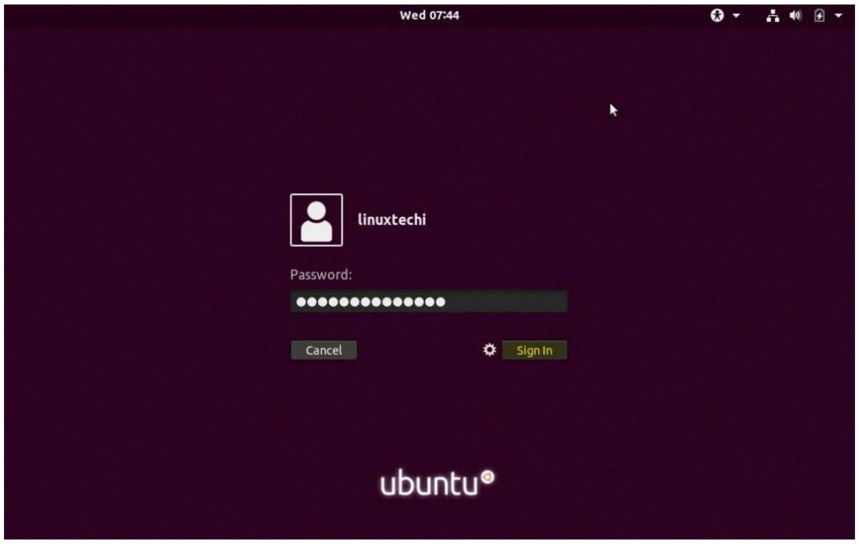
Installation is complete. You need to restart the computer in order to use the new installation.

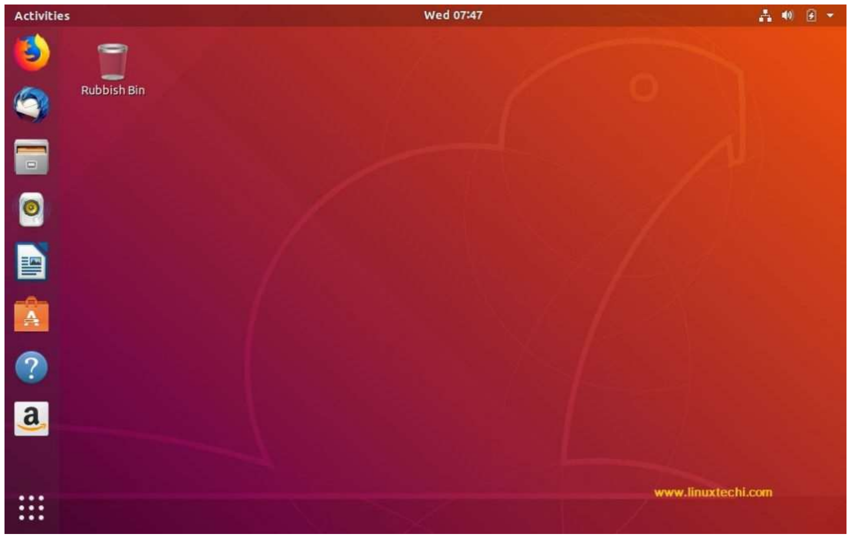
www.linuxtechi.com

Read More on : How to Install VirtualBox 6.0 on Ubuntu 18.04 LTS / 18.10 / CentOS 7

Step:11) Login to Your Ubuntu 18.04 desktop

Once your system has been rebooted after the installation then you will get the beneath login screen, enter the User name and password that you have set during installation (Step 8)


Wed 07:44
linuxtechi
Password:
Cancel
Sign In
ubuntu


Activities
Wed 07:47
Rubbish Bin
www.linuxtechi.com

Lab No 01.b

BASH Commands

Objective:
This Lab demonstrates different useful BASH Commands
Scope:
Linux BASH commands

SYSTEM INFORMATION

```
# Display Linux system information
uname -a
# Display kernel release information
uname -r
# Show which version of redhat installed
cat /etc/redhat-release
# Show how long the system has been running + load
uptime
# Show system host name
hostname
# Display the IP addresses of the host
hostname -I
# Show system reboot history
last reboot
# Show the current date and time
date
# Show this month's calendar
cal
# Display who is online
w
# Who you are logged in as
Whoami
```

HARDWARE INFORMATION

```
# Display messages in kernel ring buffer
dmesg
# Display CPU information
cat /proc/cpuinfo
# Display memory information
cat /proc/meminfo
# Display free and used memory ( -h for human readable, -m for MB, -g for GB.)
free -h
# Show info about disk sda
hdparm -i /dev/sda
```

PERFORMANCE MONITORING AND STATISTICS

```
# Display and manage the top processes
top
# Interactive process viewer (top alternative)
htop
# Display processor related statistics
mpstat 1
# Display virtual memory statistics
vmstat 1
```

USER INFORMATION AND MANAGEMENT

```
# Display the user and group ids of your current user.
id
# Display the last users who have logged onto the system.
last
# Show who is logged into the system.
```

```
who
# Show who is logged in and what they are doing.
w
# Create a group named "test".
groupadd test
# Create an account named john, with a comment of "John Smith" and create the user's home
directory.
useradd -c "John Smith" -m john
# Delete the john account.
userdel john
# Add the john account to the sales group
usermod -aG sales john
```

FILE AND DIRECTORY COMMANDS

```
# List all files in a long listing (detailed) format
ls -al
# Display the present working directory
pwd
# Create a directory
mkdir directory
# Remove (delete) file
rm file
# Remove the directory and its contents recursively
rm -r directory
# Force removal of file without prompting for confirmation
rm -f file
# Forcefully remove directory recursively
rm -rf directory
# Copy file1 to file2
cp file1 file2
# Copy source_directory recursively to destination. If destination exists, copy source_directory into
destination, otherwise create destination with the contents of source_directory.
cp -r source_directory destination
# Rename or move file1 to file2. If file2 is an existing directory, move file1 into directory file2
mv file1 file2
# Create symbolic link to linkname
ln -s /path/to/file linkname
# Create an empty file or update the access and modification times of file.
touch file
# View the contents of file
cat file
# Browse through a text file
less file
# Display the first 10 lines of file
head file
# Display the last 10 lines of file
tail file
# Display the last 10 lines of file and "follow" the file as it grows.
tail -f file
```

PROCESS MANAGEMENT

```
# Display your currently running processes
ps
# Display all the currently running processes on the system.
ps -ef
```

NETWORKING

```
# Display all network interfaces and ip address
ifconfig -a
```

```
# Display eth0 address and details
ifconfig eth0
# Query or control network driver and hardware settings
ethtool eth0
# Send ICMP echo request to host
ping host
# Display whois information for domain
whois domain
# Display DNS information for domain
dig domain
```

DISK USAGE

```
# Show free and used space on mounted filesystems
df -h
# Show free and used inodes on mounted filesystems
df -i
# Display disks partitions sizes and types
fdisk -l
# Display disk usage for all files and directories in human readable format
du -ah
# Display total disk usage off the current directory
du -sh
```

DIRECTORY NAVIGATION

```
# To go up one level of the directory tree.  (Change into the parent directory.)
cd ..

# Go to the $HOME directory
cd
# Change to the /etc directory
cd /etc
```

Homework:

1. Install Linux mint 19.x or Ubuntu 18.x on your laptop or desktop computer
2. Apply all the commands on Ubuntu Terminal

Lab No 02.a

BASH Commands with arguments and parameters

Objective:

This Lab is very targeted to help you with BASH commands using arguments and parameters

Scope:

Linux BASH

1.Date Command :

This command is used to display the current data and time.

Syntax :

```
$date
$date +%ch
```

Options :

a	= Abbrevated weekday.
A	= Full weekday.
b	= Abbrevated month.
B	= Full month.
c	= Current day and time.
C	= Display the century as a decimal number.
d	= Day of the month.
D	= Day in „mm/dd/yy" format
h	= Abbrevated month day.
H	= Display the hour.
L	= Day of the year.
m	= Month of the year.
M	= Minute.
P	= Display AM or PM
S	= Seconds
T	= HH:MM:SS format
u	= Week of the year.
y	= Display the year in 2 digit.
Y	= Display the full year.
Z	= Time zone .

To change the format :

Syntax :

```
$date „+%H-%M-%S"
```

2.Calender Command :

This command is used to display the calendar of the year or the particular month of calendar year.

Syntax :

```
a.$cal <year>
b.$cal <month> <year>
```

Here the first syntax gives the entire calendar for given year & the second Syntax gives the calendar of reserved month of that year.

3.Echo Command :

This command is used to print the arguments on the screen .

Syntax : `$echo <text>`

4.Banner Command :

It is used to display the arguments in „#" symbol .

Syntax : $banner <arguments>

5.'who' Command :

It is used to display who are the users connected to our computer currently.

Syntax : $who – option"s

Options : -

H–Display the output with headers.
b–Display the last booting date or time or when the system was lastely rebooted.

6.'who am i' Command :

Display the details of the current working directory.

Syntax : $who am i

9.'CLEAR' Command :

It is used to clear the screen.

Syntax : $clear

10.'MAN' Command :
It help us to know about the particular command and its options & working. It is like „help" command in windows .

Syntax : $man <command name>

11.LIST Command :

It is used to list all the contents in the current working directory.

Syntax : $ ls – options <arguments>
If the command does not contain any argument means it is working in the Current directory.

Options :
a– used to list all the files including the hidden files.
c– list all the files columnwise.
d- list all the directories.
m- list the files separated by commas.
p- list files include „/" to all the directories.
r- list the files in reverse alphabetical order.
f- list the files based on the list modification date.
x-list in column wise sorted order.

