Experiment 1

AIM: To install the Unix/Linux operating system on a computer system and verify a successful installation.

THEORY:

- Linux is a free, open-source, Unix-like operating system built around the Linux kernel. It is widely used for both personal and enterprise computing due to its stability, scalability, and security. The installation process involves preparing a bootable medium (such as a USB drive) containing a Linux distribution like Ubuntu, Fedora, or Debian, and then configuring the system BIOS or UEFI to boot from it.
- During installation, partitions are created for /boot, /home, and /swap directories to organize system data. Post-installation, the user account and network configurations are set up, and the GRUB bootloader ensures the system boots correctly.
- Once installation is complete, verifying the kernel version and distribution information ensures that the operating system is functioning properly. Commands such as uname -r and reading /etc/os-release are commonly used for verification. Keeping the system updated with apt update and apt upgrade ensures security and software integrity.1

PROGRAM:

```
aryann_0101@popos:~$ lsblk
NAME        MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
sda           8:0    0 476.9G  0 disk
├─sda1        8:1    0   512M  0 part  /boot/efi
├─sda2        8:2    0   100G  0 part  /
└─sda3        8:3    0 376.4G  0 part  /home
aryann_0101@popos:~$ uname -r
5.15.0-78-generic
aryann_0101@popos:~$ cat /etc/os-release | grep -E 'NAME=|VERSION='
NAME="Ubuntu"
VERSION="22.04.4 LTS (Jammy Jellyfish)"
aryann_0101@popos:~$ sudo apt update && sudo apt -y upgrade
... package index updates and upgrades ...
```

CONCLUSION:

A Linux distribution was installed and verified successfully. The kernel version, bootloader, and file systems were confirmed operational, establishing a stable platform for further experiments.

Experiment 2

AIM: To study user login and logout information in Unix/Linux.

THEORY:

- User sessions in Linux are tracked using system log files such as /var/run/utmp, /var/log/wtmp, and /var/log/btmp. These files store details about users currently logged in, login histories, and failed login attempts, respectively. Commands like who, w, and last allow users and administrators to view this data in real-time.
- Understanding login records is critical for system security and auditing. Administrators use this data to detect unauthorized access and analyze system usage patterns. Proper session tracking ensures system accountability and helps identify performance bottlenecks caused by active sessions.

PROGRAM:

```
aryann_0101@popos:~$ who
aryan  tty1   2025-10-06 09:30


aryann_0101@popos:~$ last | head
aryan  tty1   Mon Oct 6 09:30   still logged in


aryann_0101@popos:~$ finger aryan
Login: aryan   Name: Aryan Nair
Directory: /home/aryan   Shell: /bin/bash
On since Mon Oct 6 09:30 (IST) on tty1 from :0
No mail. No plan.
```

CONCLUSION:
The system successfully recorded and displayed login/logout information. Commands like who, w, and last are essential for monitoring user activity and ensuring secure access control.

Experiment 3

AIM: To study common Unix/Linux general-purpose utility commands.

THEORY:

- Linux provides numerous command-line utilities for system management, file handling, and process monitoring. Commands such as ls, cp, mv, and rm handle file operations, while ps, kill, and top are used for process management. Administrative commands like chmod and chown manage file permissions and ownership.
- These utilities form the foundation of Linux usage and scripting. The man command provides documentation for every tool, making self-learning easier. Combining commands with pipes (|) and redirection (>, <) enhances productivity and automation. Mastery of these tools is essential for any Linux user or administrator.

PROGRAM:

```
aryann_0101@popos:~$ ls -l

total 8

-rw-r--r-- 1 aryan users 1024 Oct 06 10:00 file1.txt

drwxr-xr-x 2 aryan users 4096 Oct 06 10:10 folder1


aryann_0101@popos:~$ cp file1.txt file2.txt

aryann_0101@popos:~$ mv file2.txt folder1/

aryann_0101@popos:~$ chmod 755 folder1

aryann_0101@popos:~$ date

Mon Oct 6 10:45:31 IST 2025
```

CONCLUSION:

Essential Linux commands for file and process management were successfully executed. Understanding these commands builds a strong foundation for shell scripting and system administration.

