**INDEX**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| S.No. | Name of the Experiment | Date | Page No. | Marks  Awarded | Remarks |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

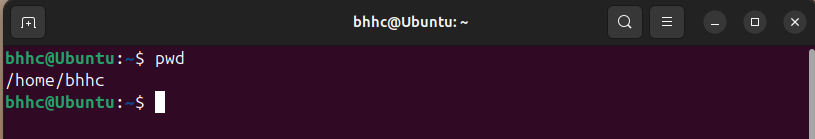
**OS RECORD**

**WEEK-1:**

**AIM:** Understanding and practical exposure towards Basic Linux commands.

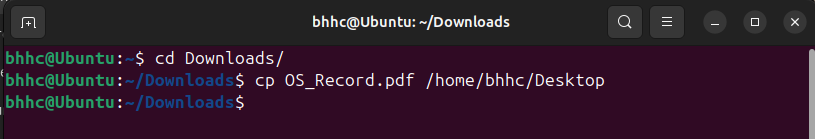
**1. pwd**

* **Definition**: Displays the full, absolute path of the current working directory, starting from the root (/).
* **Syntax**: pwd
* **Command**: pwd
* **Output**:



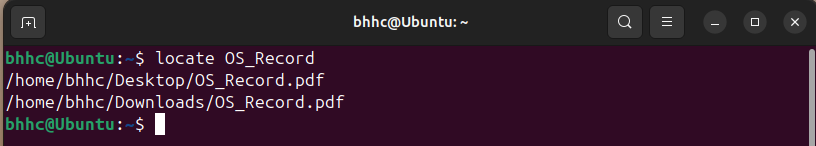
**2. cp**

* **Definition**: Copies files or directories from one location to another. Can also copy multiple files to a directory.
* **Syntax**: cp [options] source destination
* **Command**: cp OS\_Record.pdf /home/Desktop
* **Output**:



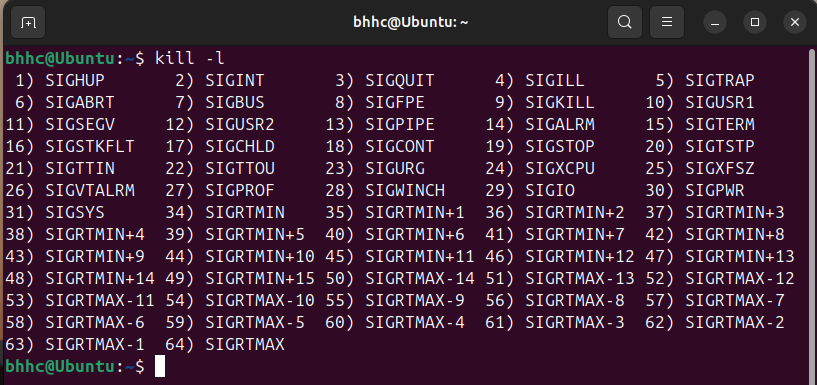
**3. locate**

* **Definition**: Searches for files and directories by name using an indexed database, making it faster than other search commands.
* **Syntax**: locate [pattern]
* **Command**: locate OS\_Record
* **Output**:



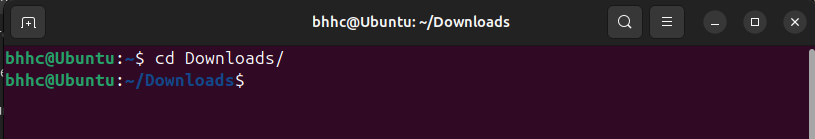
**4. kill**

* **Definition**: Terminates a process by sending it a signal, typically used to stop unresponsive programs.
* **Syntax**: kill [signal] PID
* **Command**: kill -l
* **Output**:



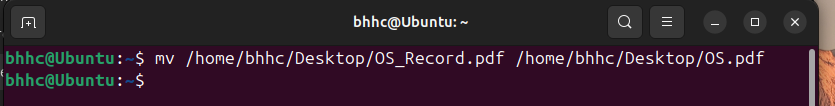
**5. cd**

* **Definition**: Changes the current working directory to the specified directory. Can navigate using relative or absolute paths.
* **Syntax**: cd [directory]
* **Command**: cd Downloads/
* **Output**:



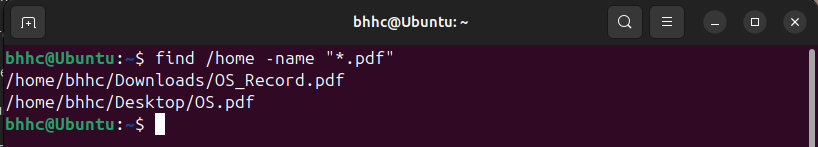
**6. mv**

* **Definition**: Moves or renames files and directories. Can transfer files between directories or update their names.
* **Syntax**: mv [source] [destination]
* **Command**: mv /home/bhhc/Desktop/OS\_Record.pdf /home/bhhc/Desktop/OS.pdf
* **Output**:



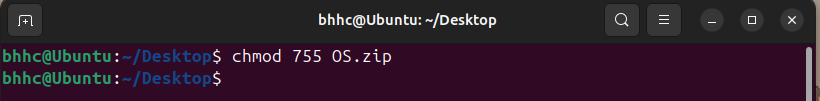
**7. find**

* **Definition**: Searches for files and directories based on conditions like name, size, or permissions, and performs actions on them if specified.
* **Syntax**: find [path] [options] [expression]
* **Command**: find /home -name "\*.pdf"
* **Output**:



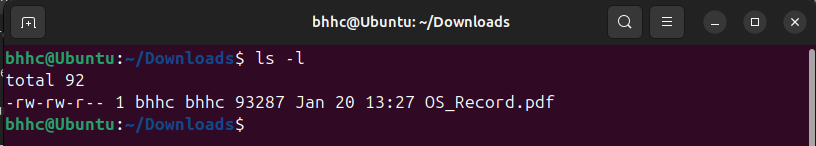
**8. chmod**

* **Definition**: Modifies the read, write, and execute permissions of a file or directory for the user, group, and others.
* **Syntax**: chmod [permissions] [file]
* **Command**: chmod 755 script.sh
* **Output**:



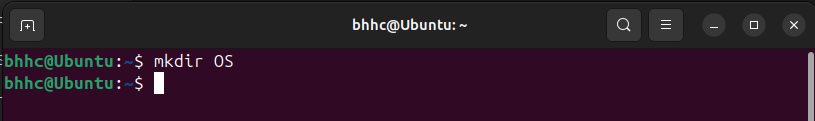
**9. ls**

* **Definition**: Lists the files and directories in the current or specified directory, with options to show hidden files or detailed metadata.
* **Syntax**: ls [options] [path]
* **Command**: ls -l
* **Output**:



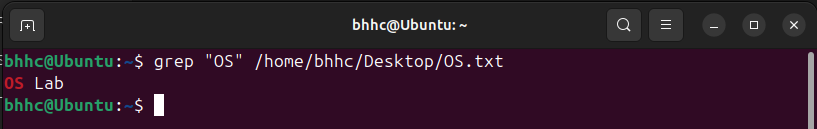
**10. mkdir**

* **Definition**: Creates a new directory. Can also create parent directories if they do not exist.
* **Syntax**: mkdir [options] directory
* **Command**: mkdir OS
* **Output**:



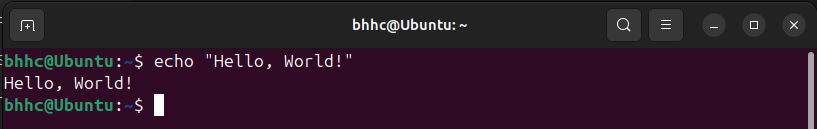
**11. grep**

* **Definition**: Searches for a specific text pattern in files or output streams and highlights matching lines.
* **Syntax**: grep [options] pattern [file]
* **Command**: grep "OS" mv /home/bhhc/Desktop/OS.pdf
* **Output**:



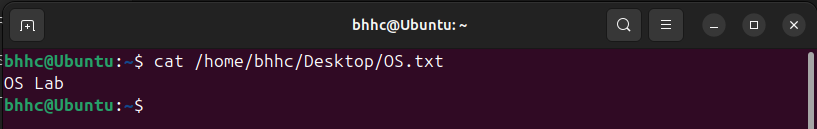
**12. echo**

* **Definition**: Displays a string or variable value to the terminal. Commonly used in scripts.
* **Syntax**: echo [string]
* **Command**: echo "Hello, World!"
* **Output**