DIRECTORY RELATED COMMANDS :

1.Present Working Directory Command :

To print the complete path of the current working directory.

Syntax : $pwd

2.MKDIR Command :

To create or make a new directory in a current directory .

Syntax : $mkdir <directory name>

3.CD Command :

To change or move the directory to the mentioned directory .

Syntax : $cd <directory name.

4.RMDIR Command :

To remove a directory in the current directory & not the current directory itself.
Syntax : $rmdir <directory name>

FILE RELATED COMMANDS :

1.CREATE A FILE :

To create a new file in the current directory we use CAT command.

Syntax : $cat > <filename.
The > symbol is redirectory we use cat command.

2.DISPLAY A FILE :

To display the content of file mentioned we use CAT command without „>" operator.

Syntax : $cat <filename.

Options –s = to neglect the warning /error message.

3.COPYING CONTENTS :

To copy the content of one file with another. If file doesnot exist, a new file is created and if the file exists with some data then it is overwritten.

Syntax : $ cat <filename source> >> <destination filename>
$ cat <source filename> >> <destination filename> it is avoid overwriting.

Options :

-n content of file with numbers included with blank lines.

Syntax :
$cat –n <filename>

4.SORTING A FILE :
To sort the contents in alphabetical order in reverse order.

Syntax :
$sort <filename >

Option : $ sort –r <filename>

5.COPYING CONTENTS FROM ONE FILE TO ANOTHER :

To copy the contents from source to destination file . so that both contents are same.

Syntax :

```
$cp <source filename> <destination filename>
$cp <source filename path > <destination filename path>
```

6.MOVE Command :

To completely move the contents from source file to destination file and to remove the source file.

Syntax :

```
$ mv <source filename> <destination filename>
```

7.REMOVE Command :

To permanently remove the file we use this command .

Syntax :

```
$rm <filename>
```

8.WORD Command :

To list the content count of no of lines , words, characters .

Syntax :

```
$wc<filename>
```

Options :

-c – to display no of characters.
-l – to display only the lines.
-w – to display the no of words.

FILTERS AND PIPES

HEAD : It is used to display the top ten lines of file.

Syntax: $head<filename>

TAIL : This command is used to display the last ten lines of file.

Syntax: $tail<filename>

PAGE : This command shows the page by page a screen full of information is displayed after which the page command displays a prompt and passes for the user to strike the enter key to continue scrolling.

Syntax: $ls –a\p

MORE : It also displays the file page by page .To continue scrolling with more command , press the space bar key.

Syntax: $more<filename>

SORT : This command is used to sort the data in some order.

Syntax: $sort<filename>

PIPE : It is a mechanism by which the output of one command can be channelled into the input of another command.

Syntax: $who | wc-l

TR :The tr filter is used to translate one set of characters from the standard inputs to another.

Syntax: $tr “[a-z]” “[A-Z]”

Lab No 02.b

Linux Directory Structure

Objective:

Objective of this lab is to demonstrate directory structure of Linux system

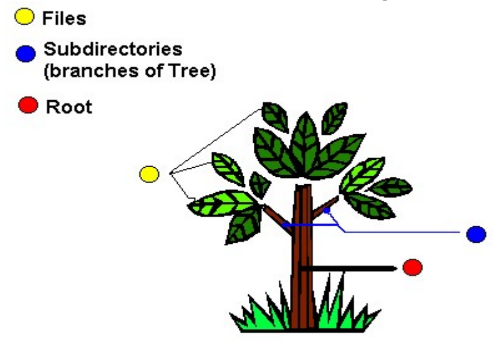
Scope:

Linux system

A file system is a logical collection of files on a partition or disk. A partition is a container for information and can span an entire hard drive if desired.

Directory Structure

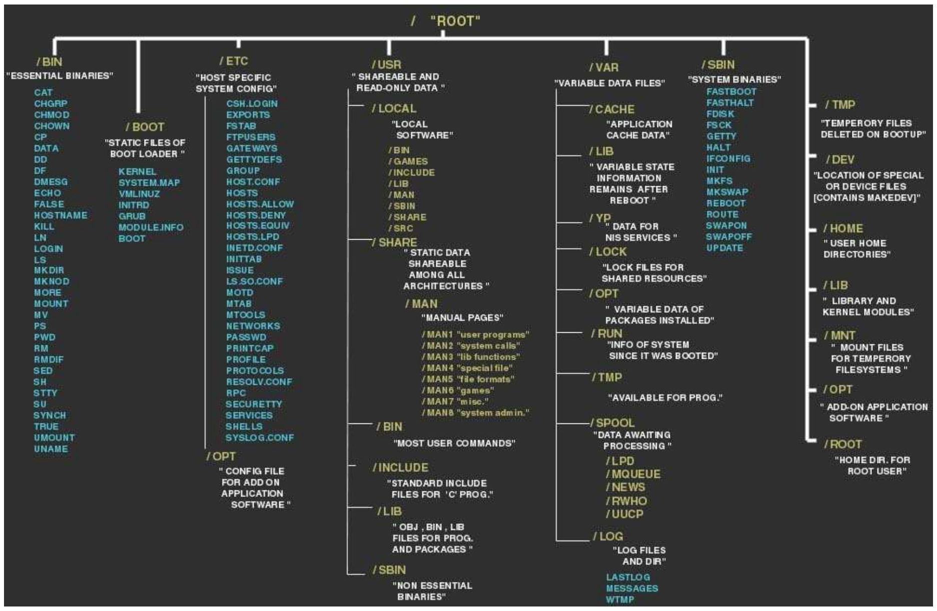
Unix uses a hierarchical file system structure, much like an upside-down tree, with root (/) at the base of the file system and all other directories spreading from there.



Linux File System is just like a tree

/	This is the root directory which should contain only the directories needed at the top level of the file structure Everything on your Linux system is located under the / directory, known as the root directory. You can think of the / directory as being similar to the C:\ directory on Windows – but this isn't strictly true, as Linux doesn't have drive letters.
/bin	This is where the executable files are located. These files are available to all users. The /bin directory contains the essential user binaries (programs) that must be present when the system is mounted in single-user mode. Applications such as Firefox are stored in /usr/bin, while important system programs and utilities such as the bash shell are located in /bin.
/dev	These are device drivers Linux exposes devices as files, and the /dev directory contains a number of special files that represent devices. These are not actual files as we know them, but they appear as files – for example, /dev/sda represents the first SATA drive in the system. If you wanted to partition it, you could start a partition editor and tell it to edit /dev/sda.
/boot	The /boot directory contains the files needed to boot the system – for example, the GRUB boot loader's files and your Linux kernels are stored here. The boot loader's configuration files aren't located here, though – they're in /etc with the other configuration files.
/etc	The /etc directory contains configuration files, which can generally be edited by hand in a text editor. Note that the /etc/ directory contains system-wide configuration files – user-specific configuration files are located in each user's home directory.

/home	The /home directory contains a home folder for each user. For example, if your user name is bob, you have a home folder located at /home/bob. This home folder contains the user's data files and user-specific configuration files. Each user only has write access to their own home folder and must obtain elevated permissions (become the root user) to modify other files on the system.
/lib	The /lib directory contains libraries needed by the essential binaries in the /bin and /sbin folder. Libraries needed by the binaries in the /usr/bin folder are located in /usr/lib.
/lost+found	Each Linux file system has a lost+found directory. If the file system crashes, a file system check will be performed at next boot. Any corrupted files found will be placed in the lost+found directory, so you can attempt to recover as much data as possible.
/media	The /media directory contains subdirectories where removable media devices inserted into the computer are mounted. For example, when you insert a CD into your Linux system, a directory will automatically be created inside the /media directory. You can access the contents of the CD inside this directory.
/mnt	Used to mount other temporary file systems, such as cdrom and floppy for the CD-ROM drive and floppy diskette drive, respectively
/opt	he /opt directory contains subdirectories for optional software packages. It's commonly used by proprietary software that doesn't obey the standard file system hierarchy – for example, a proprietary program might dump its files in /opt/application when you install it.
/proc	The /proc directory similar to the /dev directory because it doesn't contain standard files. It contains special files that represent system and process information.
/root	The /root directory is the home directory of the root user. Instead of being located at /home/root, it's located at /root. This is distinct from /, which is the system root directory.
/tmp	Holds temporary files used between system boots
/usr	Used for miscellaneous purposes, and can be used by many users. Includes administrative commands, shared files, library files, and others
/var	Typically contains variable-length files such as log and print files and any other type of file that may contain a variable amount of data
/sbin	Contains binary (executable) files, usually for system administration. For example, fdisk and ifconfig utlities



Homework:

1. Understand linux directory system.
2. Use BASH Commands on terminal
3. Use Help command to explore more commands
4. Use Command with --help parameter to explore more options
5. Use man command to explore manual about any command.

Lab No 03.a

BASH Scripting

Objective: Objective of this Lab is to understand how to write a LInux script
Scope: Linux Bash
Step 1: Choose Text Editor

Shell scripts are written using text editors. On Linux systems, there are a number to choose from: Vim, Emacs, Nano, Pico, Kedit, Gedit, Geany, Notepad++, Kate, Jed or LeafPad.

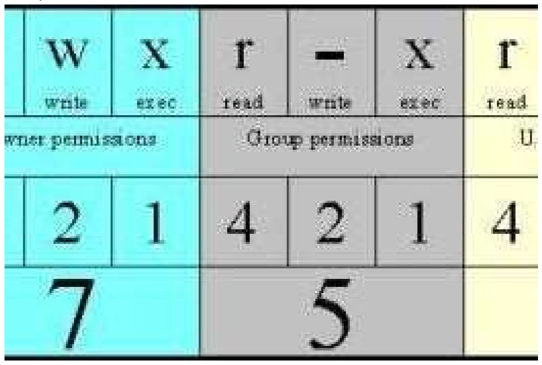
Once you have chosen a text editor, start the text editor, open a new file to begin to typing a shell script.

