QUEST FOR EXCELLENCE	**MAHARASHTRA INSTITUTE OF TECHNOLOGY AURANGABAD**	**LABORATORY MANUAL**
	PRACTICAL EXPERIMENT INSTRUCTION SHEET	
	DEPARTMENT: EMERGING SCIENCE & TECHNOLOGY	

LABORATORY : MIT-EST-OCC	A.Y.: 2023-24

Class: TY	PART: II	SUBJECT: AID373 Operating System

Experiment No:1

Aim: Installation of Windows/Linux OS.

Objective:

The workshop aims to teach participants how to install and configure Windows/Linux operating systems on personal computers or servers.

Outcomes:

Participants will learn the steps to install Windows/Linux OS.

Participants will gain hands-on experience in installation.

Participants will understand basic configuration and troubleshooting.

Pre-requisite:

Basic understanding of computer hardware.

Familiarity with BIOS/UEFI settings.

Ability to use a USB flash drive.

Theory:

Windows Installation

1. Prepare Installation Media:
    - Download the Windows installation media creation tool from the Microsoft website.
    - Insert a USB flash drive (at least 8GB) into your computer.
    - Run the media creation tool and follow the prompts to create a bootable USB drive with Windows installation files.
2. Backup Important Data:
    - Before proceeding, it's a good idea to back up any important data on your computer to avoid data loss.
3. Access BIOS/UEFI:

- Restart your computer and access the BIOS or UEFI firmware settings. This is usually done by pressing a key (like F2, F12, Del, Esc) during the boot process.
- Look for the boot order settings and set the USB drive as the first boot device.

4. Boot from USB:
   - Save the BIOS/UEFI settings and restart your computer.
   - When the computer restarts, it should boot from the USB drive. If not, you may need to repeat the previous step to ensure the USB drive is set as the primary boot device.
5. Begin Windows Installation:
   - The Windows installation screen should appear. Select your language, time and currency format, and keyboard or input method, then click "Next."
   - Click "Install Now" to begin the installation process.
6. Enter Product Key:
   - Enter your Windows product key if prompted. If you don't have a product key, you can choose to skip this step for now and enter it later.
7. Accept License Terms:
   - Read and accept the license terms, then click "Next."
8. Choose Installation Type:
   - Select "Custom: Install Windows only (advanced)" as the installation type.
9. Partition Hard Drive:
   - Select the drive where you want to install Windows and click "Next." If the drive is unallocated, you can create a new partition by clicking "New" and specifying the size.
10. Install Windows:
    - Windows will begin installing. The installation process may take some time, so be patient.
11. Complete Installation:
    - Once the installation is complete, your computer will restart. Remove the USB drive when prompted.

12. Set Up Windows:
    - Follow the on-screen instructions to complete the setup process, including setting up your user account and choosing your settings.
13. Install Drivers and Updates:
    - After Windows is installed, it's a good idea to install drivers for your hardware and update Windows to ensure you have the latest security patches and features.

You've successfully installed Windows on your PC.

Linux Installation:

1. **Choose a Linux Distribution:**
    - There are many Linux distributions available, such as Ubuntu, Fedora, and Linux Mint. Choose one that suits your needs and download the ISO file from the distribution's official website.
2. **Create Installation Media:**
    - Insert a USB flash drive (at least 4GB) into your computer.
    - Download and install a tool like Rufus (for Windows) or Etcher (for macOS and Linux) to create a bootable USB drive with the Linux ISO file.
    - Use the tool to select the Linux ISO file and the USB drive, then create the bootable drive.
3. **Backup Important Data:**
    - Before proceeding, it's a good idea to back up any important data on your computer to avoid data loss.
4. **Access BIOS/UEFI:**
    - Restart your computer and access the BIOS or UEFI firmware settings. This is usually done by pressing a key (like F2, F12, Del, Esc) during the boot process.
    - Look for the boot order settings and set the USB drive as the first boot device.
5. **Boot from USB:**
    - Save the BIOS/UEFI settings and restart your computer.

- When the computer restarts, it should boot from the USB drive. If not, you may need to repeat the previous step to ensure the USB drive is set as the primary boot device.

6. **Begin Linux Installation:**
   - The Linux installation screen should appear. Select your language and click "Install" or "Try Linux" to begin the installation process.
7. **Choose Installation Options:**
   - Follow the on-screen instructions to choose your installation options, such as language, keyboard layout, and installation type (e.g., erase disk and install Linux, install alongside existing operating system).
8. **Partition Hard Drive:**
   - If you choose to manually partition the disk, you'll need to create partitions for the root directory ("/"), swap space (optional but recommended), and any other partitions you require (e.g., /home for user data).
9. **Configure Installation:**
   - Follow the on-screen prompts to configure your installation, such as setting your time zone, creating a user account, and choosing a password.
10. **Install Linux:**
    - Click "Install" to begin the installation process. The installation may take some time, so be patient.
11. **Complete Installation:**
    - Once the installation is complete, you'll be prompted to restart your computer. Remove the USB drive when prompted.
12. **Set Up Linux:**
    - Follow the on-screen instructions to set up your Linux system, including configuring your user account and choosing your settings.
13. **Install Updates and Drivers:**
    - After Linux is installed, it's a good idea to install any updates and drivers needed for your hardware to ensure everything is working correctly.

You've successfully installed Linux on your PC.

	MAHARASHTRA INSTITUTE OF TECHNOLOGY AURANGABAD	LABORATORY MANUAL
QUEST FOR EXCELLENCE	PRACTICAL EXPERIMENT INSTRUCTION SHEET	
	DEPARTMENT: EMERGING SCIENCE & TECHNOLOGY	

LABORATORY : MIT-EST-OCC	A.Y.: 2023-24

Class: TY	PART: II	SUBJECT: AID373 Operating System

Conclusion:

PREPARED BY: Ms. Priyanka Sonwane, Ms. Suchita Waghmare (Subject Teacher)	**APPROVED BY** : Kavita V. Bhosle HESTD

QUEST FOR EXCELLENCE	**MAHARASHTRA INSTITUTE OF TECHNOLOGY AURANGABAD**	**LABORATORY MANUAL**
	PRACTICAL EXPERIMENT INSTRUCTION SHEET	
	DEPARTMENT: EMERGING SCIENCE & TECHNOLOGY	

LABORATORY : MIT-EST-OCC	A.Y.: 2023-24

Class: TY	PART: II	SUBJECT: AID373 Operating System

Experiment No:2

Aim: Hands-on Unix/Linux basic commands.