Experiment 4

AIM: To study the vi editor and understand its modes and commands.

THEORY:

- The vi editor is one of the oldest and most powerful text editors in Unix/Linux. It operates in two main modes – **Command Mode** for navigation and editing, and **Insert Mode** for text entry. The user switches to insert mode by pressing i and returns to command mode with the Esc key. Commands like :w (save), :q (quit), and :wq (save and exit) are used frequently.
- vi is preferred for its lightweight operation, availability on all systems, and ability to edit configuration files directly within the terminal. It supports text manipulation, search, and replacement using regular expressions.

PROGRAM:

```
aryann_0101@popos:~$ vi sample.txt

-- INSERT --

Hello Linux World

<Esc>

:wq


aryann_0101@popos:~$ cat sample.txt

Hello Linux World
```

CONCLUSION:
The experiment demonstrated the usage of the vi editor for creating and editing files. The user learned essential commands and understood different modes of operation.

Experiment 5

AIM: To install and implement Docker on Linux.

THEORY:

- Docker is a containerization platform that enables developers to package applications and dependencies into portable units called containers. These containers share the host operating system kernel but remain isolated, improving scalability and deployment efficiency.
- Installing Docker on Linux requires enabling repositories, installing the Docker engine, and starting the Docker service. Once installed, users can pull container images from Docker Hub and run them using the docker run command. This approach streamlines deployment and ensures consistent environments across development and production.

PROGRAM:

```
aryann_0101@popos:~$ sudo apt update

aryann_0101@popos:~$ sudo apt install docker.io -y

aryann_0101@popos:~$ sudo systemctl enable --now docker

aryann_0101@popos:~$ docker --version

Docker version 27.0.3, build a20e3a

aryann_0101@popos:~$ sudo docker run hello-world

Hello from Docker!
```

CONCLUSION:

Docker was successfully installed and verified using a test container. The experiment demonstrated the basics of container management and system-level virtualization.

Experiment 6

AIM: To study the Bash shell, Bourne shell (sh), and C shell (csh) in Unix/Linux operating systems.

THEORY:

- A shell is a command-line interpreter that serves as the user interface to the Unix/Linux kernel. It reads commands from the user and executes them either interactively or through shell scripts.
- The **Bourne shell (sh)**, developed at Bell Labs, was the original Unix shell providing control structures, loops, and basic scripting capabilities.
- The **Bash (Bourne Again Shell)** is an enhanced version of sh that includes command history, command-line editing, arrays, arithmetic operations, and improved scripting features. It is the default shell in most modern Linux distributions.
- On the other hand, the **C shell (csh)** introduced a C-like syntax and better interactive features such as aliases and job control, which were later extended in **tcsh**.
- Understanding different shells helps users write portable scripts and choose appropriate environments for automation tasks. Each shell offers unique syntax, prompting, and environment variable handling mechanisms.

PROGRAM:

```
aryann_0101@popos:~$ echo $SHELL
/bin/bash

aryann_0101@popos:~$ bash --version | head -n 1
GNU bash, version 5.1.16(1)-release

aryann_0101@popos:~$ sh -c 'echo Hello from Bourne shell'
Hello from Bourne shell

aryann_0101@popos:~$ csh -c 'echo Hello from C shell'
Hello from C shell
```

CONCLUSION:

Different Unix shells were studied and executed successfully. Bash provides the most advanced features and is the preferred shell for scripting and general use in Linux environments.

Experiment 7

AIM: To study the Unix/Linux file system tree structure.

