**INDEX**

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

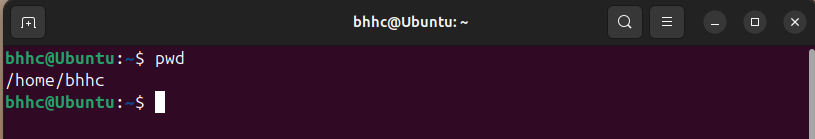
**OS RECORD**

**WEEK-1**

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

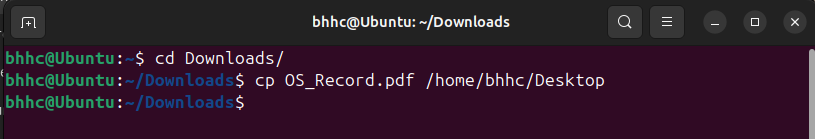
**1. pwd**

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



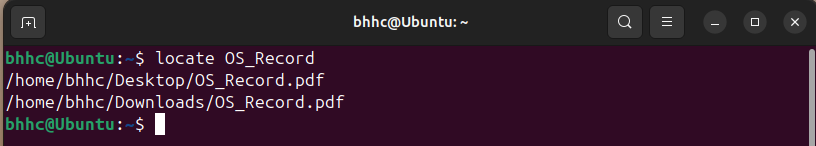
**2. cp**

* **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**:



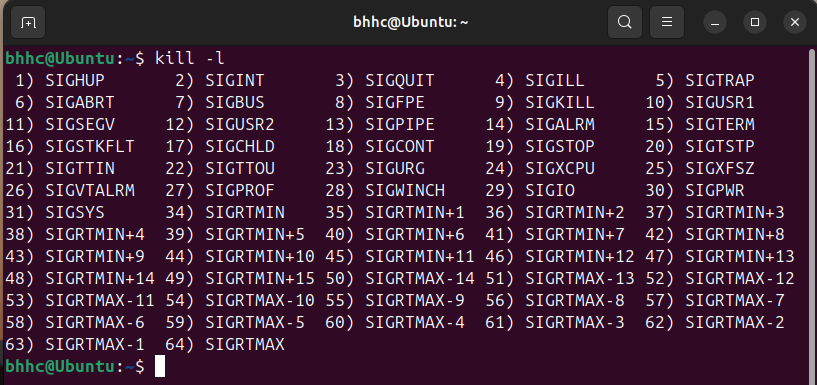
**3. locate**

* **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**:



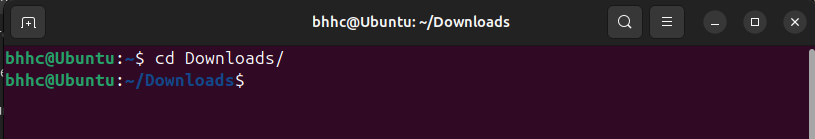
**4. kill**

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



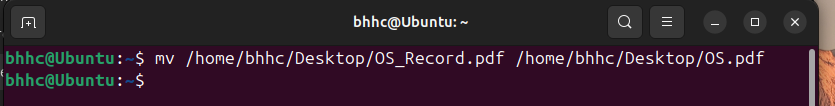
**5. cd**

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



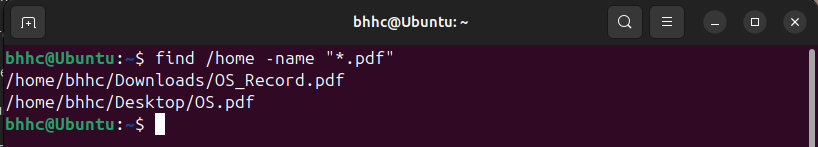
**6. mv**

* **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**:



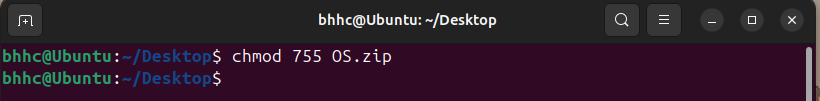
**7. find**

* **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**:



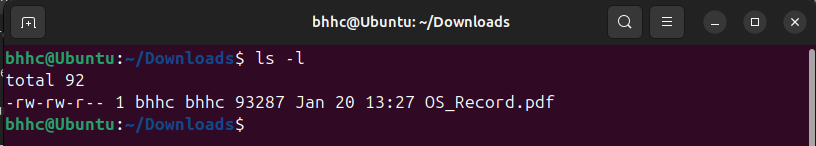
**8. chmod**

* **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**:



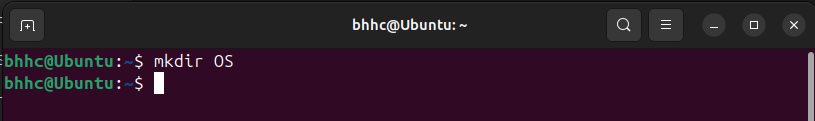
**9. ls**

* **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**:



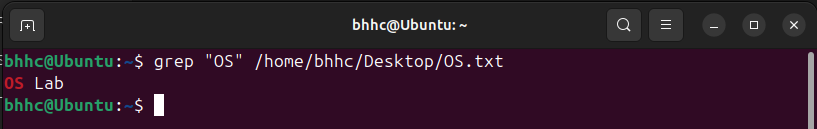
**10. mkdir**

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



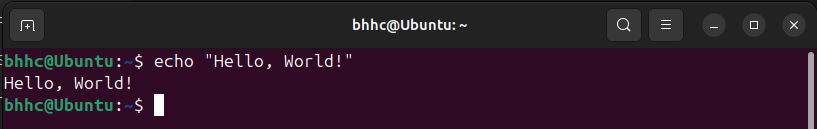
**11. grep**

* **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**:



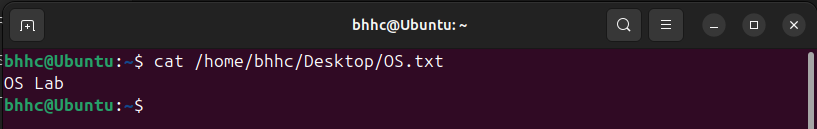
**12. echo**

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



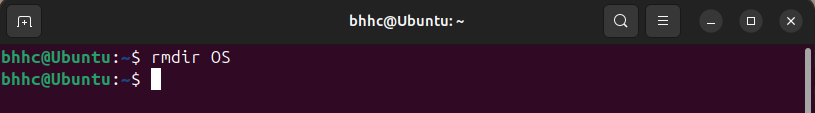
**13. cat**

* **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**:



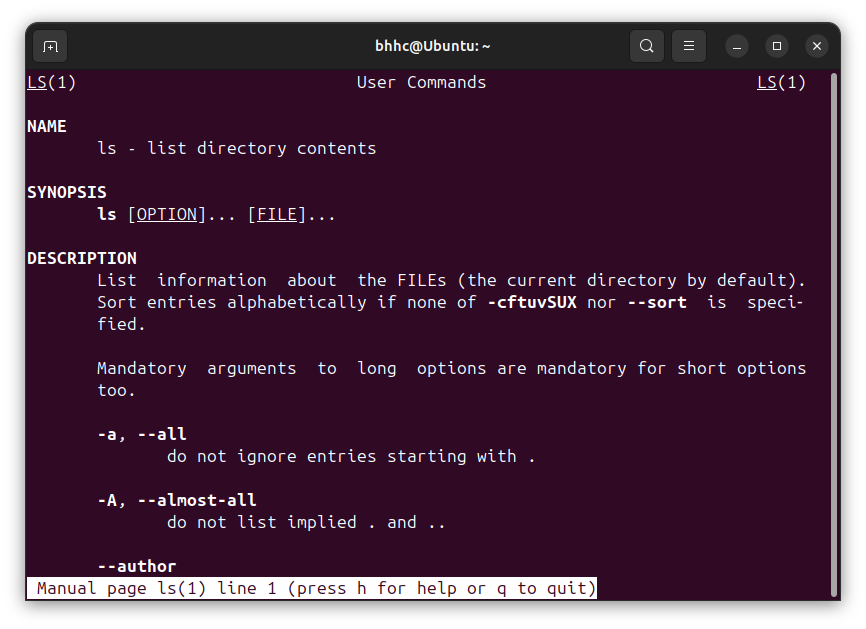
**14. rmdir**

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



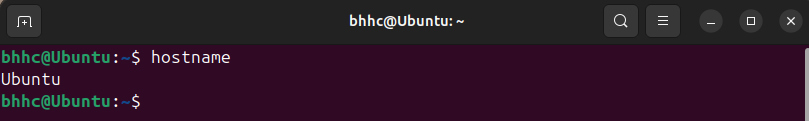
**15. man**

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



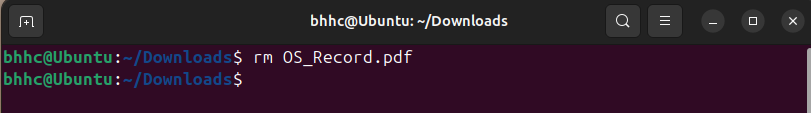
**16. hostname**

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



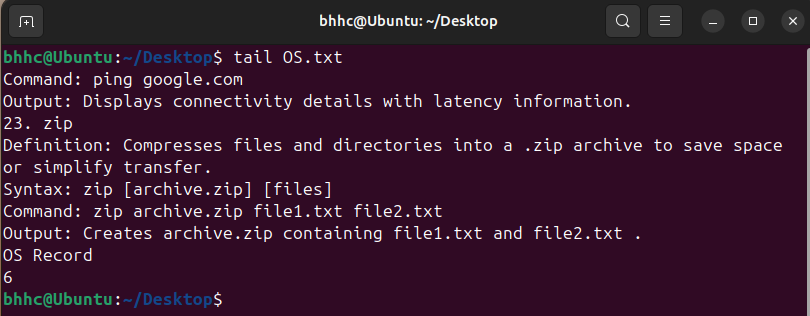
**17. rm**

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



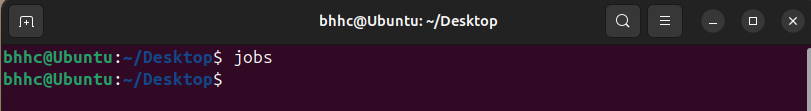
**18. tail**

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



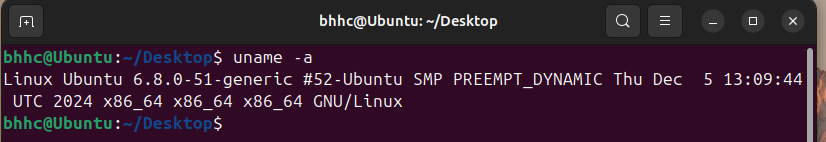
**19. jobs**

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



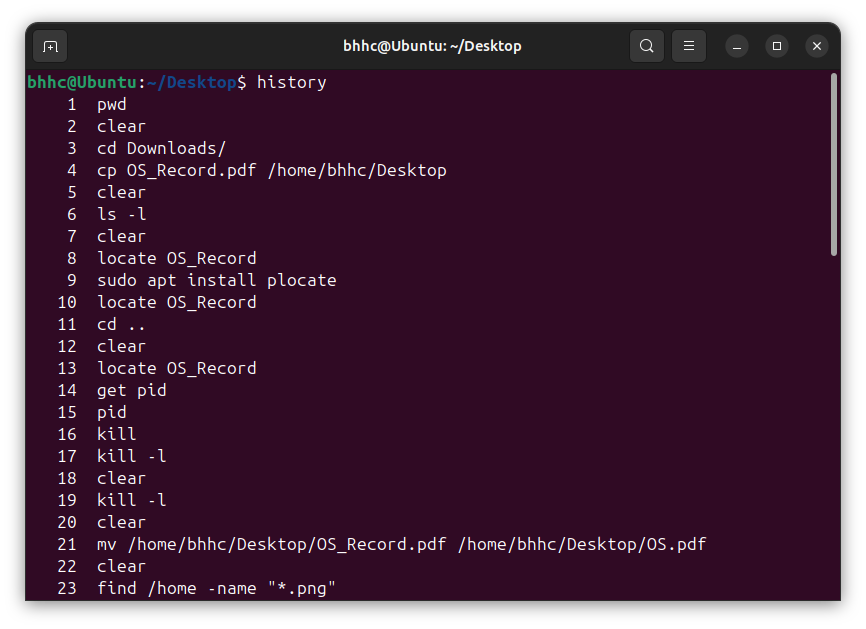
**20. uname**

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



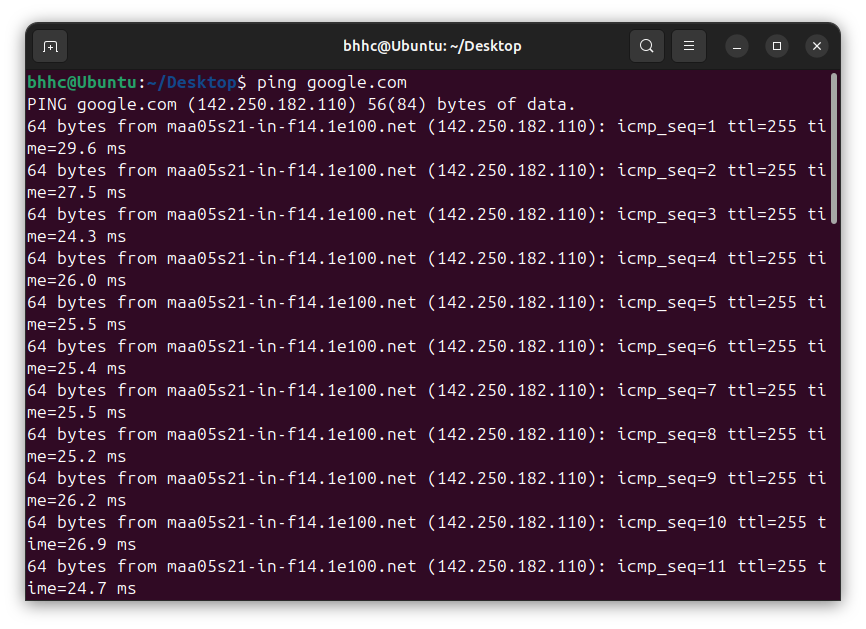
**21. history**

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

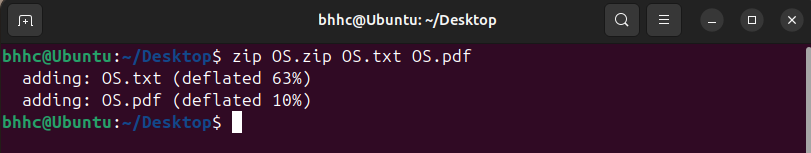


**22. ping**

* **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)**

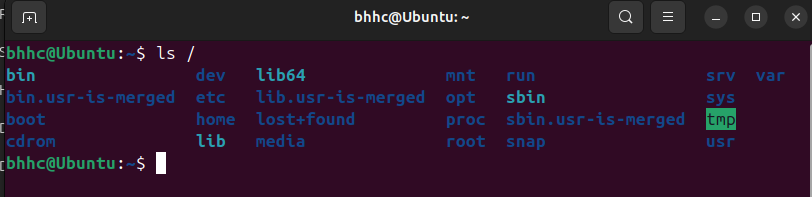
The **proc file system (procfs)** is a virtual file system created dynamically when the system boots and is removed upon shutdown. It serves as a control and information center for the kernel, containing real-time data about system processes. Additionally, it facilitates communication between kernel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