Objective:

The objective of the workshop is to introduce participants to basic Unix/Linux commands and command-line navigation.

Outcomes:

- Participants will learn essential Unix/Linux commands for file manipulation, directory navigation, and system management.
- Participants will gain hands-on experience using the command line to perform common tasks.
- Participants will be able to use Unix/Linux commands effectively in a terminal environment.

Pre-requisite:

- Basic familiarity with the Unix/Linux operating system.
- Understanding of basic computer concepts and file systems.
- Ability to open and use a terminal or command prompt.

Theory:

LINUX:

It is like UNIX, which is created by Linus Torvalds. All UNIX commands work in Linux. Linux is an open source software. The main feature of Linux is coexisting with other OS such as Windows and UNIX.

STRUCTURE OF A LINUX SYSTEM:

It consists of three parts.

a) UNIX kernel

b) Shells

c) Tools and Applications

PREPARED BY: Ms. Priyanka Sonwane, Ms. Suchita Waghmare (Subject Teacher)	**APPROVED BY** : Kavita V. Bhosle HESTD

UNIX KERNEL:

The kernel is the core of the UNIX OS. It controls all tasks, schedules all Processes, and carries out all the functions of the OS.

Decides when one program tops and another starts.

SHELL:

Shell is the command interpreter in the UNIX OS. It accepts commands from the user and analyses and interprets them

1. **echo**:
   a. Prints arguments to the screen.
   b. Example: **echo "Hello, World!"**
2. **grep**:
   a. Searches for patterns in files.
   b. Example: **grep "pattern" file.txt**
3. **awk**:
   a. A pattern scanning and processing language.
   b. Example: **awk '{print $1}' file.txt** (prints the first column of a file)
4. **sed**:
   a. A stream editor for filtering and transforming text.
   b. Example: **sed 's/old/new/g' file.txt** (replaces "old" with "new" in a file)
5. **chmod**:
   a. Changes file permissions.
   b. Example: **chmod +x script.sh** (makes a script executable)
6. **chown**:
   a. Changes file ownership.
   b. Example: **chown user:group file.txt** (changes the owner and group of a file)
7. **ps**:

   a. Displays information about running processes.
   b. Example: **ps aux** (displays a list of all processes)

8. **kill**:
   a. Sends signals to processes, typically to terminate them.
   b. Example: **kill PID** (terminates a process with the specified PID)
9. **tar**:
   a. Creates or extracts files from a tape archive (tar file).
   b. Example: **tar -cvf archive.tar files** (creates a tar archive)
10. **find**:
    a. Searches for files and directories in a directory hierarchy.
    b. Example: **find . -name "*.txt"** (finds all files with a .txt extension)

11. **pwd** (Print Working Directory):
    a. Syntax: **pwd**
12. **ls** (List):
    a. Syntax: **ls [options] [directory]**
    b. Example: **ls -l** (list files in long format), **ls -a** (list all files including hidden files)
13. **cd** (Change Directory):
    a. Syntax: **cd [directory]**
    b. Example: **cd Documents** (change to the "Documents" directory)
14. **mkdir** (Make Directory):
    a. Syntax: **mkdir [directory]**
    b. Example: **mkdir new_dir** (create a directory named "new_dir")
15. **rmdir** (Remove Directory):
    a. Syntax: **rmdir [directory]**

   b. Example: **rmdir old_dir** (remove the "old_dir" directory)

16. **touch**:
    a. Syntax: **touch [file]**
    b. Example: **touch new_file.txt** (create a new empty file)
17. **cp** (Copy):
    a. Syntax: **cp [options] source destination**
    b. Example: **cp file1.txt file2.txt** (copy "file1.txt" to "file2.txt")
18. **mv** (Move):
    a. Syntax: **mv [options] source destination**
    b. Example: **mv file1.txt dir1/** (move "file1.txt" to the "dir1" directory)
19. **rm** (Remove):
    a. Syntax: **rm [options] file**
    b. Example: **rm file.txt** (delete "file.txt")
20. **cat** (Concatenate):
    a. Syntax: **cat [file]**
    b. Example: **cat file.txt** (display the contents of "file.txt")
21. **man** (Manual):
    a. Syntax: **man [command]**
    b. Example: **man ls** (display the manual for the "ls" command)

FILE MANIPULATION COMMANDS

File manipulation commands in Unix/Linux are used to create, view, modify, and delete files. Here are some common file manipulation commands along with their syntax:

1. **touch**:
   - Syntax: **touch [file]**

- Description: Creates an empty file or updates the access and modification times of a file.
- Example: **touch new_file.txt**

2. **cat** (Concatenate):
   - Syntax: **cat [file]**
   - Description: Displays the contents of a file.
   - Example: **cat file.txt**
3. **more** and **less**:
   - Syntax: **more [file]** or **less [file]**
   - Description: Allows you to view the contents of a file one screen at a time.
   - Example: **more file.txt** or **less file.txt**
4. **cp** (Copy):
   - Syntax: **cp [options] source destination**
   - Description: Copies files and directories.
   - Example: **cp file1.txt file2.txt**
5. **mv** (Move):
   - Syntax: **mv [options] source destination**
   - Description: Moves or renames files and directories.
   - Example: **mv file1.txt dir1/** or **mv old_file.txt new_file.txt**
6. **rm** (Remove):
   - Syntax: **rm [options] file**
   - Description: Deletes files.
   - Example: **rm file.txt**
7. **mkdir** (Make Directory):
   - Syntax: **mkdir [directory]**
   - Description: Creates a new directory.