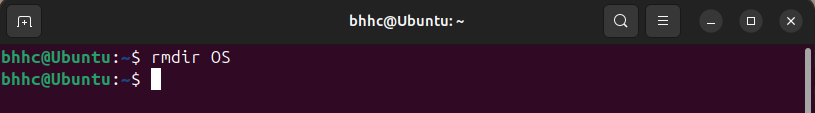
**13. cat**

* **Definition**: Displays the content of a file, combines multiple files, or creates new files.
* **Syntax**: cat [file]
* **Command**: cat /home/bhhc/Desktop/OS.pdf
* **Output**:



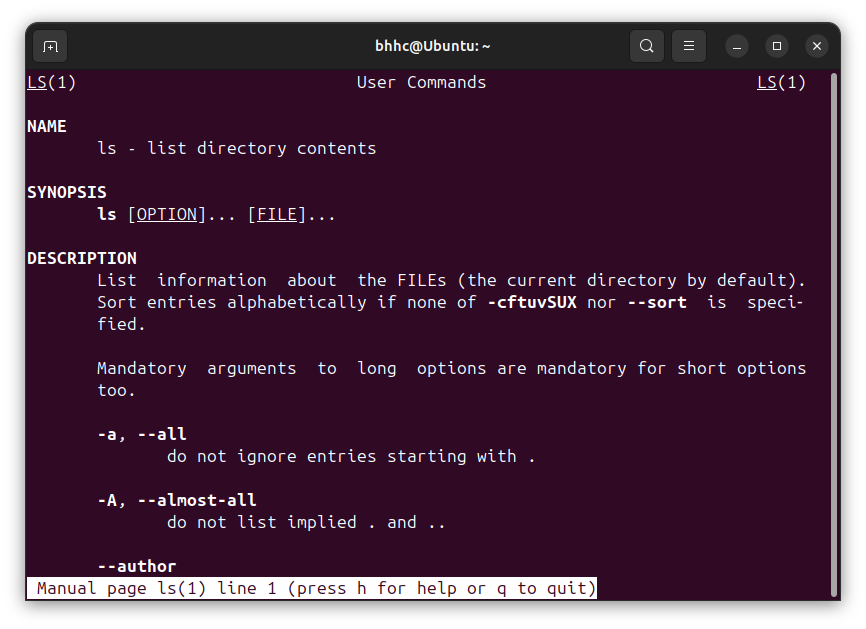
**14. rmdir**

* **Definition**: Deletes empty directories. Will not work if the directory contains files or subdirectories.
* **Syntax**: rmdir [directory]
* **Command**: rmdir OS
* **Output**:



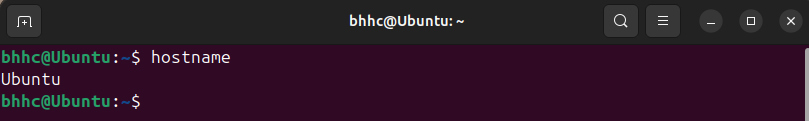
**15. man**

* **Definition**: Displays the manual page for a command, detailing its purpose, options, and examples.
* **Syntax**: man [command]
* **Command**: man ls
* **Output**:



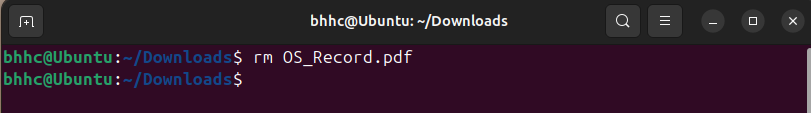
**16. hostname**

* **Definition**: Displays or sets the hostname of the system, used for network identification.
* **Syntax**: hostname
* **Command**: hostname
* **Output**:



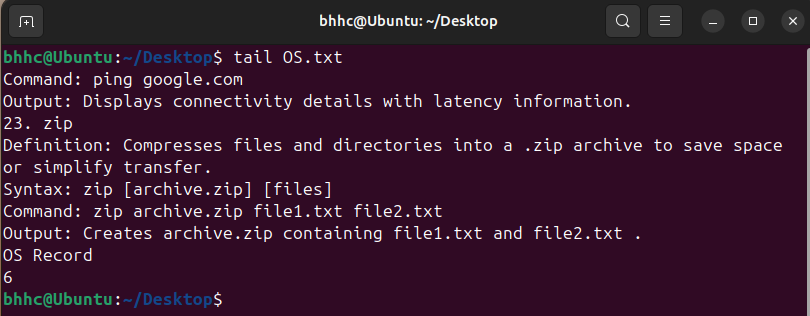
**17. rm**

* **Definition**: Deletes files and directories. With options, can recursively remove directories and their contents.
* **Syntax**: rm [options] file
* **Command**: rm OS\_Record.txt
* **Output**:



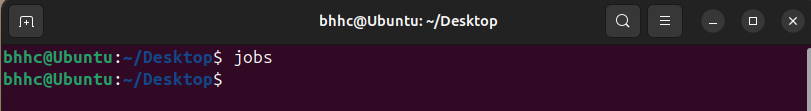
**18. tail**

* **Definition**: Displays the last few lines of a file, commonly used to monitor logs.
* **Syntax**: tail [options] [file]
* **Command**: tail OS.txt
* **Output**:



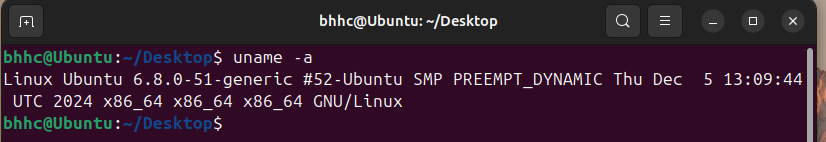
**19. jobs**

* **Definition**: Lists all active or suspended background jobs in the current shell session.
* **Syntax**: jobs
* **Command**: jobs
* **Output**:



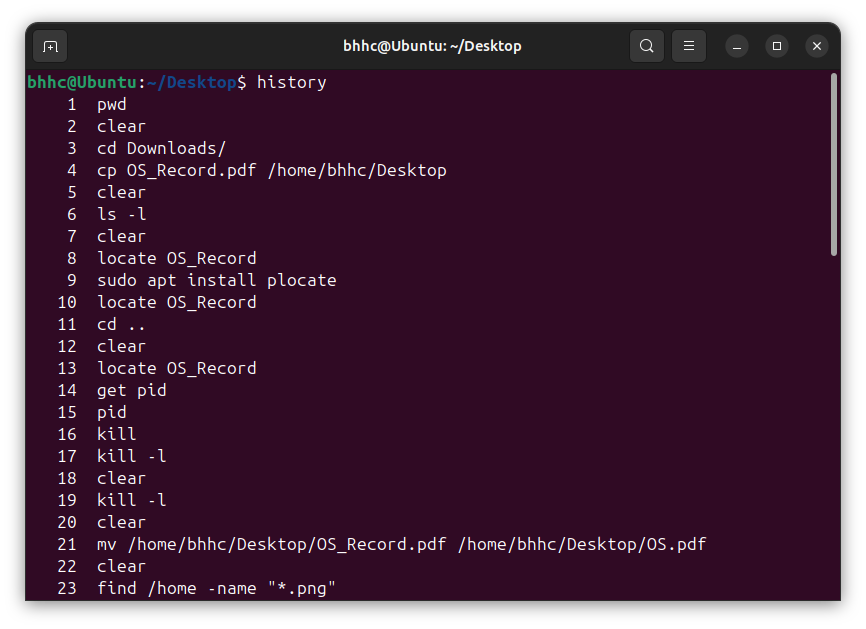
**20. uname**

* **Definition**: Provides basic information about the system, such as the kernel name and version.
* **Syntax**: uname [options]
* **Command**: uname -a
* **Output**:



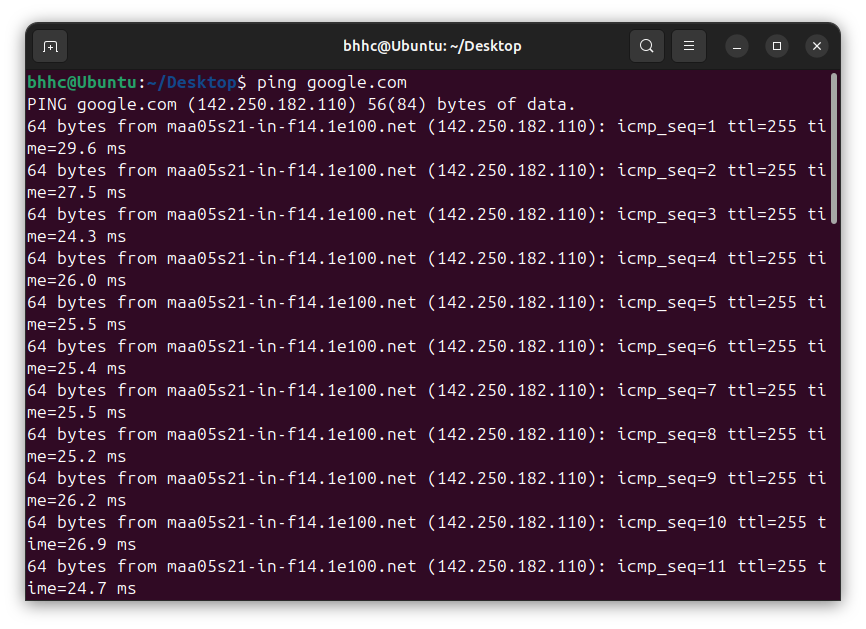
**21. history**

* **Definition**: Displays a list of previously executed commands in the terminal session.
* **Syntax**: history
* **Command**: history
* **Output**:

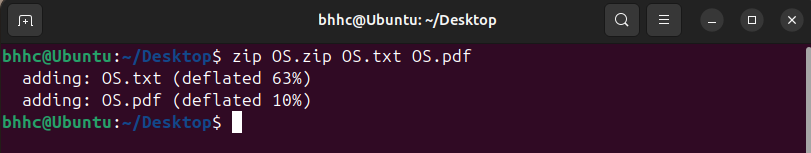


**22. ping**

* **Definition**: Tests network connectivity by sending packets to a specified host and measuring the response time.
* **Syntax**: ping [host]
* **Command**: ping google.com
* **Output**:



**23. zip**

* **Definition**: Compresses files and directories into a .zip archive to save space or simplify transfer.
* **Syntax**: zip [archive.zip] [files]
* **Command**: zip OS.zip OS.txt OS.pdf
* **Output**:

**WEEK-2:**

**AIM:** Collect the basic information about your machine using proc in Linux.

**Introduction to Proc File System (/proc)**

The **proc file system (procfs)** is a virtual file system created dynamically when the system boots and is removed upon shutdown. It serves as a control and information center for the kernel, containing real-time data about system processes. Additionally, it facilitates communication between kernel space and user space.

**1. Listing Root Directories**

To list all directories under the root (/), use the following command:

ls /

**Output**:

**2. Listing Directories Under /proc**

The /proc directory contains various subdirectories, each corresponding to a running process. To view them, use:

ls /proc

**Output**:

**3. Viewing Active Processes Using the “top” Command**

The top command provides a dynamic real-time view of system processes. It displays CPU usage, memory consumption, and process details.

Top

**Output**:

**4. Terminating Processes Using the Kill Command**

The kill command is used to terminate processes by their Process ID (PID).

kill <PID>

Alternatively, to forcefully kill a process:

kill -9 <PID>

**Output**:

**5. Using the Cat Command**

The cat command displays the contents of a file. It is useful for reading system and process information:

cat /proc/cpuinfo

cat /proc/meminfo

**Output**:

**6. Retrieving CPU Information**

To obtain basic details about the CPU, use:

cat /proc/cpuinfo

**Output**:

This provides information such as processor type, number of cores, and clock speed.

**7. Retrieving Kernel Information**

To view the kernel command line arguments:

cat /proc/cmdline

**Output**:

**8. Retrieving Memory Information**

To check memory details:

cat /proc/meminfo

**Output**:

This displays total available memory, used memory, and free memory.

**WEEK-3:**

**AIM:** Implementation of write () and read () system calls.

**System Calls:** A system call provides an interface to services provided by the operating system (OS).

**Services of OS**

1. User Interface
2. Program Execution
3. I/O Operations
4. File System Manipulations
5. Communications
6. Error Detection
7. Resource Allocation
8. Accounting
9. Protection & Security

**Kernel Mode vs User Mode**

* **Kernel Mode:** A program can access all the resources directly. However, there is no backup for kernel mode execution. If one process fails, the entire system crashes.
* **User Mode:** The safest mode for program execution, but it cannot directly access resources. It sends system calls to the kernel mode to request access.
* **Context Switching:** The process of switching between user mode and kernel mode. The calls made by the OS for context switching are known as **System Calls**.