THEORY:

- Linux organizes all files and directories in a hierarchical structure that begins at the root directory (/). This unified tree structure ensures that all devices, files, and partitions appear under a single root hierarchy. Standard directories have specific purposes as defined by the **Filesystem Hierarchy Standard (FHS).**
- For example, /bin and /sbin store essential system binaries, /etc contains configuration files, /home holds user directories, /usr stores user applications and libraries, /var maintains variable data like logs and mail, and /tmp is used for temporary files. Understanding this structure helps in effective navigation, troubleshooting, and system management.
- The file system also includes pseudo-directories such as /proc (which contains runtime kernel information) and /sys (which exposes kernel device information). Together, these make Linux highly organized and modular.

PROGRAM:

```
aryann_0101@popos:~$ ls /
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt
proc  root  run  sbin  srv  sys  tmp  usr  var
aryann_0101@popos:~$ tree -L 1 /
/
├── bin
├── boot
├── dev
├── etc
├── home
├── lib
├── lib64
├── media
├── mnt
├── opt
├── proc
├── root
├── run
```

```
├── sbin
├── srv
├── sys
├── tmp
├── usr
└── var
```

CONCLUSION:

The Linux file system follows a tree-like hierarchy beginning at the root directory. Each subdirectory serves a defined purpose, ensuring structured organization of system files and user data.

Experiment 8

AIM: To study .bashrc, /etc/bashrc, and environment variables in Unix/Linux.

THEORY:

- When a user logs into a Linux system, several startup scripts are executed to configure the shell environment. The file **/etc/bashrc** (or /etc/bash.bashrc in some distributions) sets global defaults for all users, while each user's **~/.bashrc** contains personal settings such as aliases, functions, and environment variables.
- **Environment variables** define values that affect the behavior of processes, such as PATH, HOME, USER, SHELL, and LANG. They determine the user's home directory, default shell, executable search paths, and language preferences. Using export makes a variable available to child processes.
- Editing .bashrc allows users to customize their environment — for example, adding aliases (alias ll='ls -alF') or changing the prompt. Understanding these files is essential for system customization and troubleshooting.

PROGRAM:

```
aryann_0101@popos:~$ head -n 5 ~/.bashrc

alias ll='ls -alF'

export EDITOR=vim

PS1='\u@\h:\w\$ '


aryann_0101@popos:~$ env | grep -E 'PATH|HOME|USER|SHELL'

HOME=/home/aryan

PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

USER=aryan

SHELL=/bin/bash
```

CONCLUSION:

The .bashrc and /etc/bashrc files were examined successfully. Environment variables were viewed and understood as fundamental elements defining system behavior.

Experiment 9

AIM: To write a shell script program to display the list of users currently logged in.

THEORY:

- The who command displays information about users currently logged into the system by reading from the /var/run/utmp file. Automating this command in a shell script makes it easier to monitor active sessions periodically or record session information in log files.
- Shell scripting allows combining system commands and formatting outputs efficiently. Adding timestamps with the date command improves monitoring and audit capabilities for administrators.

PROGRAM:

```
aryann_0101@popos:~$ cat logged_users.sh

#!/bin/bash

echo "-----------------------------------------------"

echo "Current Logged-In Users on $(date)"

echo "-----------------------------------------------"

who


aryann_0101@popos:~$ chmod +x logged_users.sh

aryann_0101@popos:~$ ./logged_users.sh

-----------------------------------------------

Current Logged-In Users on Mon Oct 6 11:15:22 IST 2025

-----------------------------------------------

aryan  tty1  2025-10-06 09:30

root      pts/0 2025-10-06 11:10 (:0)
```

CONCLUSION:

The script successfully displayed all active user sessions. Automating the who command helps in user management and monitoring system access.

Experiment 10

AIM: To write a shell script program that displays “HELLO WORLD”.

THEORY:

- A shell script is a sequence of Linux commands written in a text file and executed by a shell interpreter. Every script starts with a **shebang (#!)** line indicating which shell should interpret the commands. Using the echo command allows printing messages or variables to the console.
- The “HELLO WORLD” program is traditionally used to introduce new users to programming and scripting syntax. It demonstrates how to create, save, give execution permission, and run a shell script file. This simple example lays the foundation for understanding automation using shell scripts.