- Example: **mkdir new_dir**

8. **rmdir** (Remove Directory):
    - Syntax: **rmdir [directory]**
    - Description: Deletes an empty directory.
    - Example: **rmdir old_dir**
9. **chmod** (Change Mode):
    - Syntax: **chmod [options] mode file**
    - Description: Changes the permissions of a file.
    - Example: **chmod +x script.sh** (makes a script executable)
10. **chown** (Change Owner):
    - Syntax: **chown [options] user:group file**
    - Description: Changes the owner and group of a file.
    - Example: **chown user1:group1 file.txt**

VI/ VIM Editor

Vi- vi stands for "visual".vi is the most important and powerful editor.vi is a full screen editor that allows users to view and edit entire documents at the same time.vi editor was written at the University of California, at Berkley by Bill Joy, who is one of the co-founders of Sun Microsystems.

Features of vi:

It is easy to learn and has more powerful features.

Itworksgreatspeedandiscasesensitive.vihaspowerfulundofunctionsandhas3modes

1. Command mode

2. Insert mode
3. Escape or ex mode

1. **Opening a File**:
    - Syntax: **vi filename**
    - Description: Opens the specified file in VI/VIM.
2. **Switching Modes**:
    - To switch from command mode to insert mode, press **i** for insert mode.
    - To switch from insert mode to command mode, press **Esc**.
3. **Saving a File**:
    - In command mode, type **:w** and press **Enter** to save changes to the file.
4. **Quitting VI/VIM**:
    - In command mode, type **:q** and press **Enter** to quit VI/VIM.
    - If there are unsaved changes, use **:q!** to quit without saving.
5. **Saving and Quitting**:
    - To save changes and quit, use **:wq** in command mode.
6. **Moving the Cursor**:
    - Use arrow keys to move the cursor.
    - To move to the beginning of a line, press **0**.
    - To move to the end of a line, press **$**.
7. **Editing Text**:
    - In command mode, use **x** to delete the character under the cursor.
    - Use **dd** to delete the entire line.
    - Press **p** to paste deleted or yanked (copied) text.
8. **Searching for Text**:
    - To search for text, press / followed by the text to search for, then press **Enter**.

- To repeat the search in the forward direction, press **n**.

9. **Replacing Text**:
   - To replace a single character, move the cursor over the character and press **r** followed by the replacement character.
   - To replace text, press **:** to enter command mode, then use **%s/old_text/new_text/g** to replace all occurrences of **old_text** with **new_text**.
10. **Undo and Redo**:
    - To undo the last change, press **u** in command mode.
    - To redo a change that was undone, press **Ctrl** + **r**.

Conclusion:

Experiment No: 3

AIM: Implementation of FCFS CPU scheduling algorithms

Objective:

- To schedule processes based on their arrival times, executing them in the order they arrive.

Outcomes:

- Processes are executed in the order they arrive, regardless of their burst times.
- Easy to implement and understand.

Pre-requisite:

- Basic understanding of CPU scheduling algorithms.
- Knowledge of process arrival times and burst times.

THEROY:

- Simplest CPU scheduling algorithm that schedules according to arrival times of processes. The first come first serve scheduling algorithm states that the process that requests the CPU first is allocated the CPU first.

- It is implemented by using the FIFO queue. When a process enters the ready queue, its PCB is linked to the tail of the queue.

- When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. FCFS is a non-preemptive scheduling algorithm.

- Characteristics of FCFS

    - Tasks are always executed on a First-come, First-serve concept.

- FCFS is easy to implement and use.
- This algorithm is not very efficient in performance, and the wait time is quite high.

- Advantages of FCFS

  - The simplest and basic form of CPU Scheduling algorithm
  - Easy to implement
  - First come first serve method
  - It is well suited for batch systems where the longer time periods for each process are often acceptable.

- Disadvantages of FCFS

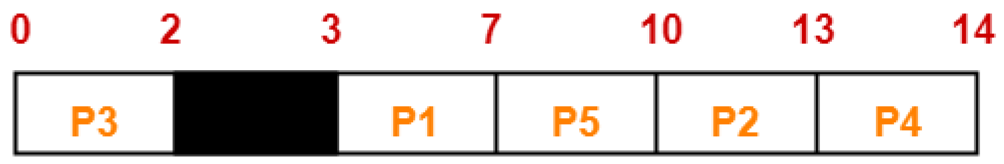
  - As it is a Non-preemptive CPU Scheduling Algorithm, hence it will run till it finishes the execution.
  - The average waiting time in the FCFS is much higher than in the others
  - It suffers from the Convoy effect.
  - Processes that are at the end of the queue, have to wait longer to finish.
  - It is not suitable for time-sharing operating systems where each process should get the same amount of CPU time.

EXAMPLE :

Consider the set of 5 processes whose arrival time and burst time are given below.

Process Id	**Arrival time**	**Burst time**
P1	3	4
P2	5	3
P3	0	2
P4	5	1
P5	4	3

Gantt Chart-



Gantt Chart

Now, we know-

- Turn Around time = Exit time – Arrival time
- Waiting time = Turn Around time – Burst time

Process Id	**Exit time**	**Turn Around time**	**Waiting time**
P1	7	7 – 3 = 4	4 – 4 = 0
P2	13	13 – 5 = 8	8 – 3 = 5
P3	2	2 – 0 = 2	2 – 2 = 0
P4	14	14 – 5 = 9	9 – 1 = 8
P5	10	10 – 4 = 6	6 – 3 = 3

Now,

- Average Turn Around time = (4 + 8 + 2 + 9 + 6) / 5 = 29 / 5 = 5.8 unit
- Average waiting time = (0 + 5 + 0 + 8 + 3) / 5 = 16 / 5 = 3.2 unit

Program:

```
#include<stdio.h>
#include<conio.h>
int main()
{
int bt[20], wt[20], tat[20], i, n;
float wtavg, tatavg;
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
```

```
printf("\nEnter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n);
getch();
}
```

O/P

Conclusion

Experiment No: 4

AIM: Implementation of SJF CPU scheduling algorithms

Objective:

- To schedule processes based on their burst times, executing the shortest job first.

Outcomes:

- Reduces average waiting time and turnaround time compared to FCFS.
- May lead to starvation for longer jobs if short jobs keep coming.

Pre-requisite:

- Understanding of CPU scheduling algorithms.
- Knowledge of process burst times.