Step 2: Type in Commands

Start to type in basic commands that you would like the script to run.
Be sure to type each command on a separate line.
For example, to print out words to the screen use the "echo" command:
echo "This statement will print out to the screen."
To list files in a directory, type:
echo "Now we are going to list files."
ls
To print the current directory you are in, type:
echo "Next we are going to print the directory we are in:"
pwd
Save the file under the name: FirstShellScript.sh
Step 3: Make File Executable

Now that the file has been saved, it needs to be made executable. This is done using the chmod command. On your Linux command line type:

chmod 555 FirstShellScript.sh



W write	X exec	r read	- write	X exec	r read
wner permissions		Group permissions			U
2	1	4	2	1	4
7		5			

This will allow you to execute the shell script to run the commands contained within it.

Step 4: Run the Shell Script

1. To run the shell script, navigate to the directory where the file you just saved exists.

2. Now type the following [be sure to type the "dot slash" before it!]:

./FirstShellScript.sh

3. Then hit the Enter key to execute it

4. The commands that you saved in the shell script will now run.

Lab No 03.b

Variables

Objective: To Understande the basics of BASH

1. SHELL Keywords

echo, read, if fi, else, case, esac, for , while , do , done, until , set, unset, readonly, shift, export, break, continue, exit, return, trap , wait, eval ,exec, ulimit , umask.

2. General things SHELL

<u>The shbang line</u>
The "shbang" line is the very first line of the script and lets the kernel know what shell will be interpreting the lines in the script. The shbang line consists of a #! followed by the full pathname to the shell, and can be followed by options to control the behavior of the shell.
EXAMPLE

```
#!/bin/sh
```

<u>Comments</u>
Comments are descriptive material preceded by a # sign. They are in effect until the end of a line and can be started anywhere on the line.
EXAMPLE

```
# this text is not
# interpreted by the shell
```

<u>Wildcards</u>
There are some characters that are evaluated by the shell in a special way. They are called shell metacharacters or "wildcards." These characters are neither numbers nor letters. For example, the *, ?, and [ ] are used for filename expansion. The <, >, 2>, >>, and | symbols are used for standard I/O redirection and pipes. To prevent these characters from being interpreted by the shell they must be quoted.

3. SHELL Variables

Shell variables change during the execution of the program.
For example:

```
myname= “Salman”
myname = "Salman Ahmed"
age=25
```

A $ sign operator is used to recall the variable values.
For example:
echo $myname will display Salman Ahmed on the screen.

<u>Local variables</u>
Local variables are in scope for the current shell. When a script ends, they
are no longer available; i.e., they go out of scope. Local variables are set and
assigned values.
EXAMPLE

```
variable_name=value
name="John Doe"
x=5
```

<u>Global variables</u>
Global variables are called environment variables. They are set for the currently
running shell and any process spawned from that shell. They go out of scope when
the script ends.
EXAMPLE

```
VARIABLE_NAME=value
export VARIABLE_NAME
PATH=/bin:/usr/bin:.
export PATH
```

Extracting values from variables To extract the value from variables, a dollar sign is used.
EXAMPLE

echo $variable_name
echo $name
echo $PATH

<u>Rules</u>

1. A variable name is any combination of alphabets, digits and an underscore („-„);
2. No commas or blanks are allowed within a variable name.
3. The first character of a variable name must either be an alphabet or an underscore.
4. Variables names should be of any reasonable length.
5. Variables name are case sensitive . That is , Name, NAME, name, Name, are all different variables.

4. Expression Command

To perform all arithematic operations .
Syntax :
Var = $value1" + $ value2"
To perform all String operations
EXAMPLE

Equality:

=	string
!=	string
-eq	number
-ne	number

Logical:

-a	and
-o	or
!	not

Logical:

AND	&&
OR	||

05.READ Statement :

To get the input from the user.
Syntax :
read x y
(no need of commas between variables)

06. ECHO Statement :

Similar to the output statement. To print output to the screen, the echo command is used.
Wildcards must be escaped with either a backslash or matching quotes.
Syntax :
Echo “String” (or) echo $ b(for variable).
EXAMPLE
echo "What is your name?"
Reading user input
The read command takes a line of input from the user and assigns it to
a variable(s) on the right-hand side. The read command can accept muliple variable names.
Each variable will be assigned a word.
EXAMPLE
echo "What is your name?"
read name
read name1 name2 ...

Lab No 03.c

Conditional Statements & Loops

Objective: To understand Conditional Statements of BASH

CONDITIONAL STATEMENTS

The if construct is followed by a command. If an expression is to be tested, it is enclosed in square brackets. The then keyword is placed after the closing parenthesis. An if must end with a fi.
Syntax :
1.if
This is used to check a condition and if it satisfies the condition if then does the next action , if not it goes to the else part.
2.if...else
Syntax :

```
If cp $ source $ target
then
  echo File copied successfully
else
  echo Failed to copy the file.
```

3.nested if
here sequence of condition are checked and the corresponding
performed accordingly.
Syntax :

```
if condition
then
  command
if condition
then
  command
else
  command
fi
fi
```

4.case ..... esac
This construct helps in execution of the shell script based on
Choice.
EXAMPLE
The if construct is:

```
if command
then
  block of statements
fi
--------------------------------------
if [ expression ]
then
  block of statements
fi
--------------------------------------
```

The if/else/else if construct is:

```
if command
then
  block of statements
elif command
then
  block of statements
elif command
Then
  block of statements
else
```

```
 block of statements
fi
------------------------------
if [ expression ]
then
 block of statements
elif [ expression ]
then
block of statements
elif [ expression ]
then
 block of statements
else
block of statements
fi
--------------------------------------
red|orange)
echo $color is red or orange
;;
*) echo "Not a color" # default
esac
```

The if/else construct is:

```
if [ expression ]
then
 block of statements
else
 block of statements
fi
--------------------------------------
```

LOOPS

There are three types of loops: while, until and for. The while loop is followed by a command or an expression enclosed in square brackets, a do keyword, a block of statements, and terminated with the done keyword. As long as the expression is true, the body of statements between do and done will be executed.
The until loop is just like the while loop, except the body of the loop will be executed as long as the expression is false.
The for loop used to iterate through a list of words, processing a word and then shifting it off, to process the next word. When all words have been shifted from the list, it ends. The for loop is followed
by a variable name, the in keyword, and a list of words then a block of statements, and terminates with
the done keyword.
The loop control commands are break and continue.
EXAMPLE

```
while command
do
 block of statements
done
------------
while [ expression ]
do
 block of statements
done
until command
do
```

```
  block of statements
done
------------
for variable in word1 word2 word3 ...
do
  block of statements
done
until [ expression ]
do
  block of statements
done
------------
until control command
do
commands
done
```

08. Break Statement :
This command is used to jump out of the loop instantly, without waiting to get the control command.

Homework:

1. Write a Bash script to compare to strings.
2. Write a Bash script to find maximum of three numbers.
3. Write a Bash script to find fibonacci Series

Lab No 03.d

Functions

Objective: To Understand the concept os File related commands and Functions in BASH.
Scope: Linux BASH

FILE TESTING

The Bourne shell uses the test command to evaluate conditional expressions and has a built-in set of options for testing attributes of files, such as whether it is a directory, a plain file (not a directory),
a readable file, and so forth.

EXAMPLE

```
-d File is a directory
-f File exists and is not a directory
–r Current user can read the file
–s File is of nonzero size
–w Current user can write to the file
–x Current user can execute the file
#!/bin/sh
1 if [ –f file ]
then
echo file exists
fi
2 if [ –d file ]
then
echo file is a directory
fi
3 if [ -s file ]
then
echo file is not of zero length
fi
4 if [ -r file -a -w file ]
then
echo file is readable and writable
Fi
```

FUNCTIONS

Compared to most programming languages, Bash functions are somewhat limited. In this tutorial, we will cover the basics of Bash functions and show you how to use them in your shell scripts.

Bash Function Declaration

The syntax for declaring a bash function is very simple. They may be declared in two different formats:

The first format starts with the function name, followed by parentheses. This is the preferred and more used format.

```
function_name () {
  commands
}
```

Single line version:

```
function_name () { commands; }
```

The second format starts with the function reserved word followed by the function name.

```
function function_name {
  commands
```

}

Single line version:

function function_name { commands; }

The command list between curly braces {} is the body of the function. The curly braces that surround the function body must be separated from the body by spaces or newlines.
Defining a function doesn't execute it. To invoke a bash function, simply use the function name. Commands between the curly braces are executed whenever the function is called in the shell script.
The function definition must be placed before any calls to the function.
When using single line "compacted" functions, a semicolon ; must follow the last command in the function.
You should always try to keep your function names descriptive.
To understand this better, take a look at the following example:

```
~/hello_world.sh
#!/bin/bash


hello_world () {
  echo 'hello, world'
}


hello_world
```

Let's analyze the code line by line:

In line 3 we are defining the function by giving it a name, and opening the curly brace { that marks the start of the function's body.
Line 4 is the function body. The function body can contain multiple commands and variable declarations.
Line 5, the closing curly bracket }, defines the end of the hello_world function.
In line 7 we are executing the function. You can execute the function as many times as you need.
If you run the script, it will print hello, world.

Variables Scope

Global variables are variables that can be accessed from anywhere in the script regardless of the scope. In Bash, all variables by default are defined as global, even if declared inside the function.

Local variables can be declared within the function body with the local keyword and can be used only inside that function. You can have local variables with the same name in different functions.