PROGRAM:

```
aryann_0101@popos:~$ cat hello.sh

#!/bin/bash

echo "HELLO WORLD"


aryann_0101@popos:~$ chmod +x hello.sh

aryann_0101@popos:~$ ./hello.sh

HELLO WORLD
```

CONCLUSION:
The script was created, executed, and verified successfully. This simple exercise demonstrated the process of writing and running shell scripts in Linux.

Experiment 11

AIM: To write a shell script program to develop a scientific calculator using bc.

THEORY:

- Linux provides the **bc (Basic Calculator)** utility, which supports arbitrary precision arithmetic and mathematical functions. It is often used in shell scripts for performing floating-point calculations, as bash itself handles only integer arithmetic. By using the -l option, scientific functions like sine (s()), cosine (c()), logarithm (l()), and exponential (e()) can be accessed.
- Shell scripts can read expressions from the user and evaluate them by passing them to bc using pipes. This allows for the creation of flexible calculators that handle real numbers, parentheses, and functions. Understanding how to combine user input, variables, and pipes is essential for command-line automation.

PROGRAM:

```
aryann_0101@popos:~$ cat calc.sh

#!/bin/bash

echo "Enter an expression (e.g., 5+7 or s(1)+3/2):"

read exp

echo "scale=5; $exp" | bc -l


aryann_0101@popos:~$ chmod +x calc.sh

aryann_0101@popos:~$ ./calc.sh

Enter an expression (e.g., 5+7 or s(1)+3/2):

s(1)+3/2

2.34147
```

CONCLUSION:

The scientific calculator was successfully implemented using the bc utility. The script demonstrated reading user input, processing expressions, and performing floating-point operations.

Experiment 12

AIM: To write a shell script program to check whether a given number is even or odd.

THEORY:

- Determining whether a number is even or odd can be done using the **modulus operator (%)**, which gives the remainder of a division. If a number divided by 2 leaves a remainder of 0, it is even; otherwise, it is odd. Shell arithmetic expansion $(( )) allows performing such operations easily.
- Using input validation ensures that only valid numeric values are processed. Conditional statements like if-else are used to test logical expressions and control the flow based on results.

PROGRAM:

```
aryann_0101@popos:~$ cat evenodd.sh

#!/bin/bash

read -p "Enter a number: " n

if [ $((n % 2)) -eq 0 ]

then

    echo "The number $n is EVEN"

else

    echo "The number $n is ODD"

fi


aryann_0101@popos:~$ chmod +x evenodd.sh

aryann_0101@popos:~$ ./evenodd.sh

Enter a number: 9

The number 9 is ODD
```

CONCLUSION:

The script successfully determined the parity of the input number using arithmetic and conditional logic.

Experiment 13

AIM: To write a shell script program to search whether an element is present in a list or not.

THEORY:

- Arrays in bash allow storing multiple values under one variable name. The for loop can iterate through array elements, and string comparison (=) is used to check if a value matches the search term. This method is known as a **linear search**, where each element is checked one by one.
- This type of scripting is useful in data handling, automation, and verifying entries in lists such as usernames or filenames. Proper quoting of array elements ensures compatibility even when values contain spaces.

PROGRAM:

```
aryann_0101@popos:~$ cat search_list.sh
#!/bin/bash
echo "Enter space-separated elements:"
read -a arr
read -p "Enter value to search: " val
found=0
for x in "${arr[@]}"
do
    if [ "$x" = "$val" ]; then
        found=1
        break
    fi
done
[ $found -eq 1 ] && echo "$val FOUND" || echo "$val NOT FOUND"


aryann_0101@popos:~$ chmod +x search_list.sh
aryann_0101@popos:~$ ./search_list.sh
Enter space-separated elements:
apple banana cherry mango
Enter value to search: banana
banana FOUND
```

CONCLUSION:

The script successfully implemented a linear search over user-input arrays, demonstrating conditional comparison and looping in bash.