**Write System Call**

**Syntax:**

#include <unistd.h>

ssize\_t write(int fd, const void \*buf, size\_t count);

* **fd** – File Descriptors:
  + 1 – Standard output device
  + 0 – Standard input device
  + 2 – Standard error device
* **Return Type:** ssize\_t – Returns the number of bytes written. If write() fails, it returns -1.

**Program 1: Basic Write System Call**

#include <unistd.h>

int main()

{

    write(1, "hello\n", 6);

    return 0;

}

**Output**:

**Program 2: Write System Call with Byte Count**

#include <stdio.h>

#include <unistd.h>

int main()

{

    int count;

    count = write(1, "hello\n", 6);

    printf("Total bytes written: %d\n", count);

    return 0;

}

**Output**:

**Read System Call**

**Syntax:**

#include <unistd.h>

ssize\_t read(int fd, void \*buf, size\_t count);

**Program 1: Read Data from Standard Input and Write to Screen**

#include <unistd.h>

#include <stdio.h>

int main()

{

    char buff[20];

    printf("\n Enter any text");

    read(0, buff, 10);

    printf("\n Your text is read as");

    write(1, buff, 10);

    return 0;

}

**Output:**

**Program 2: Read Data, Write to Screen, and Count Characters Read**

#include <unistd.h>

#include <stdio.h>

int main()

{

    int nread;

    char buff[20];

    printf("\n Enter any text");

    nread = read(0, buff, 10);

    printf("\n Your text is read as");

    write(1, buff, nread); // Print characters from the buffer on the screen

    printf("\n Number of characters read: %d", nread);

    return 0;

}

**Output:**

**WEEK-4:**

**AIM:** Implementation of open (), fork () system calls

**Open System Call:** The open() system call is used to open a file in multiple modes depending on the requirement.

**Syntax:**

int open(const char \*pathname, int flags);

int open(const char \*pathname, int flags, mode\_t mode);

The open() system call returns an integer file descriptor:

* 0 - Standard input
* 1 - Standard output
* 2 - Standard error

**Flags:**

* O\_RDONLY - Read-only mode
* O\_WRONLY - Write-only mode
* O\_RDWR - Read and write mode

**Read-Only Mode Example**

**Program to Read First 10 Characters from a File**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int main() {

    int n, fd;

    char buff[50];

    fd = open("test.txt", O\_RDONLY);

    printf("The file descriptor of the file is: %d\n", fd);

    n = read(fd, buff, 10);

    write(1, buff, n);

    return 0;

}

**Output**:

**Program to Read from One File and Write to Another**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int main() {

    int n, fd1, fd2;

    char buff[50];

    fd1 = open("test.txt", O\_RDONLY);

    fd2 = open("HELLO.txt", O\_WRONLY);

    printf("The file descriptor of test.txt is: %d\n", fd1);

    printf("The file descriptor of HELLO.txt is: %d\n", fd2);

    n = read(fd1, buff, 20);

    write(fd2, buff, n);

    return 0;

}

**Output**:

**How it Works?**

1. Create a file **test.txt** and write some content into it (more than 10 characters).
2. The open() system call opens test.txt in read-only mode and returns a file descriptor stored in fd.
3. The read() function reads 10 characters from the file into a buffer.
4. The buffer content is displayed on the screen using write().

**Expected Output:**

1. Create the file test.txt and write "1234567890abcdefghij54321" into it.
2. Compile the program open.c.
3. Run the compiled program.

**fork() System Call**

The fork() system call is used to create a new process. The new process is called a **child process**, and the original process is called the **parent process**.

**Syntax:**

#include <unistd.h>

pid\_t fork(void);

* fork() returns -1 on failure.
* On success, it returns 0 in the child process and the **process ID of the child** in the parent process.

**Why Use fork()?**

A process may need to perform **two independent tasks**. Instead of executing them sequentially, the parent process creates a **child process** to handle one task while it handles the other. This reduces execution time.

**Example Program for fork()**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

    pid\_t p;

    printf("Before fork\n");

    p = fork();

    if (p == 0) {

        printf("I am child having id %d\n", getpid());

        printf("My parent's id is %d\n", getppid());

    } else {

        printf("My child's id is %d\n", p);

        printf("I am parent having id %d\n", getpid());

    }

    printf("Common\n");

}

**Output**:

**WEEK-5:**

**AIM:** Implement a program using fork () system call to create a hierarchy of 3 process such that P2 is the child of P1 and P1 is the child of P.