To better illustrate how variables scope works in Bash, let's consider an example:

```
~/variables_scope.sh
#!/bin/bash


var1='A'
var2='B'
```

```
my_function () {
  local var1='C'
  var2='D'
  echo "Inside function: var1: $var1, var2: $var2"
}

echo "Before executing function: var1: $var1, var2: $var2"


my_function

echo "After executing function: var1: $var1, var2: $var2"
```

The script starts by defining two global variables var1 and var2. Then a function that sets a local variable var1 and modifies the global variable var2.

If you run the script, you should see the following output:

```
Before executing function: var1: A, var2: B
Inside function: var1: C, var2: D
After executing function: var1: A, var2: D
```

From the output above, we can conclude that:

If you set a local variable inside the function body with the same name as an existing global variable, it will have precedence over the global variable.
Global variables can be changed from within the function.
Return Values
Unlike functions in “real” programming languages, Bash functions don’t allow you to return a value when called. When a bash function completes, its return value is the status of the last statement executed in the function, 0 for success and non-zero decimal number in the 1 - 255 range for failure.

The return status can be specified by using the return keyword, and it is assigned to the variable $?. The return statement terminates the function. You can think of it as the function’s exit status.

```
~/return_values.sh
#!/bin/bash


my_function () {
  echo "some result"
  return 55
}


my_function
echo $?
Copy
```

```
some result
55
```

To actually return an arbitrary value from a function, we need to use other methods. The simplest option is to assign the result of the function to a global variable:

```
~/return_values.sh
#!/bin/bash

my_function () {
  func_result="some result"
}

my_function
echo $func_result
Copy
some result
```

Another, better option to return a value from a function is to send the value to stdout using echo or printf like shown below:

```
~/return_values.sh
#!/bin/bash

my_function () {
  local func_result="some result"
  echo "$func_result"
}

func_result="$(my_function)"
echo $func_result
```

Instead of simply executing the function which will print the message to stdout, we are assigning the function output to the func_result variable using the $() command substitution. The variable can later be used as needed.

Passing Arguments to Bash Functions

To pass any number of arguments to the bash function simply put them right after the function's name, separated by a space. It is a good practice to double-quote the arguments to avoid misparsing of an argument with spaces in it.
The passed parameters are $1, $2, $3 … $n, corresponding to the position of the parameter after the function's name.
The $0 variable is reserved for the function's name.
The $# variable holds the number of positional parameters/arguments passed to the function.
The $* and $@ variables holds all positional parameters/arguments passed to the function.

When double quoted, "$*" expands to a single string separated by space (the first character of IFS) - "$1 $2 $n".
When double quoted, "$@" expands to separate strings - "$1" "$2" "$n".
When not double quoted, $* and $@ are the same.
Here is an example:

```
~/passing_arguments.sh
#!/bin/bash

greeting () {
  echo "Hello $1"
}

greeting "Joe"

Hello Joe
```

Homework:
1. Write a Bash script to compare to strings using function.
2. Write a Bash script to find maximum of three numbers using function.
3. Write a Bash script to find fibonacci Series using function
4. Create calculator application using function.

Lab No 04.a

Fork System Call

Objective: To write c program to implement the Create Process system calls.

Scope: Create process in Linux Environment

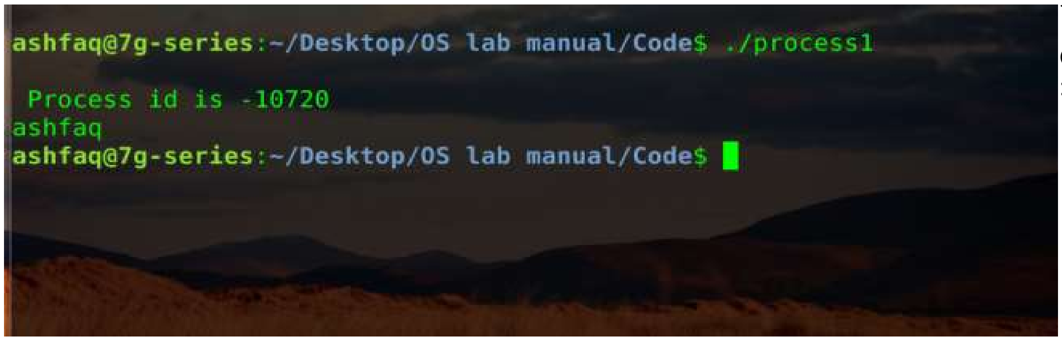
ALGORITHM:

1. Start the program.
2. Declare the pid and get the pid by using the getpid() method.
3. Create a child process by calling the fork() system call
4. Check if(pid==0) then print the child process id and then print the parent process value. Otherwise print
5. Stop the program

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
void main(int argc,char *arg[])
{
int pid; pid=fork();
if(pid<0)
{
printf("fork failed");
exit(1);
}
else if(pid==0)
{
execlp("whoami","ls",NULL);
exit(0);
}
else
{
printf("\n Process id is -%d\n",getpid());
wait(NULL);
exit(0);
}
}
```

Output:



```
ashfaq@7g-series:~/Desktop/OS lab manual/Code$ ./process1

 Process id is -10720
ashfaq
ashfaq@7g-series:~/Desktop/OS lab manual/Code$
```

Lab No 04.b

Wait Process System Calls

Objective: to Understand the wait system call

WAIT COMMAND

AIM:
To perform wait command using c program.
ALGORITHM:
STEP 1:Start the execution
STEP 2:Create process using fork and assign it to a variable
STEP 3:Check for the condition pid is equal to 0
STEP 4:If it is true print the value of i and teriminate the child process
STEP 5:If it is not a parent process has to wait until the child teriminate
STEP 6:Stop the execution
PROGRAM:

```
int i=10;
void main()
{
int pid=fork();
if(pid==0)
{
printf("initial value of i %d \n ",i);
i+=10;
printf("value of i %d \n ",i);
printf("child terminated \n");
}
else
{
wait(0);
printf("value of i in parent process %d",i);
}
}
```

OUTPUT:

```
initial value of i 10
value of i 20
child teriminated
value of i in parent process 10
```

Homework:
Create a child process and ask the child process to calculate the table of a number which was passed by parent process.

Lab No 05.a

Signal Handling

Objective: Understand how signal works in Linux

SIGNAL HANDLING
AIM:
To write a program for signal handling in UNIX.
ALGORITHM:
STEP 1:start the program
STEP 2:Read the value of pid.
STEP 3:Kill the command surely using kill-9 pid.
STEP 4:Stop the program.
PROGRAM:

```
echo program for performing KILL operations
ps
echo enter the pid
read pid
kill-9 $pid
echo finished
```

OUTPUT:

```
$sh kill.sh
program for performing KILL operations
PID CLS PRI TTY
TIME COMD
858 TS 70 pts001 0:00 ksh
858 TS 70 pts001 0:00 sh
858 TS 59 pts001 0:00 ps
enter the pid
872
killed
```

Lab No 05.b

Producer-Consumer Problem

Objective: Understand the concept of Producer and consumer

Producer-consumer problem, is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process. One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

PROGRAM

```
#include<stdio.h>
void main()
{
int buffer[10], bufsize, in, out, produce, consume, choice=0;
in = 0;
out = 0;
bufsize = 10;
while(choice !=3)
{
printf("\n1. Produce \t 2. Consume \t3. Exit");
printf("\nEnter your choice: ");
scanf("%d", &choice);
switch(choice) {
case 1: if((in+1)%bufsize==out)
printf("\nBuffer is Full");
else
{
printf("\nEnter the value: ");
scanf("%d", &produce);
buffer[in] = produce;
in = (in+1)%bufsize;
}
Break;
case 2: if(in == out)
printf("\nBuffer is Empty");
else
{
consume = buffer[out];
printf("\nThe consumed value is %d", consume);
out = (out+1)%bufsize;
}
break;
}
} }
```

OUTPUT

```
1. Produce
2. Consume
Enter your choice: 2
Buffer is Empty
1. Produce
2. Consume
Enter your choice: 1
Enter the value: 100
1. Produce
2. Consume
Enter your choice: 2
The consumed value is 100
1. Produce
2. Consume
Enter your choice: 3
3. Exit
3. Exit
3. Exit
3. Exit
```

Lab No 05.c

Sleep System Call

Objective: understand the Concept of Sleep system call

SLEEP

SLEEP COMMAND USING GETPID
AIM:
To create child with sleep command using getpid.
ALGORITHM:
STEP 1: Start the execution and create a process using fork() command.
STEP 2: Make the parent process to sleep for 10 seconds.
STEP 3:In the child process print it pid and it corresponding pid.
STEP 4: Make the child process to sleep for 5 seconds.
STEP 5: Again print it pid and it parent pid.
STEP 6: After making the sleep for the parent process for 10 seconds print it pid.
STEP 7: Stop the execution.
PROGRAM:

```
void main()
{
int pid;
pid=fork();
if (pid==0)
{
printf("\n Child Process\n");
printf("\n Child Process id is %d ",getpid());
printf("\n Its parent process id is %d",getppid());
sleep(5);
printf("Child process after sleep=5\n");
printf("\n Child Process id is %d ",getpid());
printf("\n Its parent process id is %d",getppid());
}
else
{
printf("\nParent process");
sleep(10);
printf("\n Child Process id is %d ",getpid());
printf("\n Its parent process id is %d",getppid());
printf("\nParent terminates\n");
}
}
```

OUTPUT:

```
$ cc sleepid.c
$ a.out
parent process
child process
child process id is 12691
```