Experiment 14

AIM: To write a shell script program to check whether a given file is a directory or not.

THEORY:

- Bash provides several **file test operators** to determine file types and properties. The operator -d checks whether a given path points to a directory, while -f checks for a regular file. Such tests are crucial in automation scripts to prevent errors and improve decision-making.
- File tests are often used in backup, monitoring, and maintenance scripts to verify valid paths before executing file operations. The combination of input reading, validation, and logical branching makes the script robust and reliable.

PROGRAM:

```
aryann_0101@popos:~$ cat check_dir.sh

#!/bin/bash

read -p "Enter a path: " path

if [ -d "$path" ]

then

    echo "$path is a DIRECTORY"

else

    echo "$path is NOT a directory"

fi


aryann_0101@popos:~$ chmod +x check_dir.sh

aryann_0101@popos:~$ ./check_dir.sh

Enter a path: /etc

/etc is a DIRECTORY
```

CONCLUSION:

The script successfully verified the given path using the -d test operator. This is useful in file validation and directory management tasks.

Experiment 15

AIM: To write a shell script program to count the number of files in a directory.

THEORY:

- The number of files in a directory can be determined using a combination of Linux commands. The ls -1 command lists files line by line, and piping this output to wc -l counts the number of lines, effectively giving the file count. Alternatively, find with -maxdepth and -type f can be used to count files more reliably.
- This type of script helps in system auditing and maintenance, providing insights into storage usage and directory organization. Incorporating validation ensures the script handles invalid or non-existent paths gracefully.

PROGRAM:

```
aryann_0101@popos:~$ cat count_files.sh

#!/bin/bash

read -p "Enter directory path: " dir

if [ ! -d "$dir" ]; then

    echo "Invalid directory."

    exit 1

fi

count=$(find "$dir" -maxdepth 1 -type f | wc -l)

echo "Number of files in $dir: $count"


aryann_0101@popos:~$ chmod +x count_files.sh

aryann_0101@popos:~$ ./count_files.sh

Enter directory path: /var/log

Number of files in /var/log: 36
```

CONCLUSION:

The script successfully counted the files using find and wc. Such automation assists in tracking storage and directory management.

Experiment 16

AIM: To write a shell script program to copy the contents of one file to another.

THEORY:

- Copying files is one of the most fundamental operations in Unix/Linux. The cp command is used to duplicate files and directories. Alternatively, cat with output redirection (>) can also be used to copy file contents from one file to another.
- This experiment demonstrates file handling, path validation, and redirection concepts in bash scripting. Ensuring that the source file exists and the target file is writable prevents data loss and enhances reliability.

PROGRAM:

```
aryann_0101@popos:~$ cat copy_file.sh
#!/bin/bash
read -p "Enter source file: " src
read -p "Enter destination file: " dest
if [ -f "$src" ]; then
    cp "$src" "$dest"
    echo "Copied $src to $dest successfully."
else
    echo "Source file not found!"
fi


aryann_0101@popos:~$ chmod +x copy_file.sh
aryann_0101@popos:~$ echo "Linux Practical File" > file1.txt
aryann_0101@popos:~$ ./copy_file.sh
Enter source file: file1.txt
Enter destination file: file2.txt
Copied file1.txt to file2.txt successfully.


aryann_0101@popos:~$ cat file2.txt
Linux Practical File
```

Conclusion: This experiment demonstrated how to write a shell script program to copy the contents of one file to another.

Experiment 17

AIM: To create a directory, write contents into it, and copy it to a suitable location in the home directory.

THEORY:

- Linux allows users to create directories using the mkdir command and copy them recursively using cp -r. Directories are fundamental for organizing files and projects. Automating these tasks through shell scripting helps in deployment, backups, and workspace setup.
- This experiment demonstrates directory creation, writing data into files, and recursive copying operations while maintaining file structure and permissions.