**1. Listing Root Directories:**

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

ls /

**Output**:

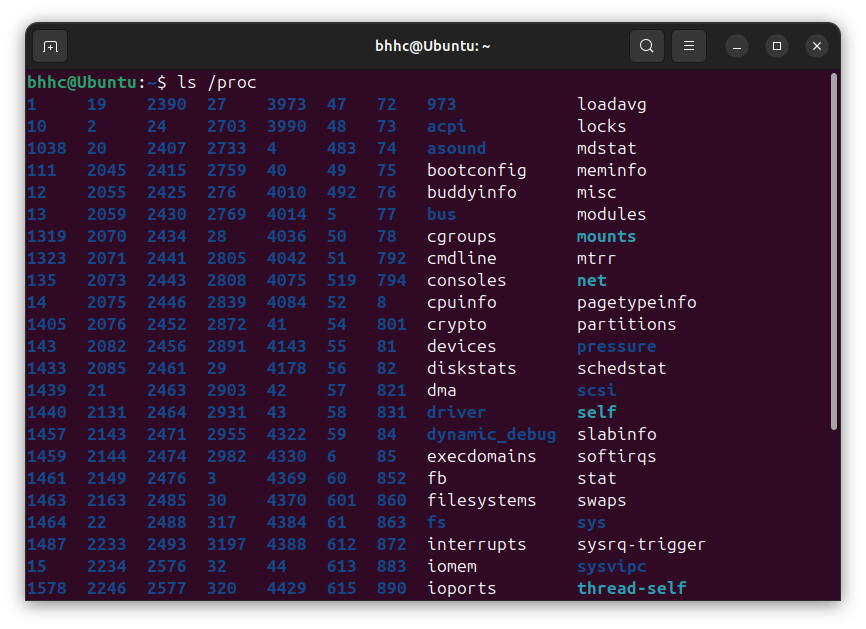


**2. Listing Directories Under /proc:**

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

ls /proc

**Output**:



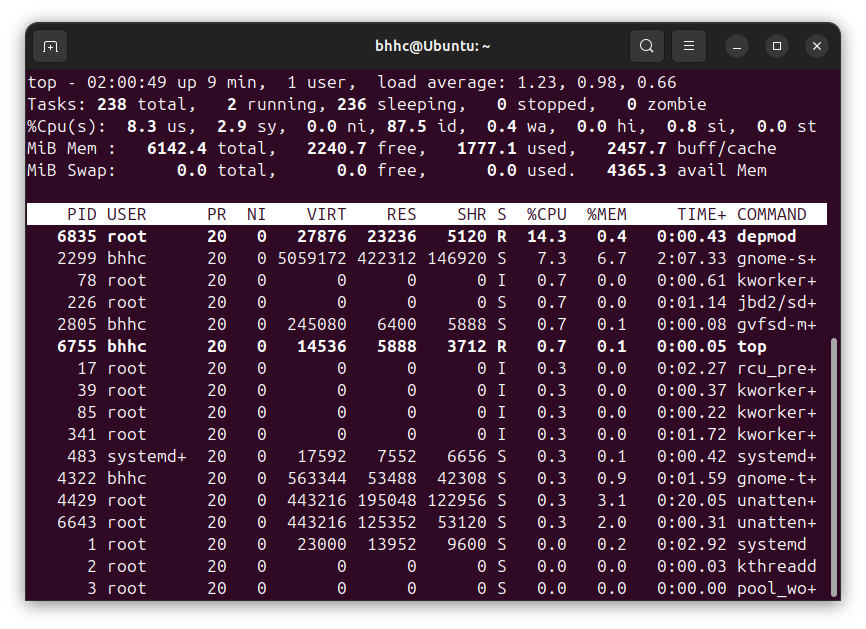
**3. Viewing Active Processes Using the “top” Command:**

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

Top

***PTO***

**Output**:



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

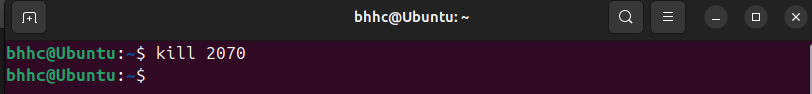
The kill command is used to terminate processes by their Process ID (PID).