```
its parent process id is 12690
child process after sleep=5
child process id is 12691
its parent process id is 12690
child process after sleep=10
child id is 12690
parent id is 11383
parent terminates
```

Lab No 06.a

File Read System Call

Objective: Understand the I/O system calls

READING FROM A FILE

AIM:
To create the file,read data from the file,update the file.
ALGORITHM:
1.Get the data from the user.
2.Open a file.
3.Read from the file.
4.Close the file.
PROGRAM:

```
#include<stdio.h>
int main()
{
char str[100];
FILE *fp;
fp=fopen("file1.dat","r");
while(!feof(fp))
{
fscanf(fp,"%s",str);
printf(" %s ",str);
}
fclose(fp);
}
```

OUTPUT:

```
This is a program to read the content of the file.
```

Lab No 06.b

File Write System Calls

Objective: Understand the concept of writing File in C

WRITING INTO A FILE

AIM:
To write a C program to write the data into a file.
ALGORITHM:
Step1.Get the data from the user.
Step2.Open a file.
Step3.Write the data from the file.
Step4.Get the data and update the file.
PROGRAM:

```
#include<stdio.h>
int main()
{
char str[100];
FILE *fp;
printf("Enter the string");
scanf(“%s”,str)
fp=fopen("file1.dat","w+");
printf("Written sucessfully in file\n");
while(!feof(fp))
{
fscanf(fp,"%s",str);
}
fprintf(fp,"%s",str);
}
```

OUTPUT:

```
$ gcc write.c
$ ./a.out
Enter the string: os lab
Written sucessfully in file
```

Lab No 06.c

File Create System Calls

Objective: Under the File creation in Linux

FILE CREATION

Date:
AIM:
To write a C program to create a file.
ALGORITHM:
Step1:Start the program.
Step2:Create the file using create function and assign a variable to it.
Step3:If the value of the variable is less then print file cannot be created ,otherwise print file is created.
Step4:Stop the program.
PROGRAM:

```
void main()
{
int id;
if(id=creat("z.txt",0)==-1)
{
printf("Cannot create the file");
exit(1);
}
else
{
printf("File is created");
exit(1);
}
}
```

OUTPUT:

```
File is created
```

Lab No 07

Multi-Threading in C

Objective: Understand the concept off Multi threading in C
Scope: Multithreading in C using pthread library

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

/*thread function definition*/
void* threadFunction(void* args)
{
    while(1)
    {
        printf("I am threadFunction.\n");
    }
}
int main()
{
    /*creating thread id*/
    pthread_t id;
    int ret;

    /*creating thread*/
    ret=pthread_create(&id,NULL,&threadFunction,NULL);
    if(ret==0){
        printf("Thread created successfully.\n");
    }
    else{
        printf("Thread not created.\n");
        return 0; /*return from main*/
    }

    while(1)
    {
        printf("I am main function.\n");
    }

    return 0;
}
```

Lab No 8.a

CPU Scheduling

First Come First Serve

Objective: Understand how FCFS Works in CPU scheduling

ALGORITHM:

Step 1: Create the number of process.
Step 2: Get the ID and Service time for each process.
Step 3: Initially, Waiting time of first process is zero and Total time for the first process is the starting time of that process.
Step 4: Calculate the Total time and Processing time for the remaining processes.
Step 5: Waiting time of one process is the Total time of the previous process.
Step 6: Total time of process is calculated by adding Waiting time and Service time.
Step 7: Total waiting time is calculated by adding the waiting time for lack process.
Step 8: Total turn around time is calculated by adding all total time of each process.
Step 9: Calculate Average waiting time by dividing the total waiting time by total number of process.
Step 10: Calculate Average turn around time by dividing the total time by the number of process.
Step 11: Display the result.

SOURCE CODE :

```
#include<stdio.h>
struct process
{
int id,wait,ser,tottime;
}p[20];
main()
{
int i,n,j,totalwait=0,totalser=0,avturn,avwait;
printf("Enter number of process");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
printf("enter process_id");
scanf("%d",&p[i].id);
printf("enter process Burst time");
scanf("%d",&p[i].ser);
}
p[1].wait=0;
p[1].tottime=p[1].ser;
for(i=2;i<=n;i++)
{
for(j=1;j<i;j++)
{
p[i].wait=p[i].wait+p[j].ser;
}
totalwait=totalwait+p[i].wait;
p[i].tottime=p[i].wait+p[i].ser;
totalser=totalser+p[i].tottime;
}
avturn=totalser/n;
avwait=totalwait/n;
printf("Id\tservice\twait\ttotal");
for(i=1;i<=n;i++)
{
printf("\n%d\t%d\t%d\t%d\n",p[i].id,p[i].ser,p[i].wait,p[i].tottime);
}
printf("average waiting time %d\n",avwait);
printf("average turnaround time %d\n",avturn);
}
```

Lab No 08.b

CPU Scheduling

Shortest Job First

Objective: Understand the concept of Shortest job first CPU scheduling Algorithm.

AIM:
To write a program to implement cpu scheduling algorithm for shortest job first scheduling.
ALGORITHM:
1. Start the program. Get the number of processes and their burst time.
2. Initialize the waiting time for process 1 as 0.
3. The processes are stored according to their burst time.
4. The waiting time for the processes are calculated a follows:
for(i=2;i<=n;i++).wt.p[i]=p[i=1]+bt.p[i-1].
5. The waiting time of all the processes summed and then the average time is calculate
6. The waiting time of each processes and average time are displayed.
7. Stop the program.

PROGRAM

```
#include<stdio.h>
#include<conio.h>
struct process
{
int pid;
int bt;
int wt;
int tt;
}p[10],temp;
int main()
{
int i,j,n,totwt,tottt;
float avg1,avg2;
clrscr();
printf("\nEnter the number of process:\t");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
p[i].pid=i;
printf("\nEnter the burst time:\t");
scanf("%d",&p[i].bt);
}
for(i=1;i<n;i++){
for(j=i+1;j<=n;j++)
{
if(p[i].bt>p[j].bt)
{
temp.pid=p[i].pid;
p[i].pid=p[j].pid;
p[j].pid=temp.pid;
temp.bt=p[i].bt;p[i].bt=p[j].bt;
p[j].bt=temp.bt;
}}}
p[1].wt=0;
p[1].tt=p[1].bt+p[1].wt;
i=2;
while(i<=n){
p[i].wt=p[i-1].bt+p[i-1].wt;
p[i].tt=p[i].bt+p[i].wt;
i++;
}
i=1;
totwt=tottt=0;
printf("\nProcess id \tbt \twt \ttt");
while(i<=n){
printf("\n\t%d \t%d \t%d t%d\n",p[i].pid,p[i].bt,p[i].wt,p[i].tt);
totwt=p[i].wt+totwt;
tottt=p[i].tt+tottt;
i++;
}
avg1=totwt/n;
avg2=tottt/n;
printf("\nAVG1=%f\t AVG2=%f",avg1,avg2);
getch();
return 0; }
```

OUTPUT:

```
enter the number of process 3
enter the burst time: 2
enter the burst time: 4
enter the burst time: 6
processid bt wt tt
1 2 0 2
2 4 2 6
3 6 6 12
AVG1=2.000000
AVG2=6.000000
```

Lab No 09.a

CPU Scheduling

Priority Scheduling

Objective: Understand the concept of Priority CPU scheduling Algorithm.

AIM:
To write a 'C' program to perform priority scheduling.
ALGORITHM:
1. Start the program.
2. Read burst time, waiting time, turn the around time and priority.
3. Initialize the waiting time for process 1 and 0.
4. Based up on the priority process are arranged
5. The waiting time of all the processes is summed and then the average waiting time
6. The waiting time of each process and average waiting time are displayed based on the priority.
7. Stop the program.

PROGRAM
```
#include<stdio.h>
#include<conio.h>
struct process
{
int pid;
int bt;
int wt;
int tt;
int prior;
}
p[10],temp;
int main()
{
int i,j,n,totwt,tottt,arg1,arg2;
clrscr();
printf("enter the number of process");
scanf("%d",&n);
for(i=1;i<=n;i++)
{
p[i].pid=i;
printf("enter the burst time");
scanf("%d",&p[i].bt);
printf("\n enter the priority");
scanf("%d",&p[i].prior);
}
for(i=1;i<n;i++)
{
for(j=i+1;j<=n;j++)
{
if(p[i].prior>p[j].prior)
{
temp.pid=p[i].pid;
p[i].pid=p[j].pid;
p[j].pid=temp.pid;
temp.bt=p[i].bt;
p[i].bt=p[j].bt;
p[j].bt=temp.bt;
temp.prior=p[i].prior;
p[i].prior=p[j].prior;
p[j].prior=temp.prior;
}
}
}
p[i].wt=0;
p[1].tt=p[1].bt+p[1].wt;
i=2;
while(i<=n)
{
p[i].wt=p[i-1].bt+p[i-1].wt;
p[i].tt=p[i].bt+p[i].wt;
i++;
}
i=1;
totwt=tottt=0;
printf("\n process to \t bt \t wt \t tt");
while(i<=n)
{
printf("\n%d\t %d\t %d\t %d\t",p[i].pid,p[i].bt,p[i].wt,p[i].tt);
totwt=p[i].wt+totwt;
tottt=p[i].tt+tottt;
i++;
}
arg1=totwt/n;
arg2=tottt/n;
printf("\n arg1=%d \t arg2=%d\t",arg1,arg2);
getch();
return 0;
```

```
    }
```

OUTPUT:

```
    enter the no of process:3
    enter the burst time:2
    enter the priority:3
    enter the burst time:4
    enter the priority:1
    enter the burst time:6
    enter the priority:2
    process to
    bt wt tt 1 4 0 4 4
    2 6 4 10 14
    3 2 10 12 22
    avg1=4
    avg2=8
```

Lab No 09.b

CPU Scheduling

Round Robin

Objective: Understand the concept of Round Robin CPU scheduling Algorithm.
AIM:
To write a program to implement cpu scheduling for Round Robin Scheduling.
ALGORITHM:
1. Get the number of process and their burst time.
2. Initialize the array for Round Robin circular queue as '0'.
3. The burst time of each process is divided and the quotients are stored on the round Robin array.
4. According to the array value the waiting time for each process and the average time are calculated as line the other scheduling.
5. The waiting time for each process and average times are displayed.
6. Stop the program.