PROGRAM:

```
aryann_0101@popos:~$ mkdir project_dir

aryann_0101@popos:~$ echo "This is my Linux Lab Project" > project_dir/readme.txt

aryann_0101@popos:~$ cp -r project_dir ~/backup_project_dir

aryann_0101@popos:~$ ls ~/backup_project_dir

readme.txt
```

CONCLUSION:

The experiment demonstrated directory creation, writing files, and recursive copying. Such operations are vital in organizing and backing up data in Linux environments.

Experiment 18

AIM: To use a pipeline and command substitution to set the length of a line in a file to a variable.

THEORY:

- Pipelines (|) in Linux connect the output of one command to the input of another, allowing data transformation in a single line. Command substitution ($(...)) captures command output and assigns it to a variable. Together, they form the core of bash automation and scripting.
- Using these concepts, users can store computed values like word count, line length, or process statistics in variables and use them dynamically within scripts.

PROGRAM:

```
aryann_0101@popos:~$ echo "Hello Linux World" > text.txt

aryann_0101@popos:~$ len=$(cat text.txt | wc -c)

aryann_0101@popos:~$ echo "The line length is $len characters."

The line length is 18 characters.
```

CONCLUSION:
This experiment demonstrated command substitution and pipelines effectively. It showed how shell scripts can capture command outputs into variables for further processing.

Experiment 19

AIM: To write a program using sed command to print duplicated lines of input.

THEORY:

- The sed (stream editor) command is used to process and transform text streams line by line. It supports search, replace, delete, and pattern-based filtering operations using regular expressions. Detecting duplicate lines in a file can be achieved by sorting data first and then using sed or uniq -d to identify repeated entries.
- This method is useful in data cleanup, log analysis, and text preprocessing. sed scripts use hold and pattern spaces to compare adjacent lines and process them according to match rules.

PROGRAM:

```
aryann_0101@popos:~$ cat data.txt
alpha
beta
gamma
alpha
beta
delta


aryann_0101@popos:~$ sort data.txt | sed 'N; s/^\(.*\)\n\1$/\1/; t dup; D; :dup; p; D'
alpha
beta
```

CONCLUSION:

The sed command successfully identified and printed duplicate lines from a sorted list, illustrating text stream processing and pattern matching.

Experiment 20

AIM: To study the process of writing a device driver or kernel module in Linux.

THEORY:

- A **Linux device driver** is a piece of software that allows the operating system to communicate with hardware components. Kernel modules can be dynamically loaded into or removed from the running kernel without rebooting, making them ideal for testing and device management.
- Writing a kernel module involves implementing initialization (module_init) and cleanup (module_exit) functions. These functions are compiled into object files using kernel headers and built using make with a Kbuild system. Tools like insmod, rmmod, and dmesg are used to load, unload, and monitor kernel messages.
- This experiment helps understand how the Linux kernel interacts with modules, memory, and device control mechanisms, providing foundational knowledge for systems programming.

PROGRAM:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>

static int __init hello_init(void) {
    printk(KERN_INFO "aryann_0101@popos: Kernel Module Loaded Successfully!\n");
    return 0;
}

static void __exit hello_exit(void) {
    printk(KERN_INFO "aryann_0101@popos: Kernel Module Removed.\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Aryan");
MODULE_DESCRIPTION("A Simple Linux Kernel Module");
```

Execution Steps:

```
aryann_0101@popos:~$ make -C /lib/modules/$(uname -r)/build M=$(pwd) modules
aryann_0101@popos:~$ sudo insmod hello.ko
aryann_0101@popos:~$ dmesg | tail
 [ 3456.123 ] aryann_0101@popos: Kernel Module Loaded Successfully!
aryann_0101@popos:~$ sudo rmmod hello
aryann_0101@popos:~$ dmesg | tail
[ 3460.987 ] aryann_0101@popos: Kernel Module Removed.
```

CONCLUSION:

A simple Linux kernel module was developed, compiled, and executed successfully. This experiment provided insights into kernel-space programming, module management, and interaction with system logs.