THEORY:

- The shortest job first (SJF) or shortest job first, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJF, also known as Shortest Job Next (SJN), can be pre-emptive or non-pre-emptive.

- Characteristics of SJF Scheduling:

    - Shortest Job first has the advantage of having a minimum average waiting time among all scheduling algorithms.
    - It is a Greedy Algorithm.
    - It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.
    - It is practically infeasible as Operating System may not know burst times and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times.

- SJF can be used in specialized environments where accurate estimates of running time are available.

- Algorithm:

  - Sort all the processes according to the arrival time.
  - Then select that process that has minimum arrival time and minimum Burst time.
  - After completion of the process make a pool of processes that arrives afterward till the completion of the previous process and select that process among the pool which is having minimum Burst time.

- Advantages of SJF:

  - SJF is better than the First come first serve(FCFS) algorithm as it reduces the average waiting time.
  - SJF is generally used for long term scheduling
  - It is suitable for the jobs running in batches, where run times are already known.

- Disadvantages of SJF:

  - SJF may cause very long turn-around times or starvation.
  - In SJF job completion time must be known earlier, but sometimes it is hard to predict.
  - Sometimes, it is complicated to predict the length of the upcoming CPU request.
  - It leads to the starvation that does not reduce average turnaround time.

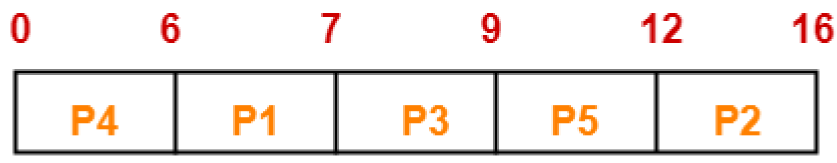
EXAMPLE

Consider the set of 5 processes whose arrival time and burst time are given below.

Process Id	**Arrival time**	**Burst time**
P1	3	1
P2	1	4
P3	4	2
P4	0	6
P5	2	3

Solution-

Gantt Chart-



Gantt Chart

Process Id	**Exit time**	**Turn Around time**	**Waiting time**
P1	7	7 – 3 = 4	4 – 1 = 3
P2	16	16 – 1 = 15	15 – 4 = 11

P3	9	9 – 4 = 5	5 – 2 = 3
P4	6	6 – 0 = 6	6 – 6 = 0
P5	12	12 – 2 = 10	10 – 3 = 7

- Average Turn Around time = (4 + 15 + 5 + 6 + 10) / 5 = 40 / 5 = 8 unit
- Average waiting time = (3 + 11 + 3 + 0 + 7) / 5 = 24 / 5 = 4.8 unit

Program:

```
#include<stdio.h>
#include<conio.h>
int main()
{
int p[20], bt[20], wt[20], tat[20], i, k, n, temp;
float wtavg, tatavg;
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
p[i]=i;
printf("Enter Burst Time for Process %d -- ", i);
scanf("%d", &bt[i]);
}
for(i=0;i<n;i++)
for(k=i+1;k<n;k++)
if(bt[i]>bt[k])
{
temp=bt[i];
bt[i]=bt[k];
bt[k]=temp;
temp=p[i];
p[i]=p[k];
p[k]=temp;
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0];
for(i=1;i<n;i++)
{
wt[i] = wt[i-1] +bt[i-1];
tat[i] = tat[i-1] +bt[i];
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n);
getch();
```

```
}
```

O/P

Conclusion

Experiment No: 5

Aim: Implement producer-consumer problem with bounded buffer solution with Semaphore.

Objective:

- To synchronize access to a shared, bounded buffer between multiple producer and consumer threads/processes using semaphores.

Outcomes:

- Ensure that producers wait if the buffer is full and consumers wait if the buffer is empty.
- Avoid issues such as race conditions and buffer overflows/underflows.

Pre-requisite:

- Understanding of multithreading or multiprocessing concepts.
- Familiarity with the programming language's threading or process management and semaphore implementation.

Theory:

The producer-consumer problem is a classic synchronization problem in computer science. It involves two processes, the producer and the consumer, who share a common, fixed-size buffer or queue. The producer's job is to produce data items and place them into the buffer, while the consumer's job is to consume these items from the buffer.

The bounded buffer solution introduces a buffer of limited size. The producer must wait if the buffer is full, and the consumer must wait if the buffer is empty. This prevents the producer from producing items when the buffer is full and the consumer from consuming items when the buffer is empty.

Semaphores are used to solve this problem by controlling access to the buffer. Semaphores are integer variables that can be accessed and modified only through two standard atomic operations: wait (P) and signal (V). A semaphore initialized to the buffer size represents the available spaces in the buffer. Another semaphore initialized to zero represents the number of items in the buffer.

One solution of this problem is to use semaphores. The semaphores which will be used here are:

m, a binary semaphore which is used to acquire and release the lock.

empty, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty.

full, a counting semaphore whose initial value is 0.

At any instant, the current value of empty represents the number of empty slots in the buffer and full represents the number of occupied slots in the buffer.

The Producer Operation

The pseudocode of the producer function looks like this:

```
do
{
   // wait until empty > 0 and then decrement 'empty'
   wait(empty);
   // acquire lock
   wait(mutex);

   /* perform the insert operation in a slot */

   // release lock
   signal(mutex);
   // increment 'full'
   signal(full);
}
while(TRUE)
```

- Looking at the above code for a producer, we can see that a producer first waits until there is at least one empty slot.
- Then it decrements the empty semaphore because there will now be one less empty slot since the producer is going to insert data in one of those slots.
- Then, it acquires a lock on the buffer, so that the consumer cannot access the buffer until the producer completes its operation.
- After performing the insert operation, the lock is released and the value of full is incremented because the producer has just filled a slot in the buffer.

The Consumer Operation

The pseudocode for the consumer function looks like this:

```
do
{
   // wait until full > 0 and then decrement 'full'
   wait(full);
   // acquire the lock
   wait(mutex);

   /* perform the remove operation in a slot */

   // release the lock
   signal(mutex);
   // increment 'empty'
   signal(empty);
}
while(TRUE);
```

- The consumer waits until there is at least one full slot in the buffer.