PROGRAM :

```
#include<stdio.h>
#include<conio.h>
struct process
{
int pid,bt,tt,wt;
};
int main()
{
struct process x[10],p[30];
int i,j,k,tot=0,m,n;
float wttime=0.0,tottime=0.0,a1,a2;
clrscr();
printf("\nEnter the number of process:\t");
scanf("%d",&n);
for(i=1;i<=n;i++){
x[i].pid=i;
printf("\nEnter the Burst Time:\t");
scanf("%d",&x[i].bt);
tot=tot+x[i].bt;
}
printf("\nTotal Burst Time:\t%d",tot);
p[0].tt=0;
k=1;
printf("\nEnter the Time Slice:\t");
scanf("%d",&m);
for(j=1;j<=tot;j++)
{
for(i=1;i<=n;i++)
{
```

```
if(x[i].bt !=0)
{
p[k].pid=i;
if(x[i].bt-m<0)
{
p[k].wt=p[k-1].tt;
p[k].bt=x[i].bt;
p[k].tt=p[k].wt+x[i].bt;
x[i].bt=0;
k++;
}
else
{
p[k].wt=p[k-1].tt;
p[k].tt=p[k].wt+m;
x[i].bt=x[i].bt-m;
k++;
}
}
}
}
printf("\nProcess id \twt \ttt");
for(i=1;i<k;i++){
printf("\n\t%d \t%d \t%d",p[i].pid,p[i].wt,p[i].tt);
wttime=wttime+p[i].wt;
tottime=tottime+p[i].tt;
a1=wttime/n;
a2=tottime/n;
}
printf("\n\nAverage Waiting Time:\t%f",a1);
printf("\n\nAverage TurnAround Time:\t%f",a2);
getch();
return 0;
}
```

OUTPUT:

```
enter the no of process3
enter the burst time3
enter the burst time5
enter the burst time7
```

```
total burst time : 15
enter the time slice: 2
process id
wt tt
1 0 2
2 2 4
3 4 6
1 6 7
wt tt
2 7 9
3 9 11
2 11 12
3 12 14
3 14 15
process id
avg waiting time: 21.666666
avg turnaround time: 26.666666
```

Lab No 10.a

Process Communication

Pipe

Objective: Understand the process communication using pip

AIM :
To write a program for create a pope processing
ALGORITHM:
1. Start the program.
2. Declare the variables.
3. Read the choice.
4. Create a piping processing using IPC.
5. Assign the variable lengths
6. “strcpy” the message lengths.
7. To join the operation using IPC .
8. Stop the program

PROGRAM :

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define MSG_LEN 64
int main(){
int result;
int fd[2];
char message[MSG_LEN];
char recvd_msg[MSG_LEN];
result = pipe (fd);
//Creating a pipe//fd[0] is for reading and fd[1] is for writing
if (result < 0)
{
perror("pipe ");
exit(1);
}
strncpy(message,"Linux World!! ",MSG_LEN);
result=write(fd[1],message,strlen(message));
if (result < 0)
{
perror("write");
exit(2);
}
strncpy(message,"Understanding ",MSG_LEN);
result=write(fd[1],message,strlen(message));
if (result < 0)
{
```

```
perror("write");
exit(2);
}
strncpy(message,"Concepts of ",MSG_LEN);
result=write(fd[1],message,strlen(message));
if (result < 0)
{
perror("write");
exit(2);
}
strncpy(message,"Piping ", MSG_LEN);
result=write(fd[1],message,strlen(message));
if (result < 0)
{
perror("write");
exit(2);
}
result=read (fd[0],recvd_msg,MSG_LEN);
if (result < 0)
{
perror("read");
exit(3);
}
printf("%s\n",recvd_msg);
return 0;}
```

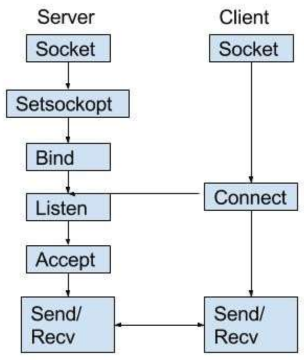
Lab No 10.b

Process Communication

Socket

Objective: Understand the concept of Socket in Linux

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while other socket reaches out to the other to form a connection. Server forms the listener socket while client reaches out to the server.



Socket creation:

int sockfd = socket(domain, type, protocol)

sockfd: socket descriptor, an integer (like a file-handle)
domain: integer, communication domain e.g., AF_INET (IPv4 protocol) , AF_INET6 (IPv6 protocol)
type: communication type

SOCK_STREAM: TCP(reliable, connection oriented)

SOCK_DGRAM: UDP(unreliable, connectionless)

protocol: Protocol value for Internet Protocol(IP), which is 0. This is the same number which appears on protocol field in the IP header of a packet.(man protocols for more details)

Setsockopt:

```
int setsockopt(int sockfd, int level, int optname,
               const void *optval, socklen_t optlen);
```

This helps in manipulating options for the socket referred by the file descriptor sockfd. This is completely optional, but it helps in reuse of address and port. Prevents error such as: “address already in use”.

Bind:

```
int bind(int sockfd, const struct sockaddr *addr,
                     socklen_t addrlen);
```

After creation of the socket, bind function binds the socket to the address and port number specified in addr(custom data structure). In the example code, we bind the server to the localhost, hence we use INADDR_ANY to specify the IP address.

Listen:

```
int listen(int sockfd, int backlog);
```

It puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The backlog, defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED.

Accept:

*int new_socket= accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);*

It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. At this point, connection is established between client and server, and they are ready to transfer data.

Stages for Client

Socket connection: Exactly same as that of server's socket creation

Connect:

```
int connect(int sockfd, const struct sockaddr *addr,
                                    socklen_t addrlen);
```

The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. Server's address and port is specified in addr.

Server:

```
//Server.c
// Server side C/C++ program to demonstrate Socket programming
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#define PORT 8080
int main(int argc, char const *argv[])
{
    int server_fd, new_socket, valread;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);
    char buffer[1024] = {0};
    char *hello = "Hello from server";

    // Creating socket file descriptor
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // Forcefully attaching socket to the port 8080
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,
                                                  &opt, sizeof(opt)))
    {
        perror("setsockopt");
        exit(EXIT_FAILURE);
    }
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons( PORT );

    // Forcefully attaching socket to the port 8080
    if (bind(server_fd, (struct sockaddr *)&address,
                                sizeof(address))<0)
    {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }
    if (listen(server_fd, 3) < 0)
    {
        perror("listen");
        exit(EXIT_FAILURE);
    }
    if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
                    (socklen_t*)&addrlen))<0)
    {
        perror("accept");
        exit(EXIT_FAILURE);
    }
    valread = read( new_socket , buffer, 1024);
    printf("%s\n",buffer );
    send(new_socket , hello , strlen(hello) , 0 );
    printf("Hello message sent\n");
    return 0;
}
```

Client:

```
    //Client.c
    // Client side C/C++ program to demonstrate Socket programming
    #include <stdio.h>
    #include <sys/socket.h>
    #include <arpa/inet.h>
    #include <unistd.h>
```

```
#include <string.h>
#define PORT 8080

int main(int argc, char const *argv[])
{
    int sock = 0, valread;
    struct sockaddr_in serv_addr;
    char *hello = "Hello from client";
    char buffer[1024] = {0};
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n Socket creation error \n");
        return -1;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Convert IPv4 and IPv6 addresses from text to binary form
    if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0)
    {
        printf("\nInvalid address/ Address not supported \n");
        return -1;
    }

    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
    {
        printf("\nConnection Failed \n");
        return -1;
    }
    send(sock , hello , strlen(hello) , 0 );
    printf("Hello message sent\n");
    valread = read( sock , buffer, 1024);
    printf("%s\n",buffer );
    return 0;
}
```

Output:

```
Client:Hello message sent
```

```
Hello from server
Server:Hello from client
Hello message sent
```

Lab No 11

Dinning Philosopher Problem

Objective: Understanding Synchronization resource sharing and Dinning philosopher problem

Dinning Philosopher problem, five philosophers are seated around a circular table. Each philosopher has a place of spaghetti, and he needs two forks to eat. Between each plate there is a fork. The life of a philosopher consists of alternate period of eating & thinking. When a philosopher gets hungry, he tries to acquire his left fork, if he gets it, it tries to acquire right fork. In this solution, we check after picking the left fork whether the right fork is available or not. If not, then philosopher puts down the left fork & continues to think. Even this can fail if all the philosophers pick the left fork simultaneously & no right forks available & putting the left fork down again. This repeats & leads to starvation.
Now, we can modify the problem by making the philosopher wait for a random amount of time instead of same time after failing to acquire right hand fork. This will reduce the problem of starvation. Solution to dinning philosophers' problem.

ALGORITHM:

STEP 1: Start the program.
STEP 2: Declare pid as integer.
STEP 3: Create the process using Fork command.
STEP 4: Check pid is less than 0 then print error else if pid is equal to 0 then execute command else parent process wait for child process.
STEP 5: Stop the program.

PROGRAM