kill <PID>

Alternatively, to forcefully kill a process:

kill -9 <PID>

**Output**:



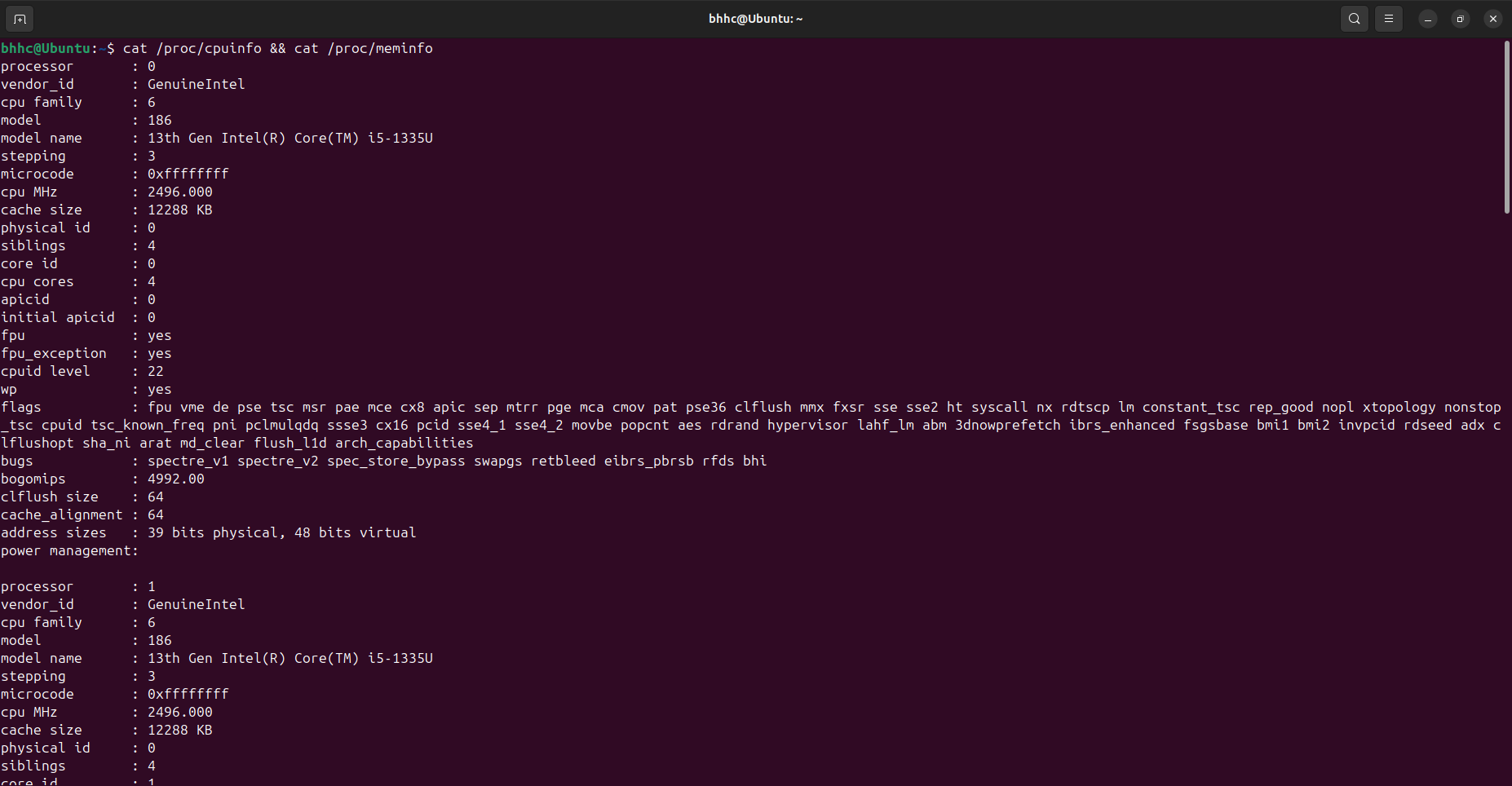
**5. Using the Cat Command:**

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**:

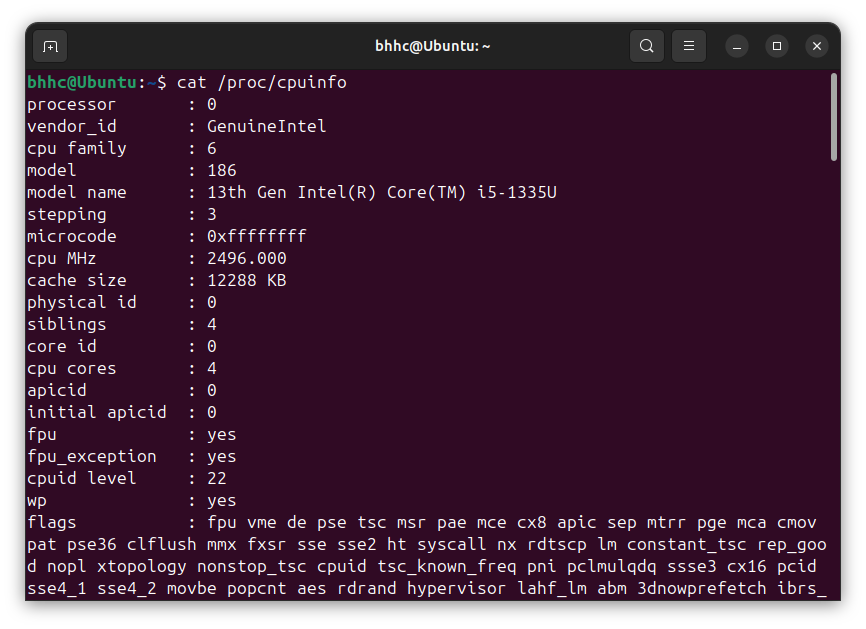


**6. Retrieving CPU Information:**

To obtain basic details about the CPU, use:

cat /proc/cpuinfo

**Output**:



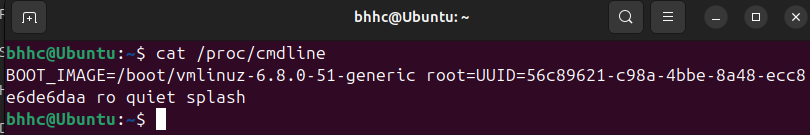
This provides information such as processor type, number of cores, and clock speed.