- Then it decrements the full semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.
- After that, the consumer acquires lock on the buffer.
- Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.
- Then, the consumer releases the lock.
- Finally, the empty semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

Conclusion:

	MAHARASHTRA INSTITUTE OF TECHNOLOGY AURANGABAD	LABORATORY MANUAL
QUEST FOR EXCELLENCE	PRACTICAL EXPERIMENT INSTRUCTION SHEET	
	DEPARTMENT: EMERGING SCIENCE & TECHNOLOGY	

LABORATORY : MIT-EST-OCC		A.Y.: 2023-24
Class: TY	PART: II	SUBJECT: AID373 Operating System

Experiment No: 6

AIM : Write a program illustrating various file handling functions.

Objective:

- To demonstrate the use of file handling functions in reading from and writing to files.

Outcomes:

- Understanding of how to open, read from, write to, and close files in a programming language.
- Ability to manipulate file content, such as counting characters, words, and lines, or performing other operations.

Pre-requisite:

- Basic knowledge of a programming language that supports file handling (e.g., C, C++, Python).
- Understanding of file operations such as opening, reading, writing, and closing files.

THEROY:

File handing in C is the process in which we create, open, read, write, and close operations on a file. C language provides different functions such as fopen(), fwrite(), fread(),fprintf(), etc. to perform input, output, and many different C file operations in our program.

In order to understand why file handling is important, let us look at a few features of using files:

- **Reusability:** The data stored in the file can be accessed, updated, and deleted anywhere and anytime providing high reusability.
- **Portability:** Without losing any data, files can be transferred to another in the computer system. The risk of flawed coding is minimized with this feature.
- **Efficient:** A large amount of input may be required for some programs. File handling allows you to easily access a part of a file using few instructions which saves a lot of time and reduces the chance of errors.

PREPARED BY: Ms. Priyanka Sonwane, Ms. Suchita Waghmare (Subject Teacher)	**APPROVED BY** : Kavita V. Bhosle HESTD

- **Storage Capacity:** Files allow you to store a large amount of data without having to worry about storing everything simultaneously in a program.

Various File Handling Operations:

- fopen() - To open a file function is used to open a file to perform operations such as reading, writing etc. In a C program, we declare a file pointer and use fopen() as below. fopen() function creates a new file if the mentioned file name does not exist.
- fclose() - To close a fclose() function closes the file that is being pointed by file pointer fp. In a C program.
- fprintf() – To write into a file function writes string into a file pointed by fp.In a c program we write string into a file.
- fgets() – To read a file function is used to read a file line by line in a c program.

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

int main()
{
   FILE *input_file, *output_file;
   char input_filename[] = "input.txt";
   char output_filename[] = "output.txt";
   char ch;
   int char_count = 0, word_count = 0, line_count = 0, in_word = 0;

   // Open the input file
   input_file = fopen(input_filename, "r");
   if (input_file == NULL)
   {
      printf("Error opening file %s.\n", input_filename);
      return 1;
   }

   // Open the output file
   output_file = fopen(output_filename, "w");
   if (output_file == NULL)
```

```
    {
        printf("Error opening file %s.\n", output_filename);
        return 1;
    }

    // Read input file character by character
    while ((ch = fgetc(input_file)) != EOF)
    {
        char_count++;

        // Check for newline character
        if (ch == '\n')
            line_count++;

        // Check for word
        if (ch == ' ' || ch == '\t' || ch == '\n')
            in_word = 0; // Not in a word
        else if (!in_word)
        {
            in_word = 1; // Start of a new word
            word_count++;
        }

        // Write character to output file
        fputc(ch, output_file);
    }

    // Close the input file
    fclose(input_file);

    // Close the output file
    fclose(output_file);

    // Output the results
    printf("Character count: %d\n", char_count);
    printf("Word count: %d\n", word_count);
    printf("Line count: %d\n", line_count);

    return 0;
}
```

QUEST FOR EXCELLENCE	**MAHARASHTRA INSTITUTE OF TECHNOLOGY AURANGABAD**	**LABORATORY MANUAL**
	PRACTICAL EXPERIMENT INSTRUCTION SHEET	
	DEPARTMENT: EMERGING SCIENCE & TECHNOLOGY	

LABORATORY : MIT-EST-OCC		A.Y.: 2023-24
Class: TY	PART: II	SUBJECT: AID373 Operating System

O/P

Conclusion

PREPARED BY: Ms. Priyanka Sonwane, Ms. Suchita Waghmare (Subject Teacher)	**APPROVED BY** : Kavita V. Bhosle HESTD

Experiment No: 7

AIM: Write a program for copying the content of one file to another.

Objective:

- To copy the content of one file to another, either as a simple file duplication or for processing the content in some way during the copy.

Outcomes:

- Successfully duplicate the content of the source file into the destination file.
- Optionally, perform any required processing on the content during the copy.

Pre-requisite:

- Basic understanding of file handling in the chosen programming language.
- Knowledge of how to open, read from, write to, and close files.

THEROY:

The program takes the name of the source and target files as input from the user, reads the content of the source file, and copies it into the target file. At the end, it prints a success message on the output window.

To copy the content of one file into another in C programming language, you mainly need three built-in functions, fopen(), fgetc() and fputc().

fopen() function is used to open a file in a specified mode such as read(“r”), write(“w”), etc.

fgetc() function is used to read each character of a file one by one.

fputc()function on the other hand is to write content into a file.

Steps:

- fopen() needs two parameters first, the name or path of the file which you want to open and second, the mode in which you want to open the file which can be “r”, “w”, etc.
- If the file successfully opens, it returns a FILE pointer which points to the first character of the file. But, If some error occurs, or the file does not exist.

- fopen() function returns a NULL pointer. So, to copy the content of one file into another all you have to do is, open the source file in read(“r”) mode and the target file in write(“w”) mode.
- fopen() function Once both files are successfully opened, start reading each character of the source file using the fgetc() function and write into the target file using the fputc() function.