**Program:**

#include <sys/types.h>

#include <unistd.h>

#include <stdio.h>

int main(void) {

    pid\_t pid1, pid2;

    int status;

    pid1 = fork();

    if (pid1 == 0) { // P1 child process

        printf("\n I am the child P1 of parent(P0) and my ID is %d\n", getpid());

        printf("\n My parent is (P0) and whose process ID is %d\n", getppid());

        pid2 = fork();

        if (pid2 == 0) { // P2 child process

            printf("\n I am the P1's child process (P2) and my ID is %d\n", getpid());

            printf("\n My parent is P1 process whose ID is %d\n", getppid());

        } else {

            waitpid(pid2, NULL, 0);

            printf("\n I am P1 my process id is %d\n", getpid());

            printf("\nMy child is P2 whose id is %d\n", pid2);

        }

    } else { // P0 parent process

        waitpid(pid1, NULL, 0);

        printf("\nI am P0 my process id is %d\n", getpid());

        printf("\nMy child is P1 whose id is %d\n", pid1);

    }

    return 0;

}

**Output:**

**WEEK-6:**

* 1. **AIM:** Program to create an Orphan process.

**Program:**

#include <stdio.h>

#include <unistd.h>

#include <sys/types.h>

int main() {

    pid\_t p;

    p = fork();

    if (p == 0) {

        sleep(5);

        printf("I am child having PID %d\n", getpid());

        printf("My parent PID is %d\n", getppid());

    } else {

        printf("I am parent having PID %d\n", getpid());

        printf("My child PID is %d\n", p);

    }

    return 0;

}

**Output:**

* 1. **AIM:** Create two child process C1 and C2. Make sure that only C2 becomes an Orphan process.

**Program:**

#include <stdio.h>

#include<unistd.h>

#include<sys/types.h>

int main() {

    pid\_t c,c1;

    printf("before fork of c:\n");

    c=fork();

    if(c==0)

    {

        printf("I am c1 having id:%d\n",getpid());

        printf("my parent c id is:%d\n",getppid());

        printf("before fork of c1:\n");

        c1=fork();

        if(c1==0)

        {

            sleep(2);

            printf("after termination\n");

            printf("i am c2 having id:%d\n",getpid());

            printf("my parent c1 id is:%d\n",getppid());

        }

        else

        {

            printf("I am c1 having id:%d\n",getpid());

        printf("my child c2 id is:%d\n",c1);

        }

    }

    else

    {

         printf("I am c having id:%d\n",getpid());

        printf("my child c1 id is:%d\n",c);

    }

    return 0;

}

**Output:**

**WEEK-7:**

1. **AIM:** Program to create threads in Linux. Thread prints 0-4 while the main process prints 20-24

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

void \*thread\_function(void \*arg);

int i, j;

int main() {

    pthread\_t a\_thread;

    pthread\_create(&a\_thread, NULL, thread\_function, NULL);

    pthread\_join(a\_thread, NULL);

    printf("Inside Main Program\n");

    for (j = 20; j < 25; j++) {

        printf("%d\n", j);

        sleep(1);

    }

}

void \*thread\_function(void \*arg) {

    printf("Inside Thread\n");

    for (i = 0; i < 5; i++) {

        printf("%d\n", i);

        sleep(1);

    }

}

**Output:**

1. **AIM:** Program to create a thread. The thread prints numbers from zero to n, where value of n is passed from the main process to the thread. The main process also waits for the thread to finish first and then prints from 20-24.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#include <string.h>

void \*thread\_function(void \*arg);

int i, n, j;

int main() {

    char \*m = "5";

    pthread\_t a\_thread;

    void \*result;

    pthread\_create(&a\_thread, NULL, thread\_function, m);

    pthread\_join(a\_thread, &result);

    printf("Thread joined\n");

    for (j = 20; j < 25; j++) {

        printf("%d\n", j);

        sleep(1);

    }

    printf("thread returned %s\n", (char \*)result);

    return 0;

}

void \*thread\_function(void \*arg) {

    int sum = 0;

    n = atoi(arg);

    for (i = 0; i < n; i++) {

        printf("%d\n", i);

        sleep(1);

    }

    pthread\_exit("Done");

}

**Output:**