```
#include<stdio.h>

#define n 4

int compltedPhilo = 0,i;

struct fork{
int taken;
}ForkAvil[n];

struct philosp{
int left;
int right;
}Philostatus[n];

void goForDinner(int philID){ //same like threads concept here cases implemented
if(Philostatus[philID].left==10 && Philostatus[philID].right==10)
        printf("Philosopher %d completed his dinner\n",philID+1);
//if already completed dinner
else if(Philostatus[philID].left==1 && Philostatus[philID].right==1){
            //if just taken two forks
            printf("Philosopher %d completed his dinner\n",philID+1);

            Philostatus[philID].left = Philostatus[philID].right = 10; //remembering that he completed
```

```
dinner by assigning value 10
            int otherFork = philID-1;

            if(otherFork== -1)
                otherFork=(n-1);

            ForkAvil[philID].taken = ForkAvil[otherFork].taken = 0; //releasing forks
            printf("Philosopher %d released fork %d and fork %d\n",philID+1,philID+1,otherFork+1);
            compltedPhilo++;
        }
        else if(Philostatus[philID].left==1 && Philostatus[philID].right==0){ //left already taken, trying
for right fork
                if(philID==(n-1)){
                    if(ForkAvil[philID].taken==0){
                        ForkAvil[philID].taken = Philostatus[philID].right = 1;
                        printf("Fork %d taken by philosopher %d\n",philID+1,philID+1);
                    }else{
                        printf("Philosopher %d is waiting for fork %d\n",philID+1,philID+1);
                    }
                }else{ //except last philosopher case
                    int dupphilID = philID;
                    philID-=1;

                    if(philID== -1)
                        philID=(n-1);

                    if(ForkAvil[philID].taken == 0){
                        ForkAvil[philID].taken = Philostatus[dupphilID].right = 1;
                        printf("Fork %d taken by Philosopher %d\n",philID+1,dupphilID+1);
                    }else{
                        printf("Philosopher %d is waiting for Fork %d\n",dupphilID+1,philID+1);
                    }
                }
            }
            else if(Philostatus[philID].left==0){ //nothing taken yet
                    if(philID==(n-1)){
                        if(ForkAvil[philID-1].taken==0){
                            ForkAvil[philID-1].taken = Philostatus[philID].left = 1;
                            printf("Fork %d taken by philosopher %d\n",philID,philID+1);
                        }else{
                            printf("Philosopher %d is waiting for fork %d\n",philID+1,philID);
                        }
                    }else{ //except last philosopher case
                        if(ForkAvil[philID].taken == 0){
                            ForkAvil[philID].taken = Philostatus[philID].left = 1;
                            printf("Fork %d taken by Philosopher %d\n",philID+1,philID+1);
                        }else{
                            printf("Philosopher %d is waiting for Fork %d\n",philID+1,philID+1);
                        }
                    }
        }else{}
}

int main(){
for(i=0;i<n;i++)
        ForkAvil[i].taken=Philostatus[i].left=Philostatus[i].right=0;

while(compltedPhilo<n){
for(i=0;i<n;i++)
            goForDinner(i);
printf("\nTill now num of philosophers completed dinner are %d\n\n",compltedPhilo);
}

return 0;
}
```

OUTPUT

```
Fork 1 taken by Philosopher 1
Fork 2 taken by Philosopher 2
Fork 3 taken by Philosopher 3
Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 0

Fork 4 taken by Philosopher 1
Philosopher 2 is waiting for Fork 1
Philosopher 3 is waiting for Fork 2
Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 0

Philosopher 1 completed his dinner
Philosopher 1 released fork 1 and fork 4
Fork 1 taken by Philosopher 2
Philosopher 3 is waiting for Fork 2
Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 1

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 2 released fork 2 and fork 1
Fork 2 taken by Philosopher 3
Philosopher 4 is waiting for fork 3

Till now num of philosophers completed dinner are 2

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Philosopher 3 released fork 3 and fork 2
Fork 3 taken by philosopher 4

Till now num of philosophers completed dinner are 3

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Fork 4 taken by philosopher 4

Till now num of philosophers completed dinner are 3

Philosopher 1 completed his dinner
Philosopher 2 completed his dinner
Philosopher 3 completed his dinner
Philosopher 4 completed his dinner
Philosopher 4 released fork 4 and fork 3

Till now num of philosophers completed dinner are 4
```

Lab No 12

Deadlocks
Banker’s Algorithm

Objective: Understand the concept of Banker's Algorithm
Scope: Deadlock
In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state.
Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock. Deadlock avoidance is one of the techniques for handling deadlocks. This approach requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.
Banker's algorithm is a deadlock avoidance algorithm that is applicable to a system with multiple instances of
each resource type.

```
#include<stdio.h>
struct file
{
int all[10];
int max[10];
int need[10];
int flag;
};
void main()
{
struct file f[10];
int fl;
int i, j, k, p, b, n, r, g, cnt=0, id, newr;
int avail[10],seq[10];
clrscr();
printf("Enter number of processes -- ");
scanf("%d",&n);
printf("Enter number of resources -- ");
scanf("%d",&r);
for(i=0;i<n;i++)
{
printf("Enter details for P%d",i);
printf("\nEnter allocation\t -- \t");
for(j=0;j<r;j++)
scanf("%d",&f[i].all[j]);
printf("Enter Max\t\t -- \t");
for(j=0;j<r;j++)
```

```
scanf("%d",&f[i].max[j]);
f[i].flag=0;
}
printf("\nEnter Available Resources\t -- \t");
for(i=0;i<r;i++)
scanf("%d",&avail[i]);
printf("\nEnter New Request Details -- ");
printf("\nEnter pid \t -- \t");
scanf("%d",&id);
printf("Enter Request for Resources \t -- \t");
for(i=0;i<r;i++)
{
scanf("%d",&newr);
f[id].all[i] += newr;
avail[i]=avail[i] - newr;
}
for(i=0;i<n;i++)
{
for(j=0;j<r;j++)
{
f[i].need[j]=f[i].max[j]-f[i].all[j];
if(f[i].need[j]<0)
f[i].need[j]=0;
}
}
cnt=0;
fl=0;
while(cnt!=n)
{
g=0;
for(j=0;j<n;j++)
{
if(f[j].flag==0)
{
b=0;
for(p=0;p<r;p++)
{
if(avail[p]>=f[j].need[p])
b=b+1;
else
b=b-1;
}
if(b==r)
{
printf("\nP%d is visited",j);
seq[fl++]=j;
f[j].flag=1;
for(k=0;k<r;k++)
avail[k]=avail[k]+f[j].all[k];
cnt=cnt+1;
printf("(");
for(k=0;k<r;k++)
printf("%3d",avail[k]);
printf(")");
g=1;
}
}
}
if(g==0)
{
printf("\n REQUEST NOT GRANTED -- DEADLOCK OCCURRED");
printf("\n SYSTEM IS IN UNSAFE STATE");
goto y;
}
}
printf("\nSYSTEM IS IN SAFE STATE");
printf("\nThe Safe Sequence is -- (");
for(i=0;i<fl;i++)
printf("P%d ",seq[i]);
printf(")");
y: printf("\nProcess\t\tAllocation\t\tMax\t\t\tNeed\n");
for(i=0;i<n;i++)
{
printf("P%d\t",i);
for(j=0;j<r;j++)
printf("%6d",f[i].all[j]);
for(j=0;j<r;j++)
printf("%6d",f[i].max[j]);
```

```
for(j=0;j<r;j++)
printf("%6d",f[i].need[j]);
printf("\n");
}
getch();
}
```

OUTPUT:

```
Enter number of processes 5
Enter number of resources 3 0 1
Enter details for P0 7 0
Enter allocation
--
Enter Max
--
-
-- 5
3 0 1
7 0
5
Enter details for P1
Enter allocation
Enter Max --
-- 2
3 0
2 0
2
Enter details for P2
Enter allocation
Enter Max --
-- 3
9 0
0 2
2
Enter details for P3
Enter allocation
Enter Max --
-- 2
2 1
2 1
```

```
2
Enter details for P4
Enter allocation
Enter Max --
-- 0
4 0
3 2
3
3 3 2 -- 1
Enter Available Resources --
Enter New Request Details --
Enter pid
-- 1
Enter Request for Resources
0
3
2
OUTPUT
P1 is visited( 5 3 2)
P3 is visited( 7 4 3)
P4 is visited( 7 4 5)
P0 is visited( 7 5 5)
P2 is visited( 10 5 7)
SYSTEM IS IN SAFE STATE
The Safe Sequence is -- (P1 P3 P4 P0 P2 )
Process         Allocation        Max         Need
P0              0 1 0             7 5 3       7 4 3
P1              3 0 2             3 2 2       0 2 0
P2              3 0 2             9 0 2       6 0 0
p3              2 1 1             2 2 2       0 1 1
P4              0 0 2             4 3 3       4 3 1
```

Lab No 13.a

Memory Management
First Fit

Objective: Understand the Memory management MFT techniques

AIM:
To implement first fit, best fit algorithm for memory management.
ALGORITHM:
1. Start the program.
2. Get the segment size, number of process to be allocated and their corresponding size.
3. Get the options. If the option is '2' call first fit function.
4. If the option is '1' call best fit function. Otherwise exit.
5. For first fit, allocate the process to first possible segment which is free and set the personnel slap as '1'. So that none of process to be allocated to segment which is already allocated and vice versa.
6. For best fit, do the following steps,.
7. Sorts the segments according to their sizes.
8. Allocate the process to the segment which is equal to or slightly greater than the process size and set the flag as the '1' .So that none of the process to be allocated to the segment which is already allocated and vice versa. Stop the program.
8. Stop the program

PROGRAM:

```
#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 - 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)
//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();
}
```

OUTPUT:

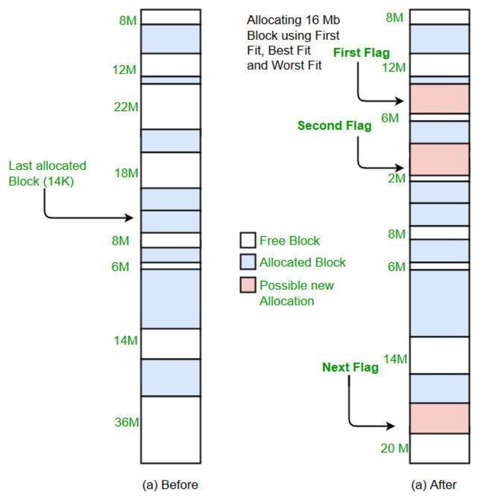
```
Enter the number of blocks: 3
Enter the number of files: 2
Enter the size of the blocks:-
Block 1: 5
Block 2:
2
Block 3: 7
Enter the size of the files:-
File 1: 1
File 2: 4
OUTPUT
File No
File Size
Block No
Block Size
1 1 3 7 6
2 4 1 5 1
Fragment
```

Lab No 13.b

Memory Management
Best Fit

Objective: Understand the concept of Best Fit

Best fit allocates the process to a partition which is the smallest sufficient partition among the free available partitions



```
// C++ implementation of Best - Fit algorithm
#include<bits/stdc++.h>
using namespace std;

// Function to allocate memory to blocks as per Best fit
// algorithm
void bestFit(int blockSize[], int m, int processSize[], int n)
{
    // Stores block id of the block allocated to a
    // process
    int allocation[n];

    // Initially no block is assigned to any process
    memset(allocation, -1, sizeof(allocation));

    // pick each process and find suitable blocks
    // according to its size ad assign to it
    for (int i=0; i<n; i++)
    {
        // Find the best fit block for current process
        int bestIdx = -1;
        for (int j=0; j<m; j++)
        {
            if (blockSize[j] >= processSize[i])
```

```
                {
                    if (bestIdx == -1)
                        bestIdx = j;
                    else if (blockSize[bestIdx] > blockSize[j])
                        bestIdx = j;
                }
            }

            // If we could find a block for current process
            if (bestIdx != -1)
            {
                // allocate block j to p[i] process
                allocation[i] = bestIdx;

                // Reduce available memory in this block.
                blockSize[bestIdx] -= processSize[i];
            }
        }

        cout << "\nProcess No.\tProcess Size\tBlock no.\n";
        for (int i = 0; i < n; i++)
        {
            cout << " " << i+1 << "\t\t" << processSize[i] << "\t\t";
            if (allocation[i] != -1)
                cout << allocation[i] + 1;
            else
                cout << "Not Allocated";
            cout << endl;
        }
    }

    // Driver code
    int main()
    {
        int blockSize[] = {100, 500, 200, 300, 600};
        int processSize[] = {212, 417, 112, 426};
        int m = sizeof(blockSize)/sizeof(blockSize[0]);
        int n = sizeof(processSize)/sizeof(processSize[0]);
        bestFit(blockSize, m, processSize, n);

        return 0 ;
    }
```

Output:

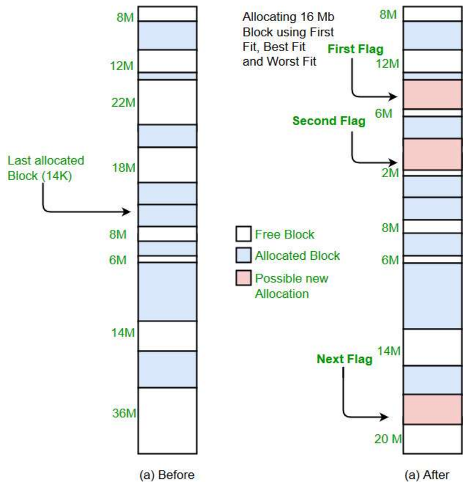
```
    Process No.   Process Size   Block no.
     1        212        4
     2        417        2
     3        112        3
     4        426        5
```

Lab No 13.c

Memory Management
Worst Fit

Objective: Understand the concept of Worst Fit

Worst Fit allocates a process to the partition which is largest sufficient among the freely available partitions available in the main memory. If a large process comes at a later stage, then memory will not have space to accommodate it.



```
// C++ implementation of worst - Fit algorithm
#include<bits/stdc++.h>
using namespace std;

// Function to allocate memory to blocks as per worst fit
// algorithm
void worstFit(int blockSize[], int m, int processSize[],
                                                    int n)
{
    // Stores block id of the block allocated to a
    // process
    int allocation[n];

    // Initially no block is assigned to any process
    memset(allocation, -1, sizeof(allocation));

    // pick each process and find suitable blocks
    // according to its size ad assign to it
    for (int i=0; i<n; i++)
    {
        // Find the best fit block for current process
        int wstIdx = -1;
        for (int j=0; j<m; j++)
        {
                if (blockSize[j] >= processSize[i])
                {
                    if (wstIdx == -1)
                        wstIdx = j;
                    else if (blockSize[wstIdx] < blockSize[j])
                        wstIdx = j;
                }
            }

            // If we could find a block for current process
            if (wstIdx != -1)
            {
                // allocate block j to p[i] process
                allocation[i] = wstIdx;

                // Reduce available memory in this block.
                blockSize[wstIdx] -= processSize[i];
            }
        }

        cout << "\nProcess No.\tProcess Size\tBlock no.\n";
        for (int i = 0; i < n; i++)
        {
            cout << " " << i+1 << "\t\t" << processSize[i] << "\t\t";
            if (allocation[i] != -1)
                cout << allocation[i] + 1;
            else
                cout << "Not Allocated";
            cout << endl;
        }
    }

    // Driver code
    int main()
    {
        int blockSize[] = {100, 500, 200, 300, 600};
        int processSize[] = {212, 417, 112, 426};
        int m = sizeof(blockSize)/sizeof(blockSize[0]);
        int n = sizeof(processSize)/sizeof(processSize[0]);
        worstFit(blockSize, m, processSize, n);

        return 0 ;
}
```

Output:

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

Lab No 14

Memory Management
Paging

Objective: Understand the concept of Paging

In computer operating systems, paging is one of the memory management schemes by which a computer stores and retrieves data from the secondary storage for use in main memory. In the paging memory-management scheme, the operating system retrieves data from secondary storage in same-size blocks called pages. Paging is a memory-management scheme that permits the physical address space a process to be noncontiguous. The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from their source.

PROGRAM

```
#include<stdio.h>
#include<conio.h>
main()
{
int ms, ps, nop, np, rempages, i, j, x, y, pa, offset;
int s[10], fno[10][20];
clrscr();
printf("\nEnter the memory size -- ");
scanf("%d",&ms);
printf("\nEnter the page size -- ");
scanf("%d",&ps);
nop = ms/ps;
printf("\nThe no. of pages available in memory are -- %d ",nop);
printf("\nEnter number of processes -- ");
scanf("%d",&np);
rempages = nop;
for(i=1;i<=np;i++)
{
printf("\nEnter no. of pages required for p[%d]-- ",i);
scanf("%d",&s[i]);
if(s[i] >rempages)
{
printf("\nMemory is Full");
break;
}
rempages = rempages - s[i];
printf("\nEnter pagetable for p[%d] --- ",i);
for(j=0;j<s[i];j++)
scanf("%d",&fno[i][j]);
}
printf("\nEnter Logical Address to find Physical Address ");
printf("\nEnter process no. and pagenumber and offset -- ");
scanf("%d %d %d",&x,&y, &offset);
if(x>np || y>=s[i] || offset>=ps)
printf("\nInvalid Process or Page Number or offset");
else
{
pa=fno[x][y]*ps+offset;
printf("\nThe Physical Address is -- %d",pa);
}
getch();
}
```

OUTPUT:

```
Enter the memory size – 1000
Enter the page size -- 100
The no. of pages available in memory are -- 10
Enter number of processes -- 3
Enter no. of pages required for p[1] -- 4
Enter pagetable for p[1] --- 8 6 9 5
Enter no. of pages required for p[2] -- 5
Enter pagetable for p[2] --- 1  4 5 7 3
Enter no. of pages required for p[3] -- 5 3 60

Memory is Full
Enter Logical Address to find Physical Address
Enter process no. and pagenumber and offset -- 2 3 60
The Physical Address is -- 760
```

Lab No 15

File Management
Single Level Directory

Objective: Understand the concept of organize the file using single level directory

ALGORITHM:

Step-1: Start the program.
Step-2: Declare the count, file name, graphical interface.
Step-3: Read the number of files
Step-4: Read the file name
Step-5: Declare the root directory
Step-6: Using the file eclipse function define the files in a single level
Step-7: Display the files
Step-8: Stop the program

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<graphics.h>
void main()
{
int gd=DETECT,gm,count,i,j,mid,cir_x;
char fname[10][20];
clrscr();
initgraph(&gd,&gm,"c:/tc/bgi");
cleardevice();
setbkcolor(GREEN);
printf("enter number of files");
scanf("&d",&count);
if(i<count)
// for(i=0;i<count;i++)
{
cleardevice();
setbkcolor(GREEN);
printf("enter %d file name:",i+1);
scanf("%s",fname[i]);
setfillstyle(1,MAGENTA);
mid=640/count;
cir_x=mid/3;
bar3d(270,100,370,150,0,0);
settextstyle(2,0,4);
settextjustify(1,1);
outtextxy(320,125,"root directory");
setcolor(BLUE);
i++;
for(j=0;j<=i;j++,cir_x+=mid)
{
line(320,150,cir_x,250);
fillellipse(cir_x,250,30,30);
outtextxy(cir_x,250,fname[i]);
}}
getch();
}
```