```
#include <stdio.h>
#include <stdlib.h> // For exit()

int main()
{
        FILE *fptr1, *fptr2;
        char filename[100], c;

        printf("Enter the filename to open for reading \n");
        scanf("%s", filename);

        // Open one file for reading
        fptr1 = fopen(filename, "r");
        if (fptr1 == NULL)
        {
                printf("Cannot open file %s \n", filename);
                exit(0);
        }

        printf("Enter the filename to open for writing \n");
        scanf("%s", filename);

        // Open another file for writing
        fptr2 = fopen(filename, "w");
        if (fptr2 == NULL)
        {
                printf("Cannot open file %s \n", filename);
                exit(0);
        }
        // Read contents from file
        c = fgetc(fptr1);
        while (c != EOF)
        {
                fputc(c, fptr2);
                c = fgetc(fptr1);
        }

        printf("\nContents copied to %s", filename);

        fclose(fptr1);
        fclose(fptr2);
        return 0;
}
```

Sample Input:

Enter the Name of the source file : source.txt.
Enter the Name of the Target file : target.txt.

Sample Output:

Contents of source.txt copied into target.txt successfully.

Conclusion:

Experiment No: 8

Aim: Implementation of various memory allocation algorithms, (First fit, best fit and Worst fit).

Objective:

- To allocate memory blocks to processes efficiently using different algorithms: First Fit, Best Fit, and Worst Fit.

Outcomes:

- Understand how each algorithm works and its impact on memory fragmentation.
- Implement the algorithms to allocate memory blocks based on process requirements.

Pre-requisite:

- Understanding of memory management in operating systems.
- Familiarity with programming concepts such as arrays, loops, and functions.

Theory:

First Fit

The First Fit algorithm allocates the first available block of memory that is large enough to satisfy a request. It searches through the memory blocks from the beginning until it finds a suitable block.

Implementation:

- Initialize a list of free memory blocks.
- When a memory allocation request is made:
    - Iterate through the list of free memory blocks.
    - Allocate memory from the first block that is large enough to satisfy the request.
    - If no block is large enough, allocate a new block from the system.

Best Fit The Best Fit algorithm allocates the smallest available block of memory that is large enough to satisfy a request. It searches through all the free memory blocks to find the block that fits the request most closely.

Implementation:

- Initialize a list of free memory blocks.
- When a memory allocation request is made:
    - Iterate through the list of free memory blocks to find the smallest block that is large enough to satisfy the request.
    - Allocate memory from this block.
    - If no block is large enough, allocate a new block from the system.

Worst Fit

The Worst Fit algorithm allocates the largest available block of memory. It searches through all the free memory blocks to find the largest block, which is then used to satisfy the request.

Implementation:

- Initialize a list of free memory blocks.
- When a memory allocation request is made:
    - Iterate through the list of free memory blocks to find the largest block.
    - Allocate memory from this block.
    - If no block is large enough, allocate a new block from the system.

Program:

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

#define MAX_BLOCKS 100

// Structure to represent a memory block
typedef struct {
   int id;
   int size;
   int allocated;
} MemoryBlock;
```

```
// Function prototypes
void firstFit(MemoryBlock blocks[], int numBlocks, int requestSize);
void bestFit(MemoryBlock blocks[], int numBlocks, int requestSize);
void worstFit(MemoryBlock blocks[], int numBlocks, int requestSize);
void printBlocks(MemoryBlock blocks[], int numBlocks, const char* allocationAlgorithm);

int main() {
   MemoryBlock blocks[MAX_BLOCKS];
   int numBlocks, i, requestSize;

   printf("Enter the number of memory blocks: ");
   scanf("%d", &numBlocks);

   // Input memory blocks
   for (i = 0; i < numBlocks; i++) {
      printf("Enter size of block %d: ", i + 1);
      scanf("%d", &blocks[i].size);
      blocks[i].id = i + 1;
      blocks[i].allocated = 0;
   }

   // Print initial state of memory blocks
   printf("\nInitial state of memory blocks:\n");
   printBlocks(blocks, numBlocks, "None");

   // Process memory allocation requests
   printf("\nEnter size of memory request: ");
   scanf("%d", &requestSize);

   printf("\nFirst Fit Allocation:\n");
   firstFit(blocks, numBlocks, requestSize);
   printBlocks(blocks, numBlocks, "First Fit");

   printf("\nBest Fit Allocation:\n");
   bestFit(blocks, numBlocks, requestSize);
   printBlocks(blocks, numBlocks, "Best Fit");

   printf("\nWorst Fit Allocation:\n");
   worstFit(blocks, numBlocks, requestSize);
```

```
    printBlocks(blocks, numBlocks, "Worst Fit");

    return 0;
}

// First Fit Allocation Algorithm
void firstFit(MemoryBlock blocks[], int numBlocks, int requestSize) {
    int i;
    for (i = 0; i < numBlocks; i++) {
        if (blocks[i].allocated == 0 && blocks[i].size >= requestSize) {
            blocks[i].allocated = 1;
            break;
        }
    }
}

// Best Fit Allocation Algorithm
void bestFit(MemoryBlock blocks[], int numBlocks, int requestSize) {
    int i, bestFitIndex = -1, minFragmentation = INT_MAX;
    for (i = 0; i < numBlocks; i++) {
        if (blocks[i].allocated == 0 && blocks[i].size >= requestSize) {
            int fragmentation = blocks[i].size - requestSize;
            if (fragmentation < minFragmentation) {
                minFragmentation = fragmentation;
                bestFitIndex = i;
            }
        }
    }
    if (bestFitIndex != -1)
        blocks[bestFitIndex].allocated = 1;
}

// Worst Fit Allocation Algorithm
void worstFit(MemoryBlock blocks[], int numBlocks, int requestSize) {
    int i, worstFitIndex = -1, maxFragmentation = -1;
    for (i = 0; i < numBlocks; i++) {
        if (blocks[i].allocated == 0 && blocks[i].size >= requestSize) {
            int fragmentation = blocks[i].size - requestSize;
            if (fragmentation > maxFragmentation) {
                maxFragmentation = fragmentation;
            worstFitIndex = i;
         }
      }
   }

if (worstFitIndex != -1)
      blocks[worstFitIndex].allocated = 1;
}

// Function to print memory blocks
void printBlocks(MemoryBlock blocks[], int numBlocks, const char* allocationAlgorithm) {
   int i;
   printf("Block\tSize\tAllocated (%s)\n", allocationAlgorithm);
   for (i = 0; i < numBlocks; i++) {
      printf("%d\t%d\t%s\n", blocks[i].id, blocks[i].size, blocks[i].allocated ? "Yes" : "No");
   }
}
```