**7. Retrieving Kernel Information:**

To view the kernel command line arguments:

cat /proc/cmdline

**Output**:

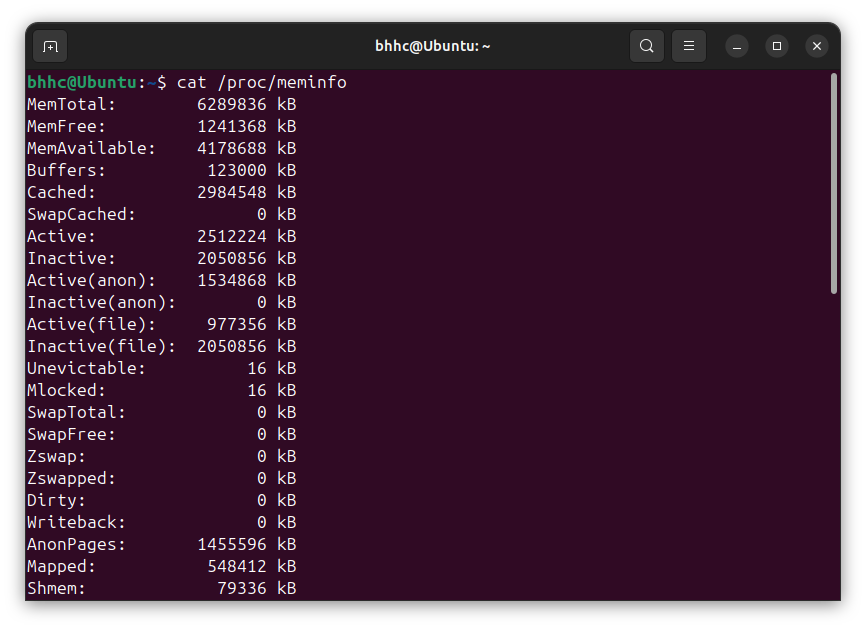


**8. Retrieving Memory Information:**

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()

{

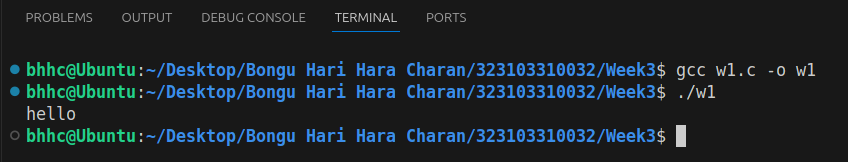
    write(1, "hello\n", 6);

    return 0;

}

***PTO***

**Output**:



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

#include <stdio.h>

#include <unistd.h>

int main()

{

    int count;

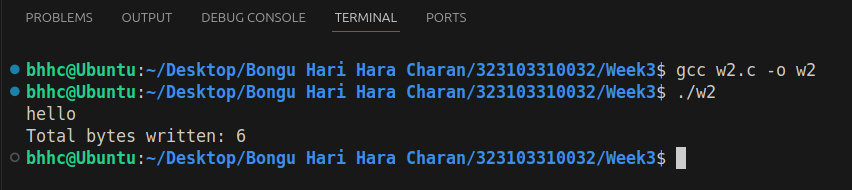
    count = write(1, "hello\n", 6);

    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");

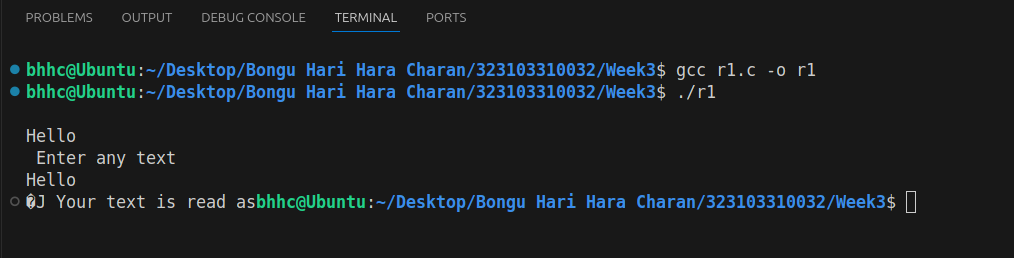
    write(1, buff, 10);

    return 0;