OUTPUT:

```
Enter the number of memory blocks: 5
Enter size of block 1: 20
Enter size of block 2: 15
Enter size of block 3: 25
Enter size of block 4: 30
Enter size of block 5: 40

Initial state of memory blocks:
Block   Size    Allocated (None)
1       20      No
2       15      No
3       25      No
4       30      No
5       40      No

Enter size of memory request: 18

First Fit Allocation:
```

```
Block  Size  Allocated (First Fit)
1      20    Yes
2      15    No
3      25    No
4      30    No
5      40    No

Best Fit Allocation:
Block  Size  Allocated (Best Fit)
1      20    Yes
2      15    No
3      25    Yes
4      30    No
5      40    No



Worst Fit Allocation:
Block  Size  Allocated (Worst Fit)
1      20    Yes
2      15    No
3      25    Yes
4      30    No
5      40    Yes
```

Conclusion:

QUEST FOR EXCELLENCE	**MAHARASHTRA INSTITUTE OF TECHNOLOGY AURANGABAD**	**LABORATORY MANUAL**
	PRACTICAL EXPERIMENT INSTRUCTION SHEET	
	DEPARTMENT: EMERGING SCIENCE & TECHNOLOGY	

LABORATORY : MIT-EST-OCC	A.Y.: 2023-24

Class: TY	PART: II	SUBJECT: AID373 Operating System

Experiment No: 9

Aim: Implementation of FIFO page replacement algorithms.

Objective: The objective of the FIFO (First-In-First-Out) page replacement algorithm is to manage memory effectively by replacing the oldest page in memory with the new page that needs to be loaded.

Outcome: The outcome of using the FIFO algorithm is that it can lead to high page fault rates, especially in situations where the oldest pages in memory are frequently accessed or are part of a working set.

Prerequisites:

Understanding of virtual memory and paging.
Knowledge of page faults.
Basic understanding of data structures, specifically queues.

Theory:

The FIFO (First-In-First-Out) page replacement algorithm is one of the simplest page replacement algorithms used in operating systems. It works on the principle of replacing the oldest page in memory (the page that has been in memory the longest) with the new page that needs to be loaded. Here's a summary of how the FIFO algorithm works:

1. **Page Table**: Each process has a page table that maps logical addresses to physical addresses. The page table keeps track of which pages are currently in memory and which ones are on disk.
2. **Page Fault:** When a process accesses a page that is not currently in memory (a page fault occurs), the operating system must decide which page to remove from memory to make space for the new page.
3. **Replacement:** In the FIFO algorithm, the operating system maintains a queue of pages in memory, with the oldest page at the front of the queue and the newest page at the end. When a page fault occurs, the operating system removes the page at the front of the queue (the oldest page) and loads the new page into memory.

PREPARED BY: Ms. Priyanka Sonwane, Ms. Suchita Waghmare (Subject Teacher)	**APPROVED BY** : Kavita V. Bhosle HESTD

4. **Implementation:** The FIFO algorithm is easy to implement using a queue data structure. Each time a page is accessed, it is added to the end of the queue. When a page needs to be replaced, the page at the front of the queue is removed.
5. **Advantages:** The main advantage of the FIFO algorithm is its simplicity. It is easy to implement and requires minimal bookkeeping.
6. **Disadvantages:** However, FIFO can suffer from the "Belady's anomaly," where increasing the number of page frames can actually increase the number of page faults. This is because FIFO does not consider the frequency of page access or the relevance of pages to the current workload.
7. **Example:** Consider a scenario where a process accesses pages in the order 1, 2, 3, 4, 1, 2, 5. If the system has only 3 page frames, the page faults would occur for pages 4, 1, and 5. The page frame queue would look like this:
   - Initially: []
   - After accessing page 1: [1]
   - After accessing page 2: [1, 2]
   - After accessing page 3: [1, 2, 3]
   - After accessing page 4: [2, 3, 4]
   - After accessing page 1: [3, 4, 1]
   - After accessing page 2: [4, 1, 2]
   - After accessing page 5: [1, 2, 5]

Program:

```
#include <stdio.h>
#include <stdbool.h>
// Function to check if a page exists in a frame
bool isInFrame(int page, int frames[], int frameSize) {
    for (int i = 0; i < frameSize; i++) {
        if (frames[i] == page)
            return true;
    }
    return false;
}
// Function to find the index of the page that entered first (FIFO)
int findFIFOIndex(int pages[], int frames[], int frameSize, int currentPageIndex) {
    int oldestPageIndex = currentPageIndex;
    for (int i = 0; i < frameSize; i++) {
        int j;
        for (j = currentPageIndex - 1; j >= 0; j--) {
            if (frames[i] == pages[j]) {
                if (j < oldestPageIndex) {
                    oldestPageIndex = j;
                }
                break;
            }
        }
        if (j == -1)
            return i;
    }
    return oldestPageIndex;
}

// Function to display the frames
void displayFrames(int frames[], int frameSize) {
    for (int i = 0; i < frameSize; i++) {
        if (frames[i] == -1)
            printf("X ");
        else
            printf("%d ", frames[i]);
    }
    printf("\n");
}

// FIFO Page Replacement Algorithm
void fifo(int pages[], int numPages, int frameSize) {
    int frames[frameSize];
    int pageFaults = 0;
    int frameIndex = 0;
    for (int i = 0; i < frameSize; i++)
        frames[i] = -1;

    for (int i = 0; i < numPages; i++) {
        if (!isInFrame(pages[i], frames, frameSize)) {
            pageFaults++;
            frames[frameIndex] = pages[i];
            frameIndex = (frameIndex + 1) % frameSize;
        }
        printf("Page %d: ", pages[i]);
        displayFrames(frames, frameSize);
    }
    printf("\nTotal Page Faults: %d\n", pageFaults);
}
int main() {
    int numPages, frameSize;
    printf("Enter the number of pages: ");
    scanf("%d", &numPages);
    int pages[numPages];
    printf("Enter the page references: ");
    for (int i = 0; i < numPages; i++) {
        scanf("%d", &pages[i]);
    }
    printf("Enter the frame size: ");
    scanf("%d", &frameSize);
    fifo(pages, numPages, frameSize);
    return 0;
}
```

OUTPUT:

```
Enter the number of pages: 7
Enter the page references:
1
3
0
3
5
6
3
Enter the frame size: 3
Page 1: 1 X X
Page 3: 1 3 X
Page 0: 1 3 0
Page 3: 1 3 0
Page 5: 5 3 0
Page 6: 5 6 0
Page 3: 5 6 3

Total Page Faults: 6
```

Conclusion:

QUEST FOR EXCELLENCE	**MAHARASHTRA INSTITUTE OF TECHNOLOGY AURANGABAD**	**LABORATORY MANUAL**
	PRACTICAL EXPERIMENT INSTRUCTION SHEET	
	DEPARTMENT: EMERGING SCIENCE & TECHNOLOGY	

LABORATORY : MIT-EST-OCC	A.Y.: 2023-24

Class: TY	PART: II	SUBJECT: AID373 Operating System

Experiment No: 10

Aim: Implementation of FCFS Disk Scheduling algorithm.

Objective: The objective of implementing the FCFS (First-Come, First-Served) Disk Scheduling algorithm is to manage the ordering of disk I/O requests based on their arrival times, ensuring that they are processed in the order they are received.

Outcome: The outcome of using the FCFS algorithm is that it can lead to high average response times, especially if there are long seek times between consecutive disk requests.

Prerequisites:

- Understanding of disk I/O operations: Familiarity with how data is read from and written to a disk, including concepts such as seek time and rotational latency.
- Knowledge of disk scheduling algorithms: Understanding of different disk scheduling algorithms, their characteristics, and how they impact disk performance.
- Basic programming skills: Ability to implement algorithms in a programming language, as the FCFS algorithm needs to be coded to handle disk I/O requests.

Theory :

The First-Come, First-Served (FCFS) Disk Scheduling algorithm is a simple disk scheduling algorithm that processes requests in the order they arrive in the disk queue. Here's a summary of the theory behind the FCFS algorithm:

Disk Queue: As I/O requests arrive from different processes, they are added to the disk queue. Each request includes the location on the disk (track number) that needs to be accessed.

Head Movement: The disk has a read/write head that moves across the disk surface to access data. The head movement is a major factor in the efficiency of disk scheduling algorithms.

Processing Order: In the FCFS algorithm, the disk processes requests in the order they arrive. The head moves sequentially from one request to the next, regardless of the location of the requests on the disk.

PREPARED BY: Ms. Priyanka Sonwane, Ms. Suchita Waghmare (Subject Teacher)	**APPROVED BY** : Kavita V. Bhosle HESTD

Seek Time: The time taken by the disk arm to move the read/write head to the track containing the requested sector is called the seek time. FCFS scheduling does not attempt to minimize seek time.

Rotational Latency: After the head reaches the correct track, it may need to wait for the desired sector to rotate under the head. This waiting time is called rotational latency and is not considered in FCFS scheduling.

Efficiency: FCFS is simple to implement but may not be efficient in terms of reducing access times. It can lead to a phenomenon known as the "head-of-the-line" (HOL) blocking, where a request for a distant track can prevent closer requests from being serviced quickly.

Example: Consider a disk queue with requests arriving in the order A, B, C, D, E, where each request specifies the track number to be accessed. If the head is initially at track 50 and the tracks are numbered from 0 to 199, the sequence of head movements for FCFS would be: 50 → A → B → C → D → E.

Program:

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

// Function to calculate the total head movement
int calculateTotalHeadMovement(int *requests, int numRequests, int initialPosition) {
   int totalHeadMovement = 0;

   // Iterate through the requests and calculate head movement
   for (int i = 0; i < numRequests; i++) {
      // Calculate absolute difference between current request and previous request
      int movement = abs(requests[i] - initialPosition);
      // Add it to total head movement
      totalHeadMovement += movement;
      // Update initial position
      initialPosition = requests[i];
   }

   return totalHeadMovement;
}
```

```
int main() {
   int numRequests;
   printf("Enter the number of disk requests: ");
   scanf("%d", &numRequests);

   int requests[numRequests];
   printf("Enter the disk requests: ");
   for (int i = 0; i < numRequests; i++) {
      scanf("%d", &requests[i]);
   }

   int initialPosition;
   printf("Enter the initial head position: ");
   scanf("%d", &initialPosition);

   // Calculate total head movement using FCFS algorithm
   int totalHeadMovement = calculateTotalHeadMovement(requests, numRequests,
initialPosition);
   printf("Total head movement: %d\n", totalHeadMovement);

   return 0;
}
```

OUTPUT:

```
Enter the number of disk requests: 8
Enter the disk requests:
176
79
34
60
92
11
41
114
Enter the initial head position: 50
Total head movement: 510
```

Conclusion:

	MAHARASHTRA INSTITUTE OF TECHNOLOGY AURANGABAD	LABORATORY MANUAL
QUEST FOR EXCELLENCE	PRACTICAL EXPERIMENT INSTRUCTION SHEET	
	DEPARTMENT: EMERGING SCIENCE & TECHNOLOGY	

LABORATORY : MIT-EST-OCC		A.Y.: 2023-24
Class: TY	PART: II	SUBJECT: AID373 Operating System

Experiment No: 11

Aim: Case study: Red Hat Linux OS

Objective: Red Hat Linux aims to provide a stable and secure operating system for use in enterprise environments, offering support for critical business applications and services.

Outcome: Red Hat Linux has been successful in establishing itself as a leading operating system for enterprise servers. It is known for its reliability, scalability, and long-term support, with updates and security patches provided through Red Hat's subscription-based model.

Theory:

Conclusion:

PREPARED BY: Ms. Priyanka Sonwane, Ms. Suchita Waghmare (Subject Teacher)	**APPROVED BY** : Kavita V. Bhosle HESTD