}

**PTO**

**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

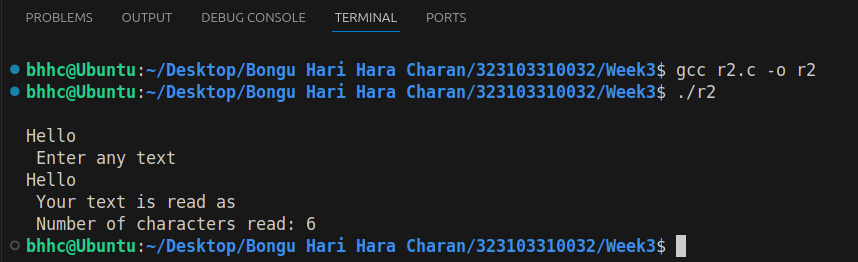
    printf("\n Number of characters read: %d", nread);

printf("\n");

    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);

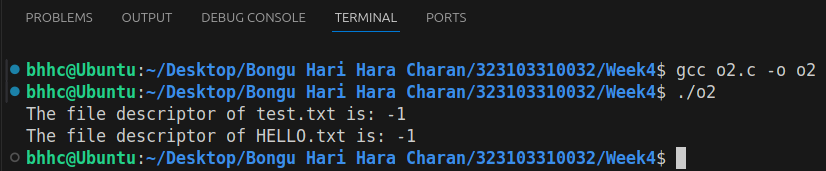
    n = read(fd1, buff, 20);

    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);

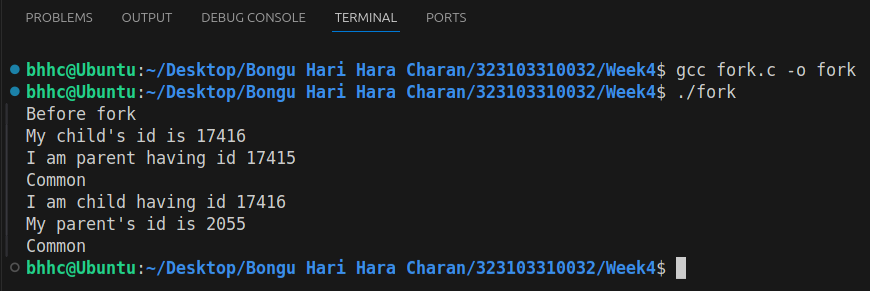
        printf("I am parent having id %d\n", getpid());

    }

    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/wait.h>

#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);

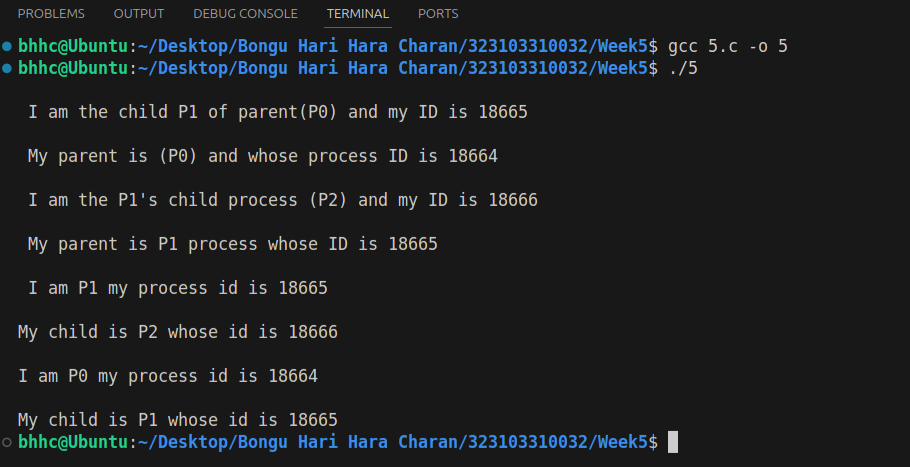
        printf("\nI am P0 my process id is %d\n", getpid());

        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

    {

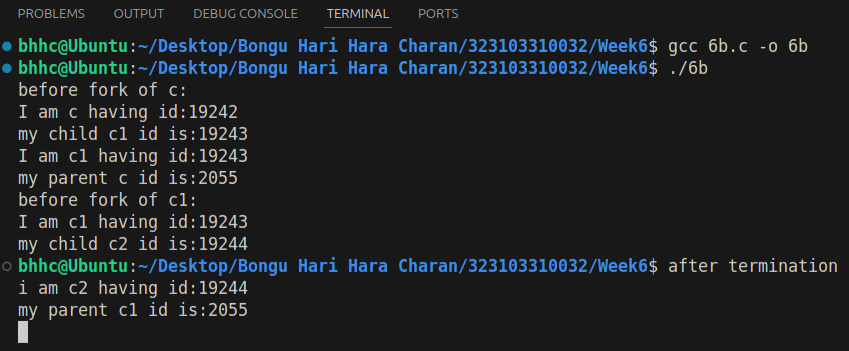
         printf("I am c having id:%d\n",getpid());

        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++) {

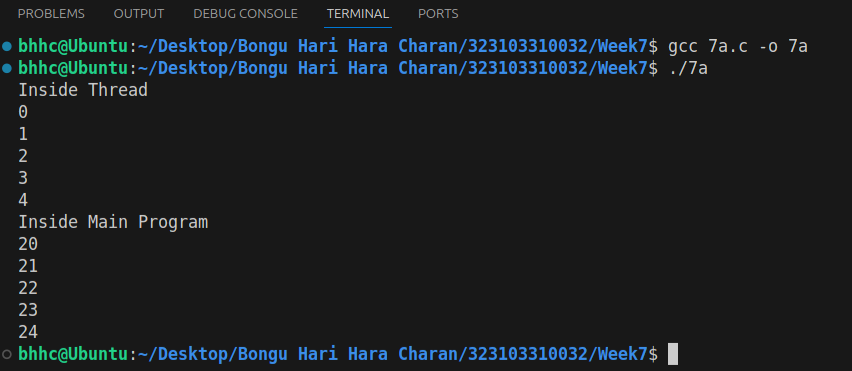
        printf("%d\n", i);

        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);

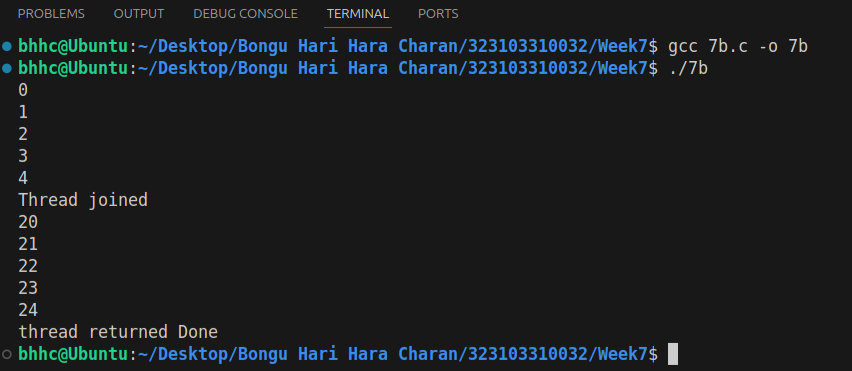
    }

    pthread\_exit("Done");

}

***PTO***

**Output:**



**WEEK-8**

**AIM:** Implement Non-pre-emptive/Pre-emptive CPU Scheduling Algorithms to Find Turnaround Time and Waiting Time

* + 1. **FIRST COME FIRST SERVE (FCFS):**

**AIM**: To write a C program to simulate the CPU scheduling algorithm **First Come First Serve (FCFS)**.

**DESCRIPTION**: The average waiting time using the FCFS algorithm is calculated by first keeping the waiting time of the first process as zero. The waiting time for the second process is the burst time of the first process, and the waiting time for the third process is the sum of the burst times of the first and second processes, and so on. After calculating all the waiting times, the average waiting time is calculated as the average of all the waiting times. The FCFS algorithm serves the process that arrives first.

**ALGORITHM**:

1. Start the process
2. Accept the number of processes in the ready queue
3. For each process in the ready queue, assign the process name and burst time
4. Set the waiting time of the first process as '0' and its burst time as its turnaround time
5. For each process in the ready queue, calculate:
   * Waiting time (n) = Waiting time (n-1) + Burst time (n-1)
   * Turnaround time (n) = Waiting time (n) + Burst time (n)
6. Calculate:
   * Average waiting time = Total waiting time / Number of processes
   * Average turnaround time = Total turnaround time / Number of processes
7. Stop the process

**Program:**

#include <stdio.h>

#include <stdlib.h>

**void** main() {

**int** bt[20], wt[20], tat[20], i, n;

**float** wtavg, tatavg;

system("clear");

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]);

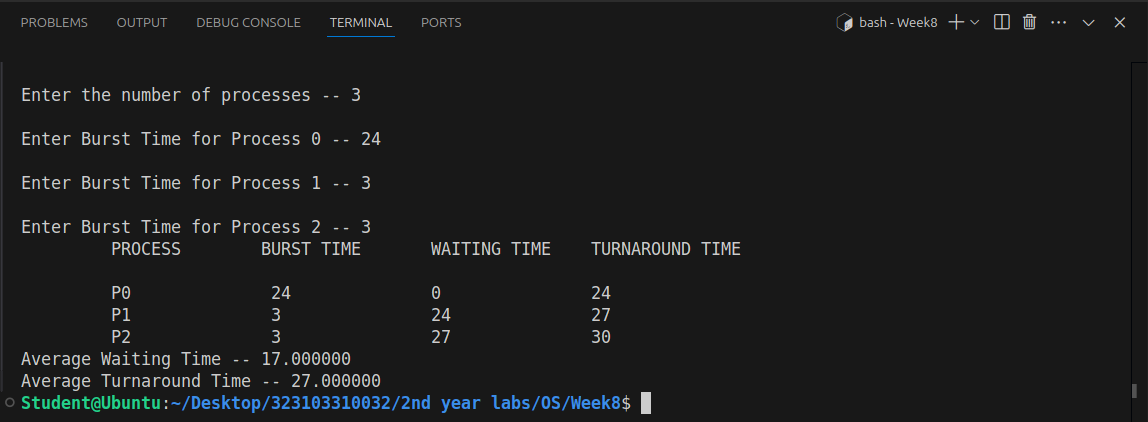
printf("\nAverage Waiting Time -- %f", wtavg / n);

printf("\nAverage Turnaround Time -- %f\n", tatavg / n);

getchar();

}

**Output:**



**B) SHORTEST JOB FIRST (Non-Preemption):**

**AIM**: To write a program to simulate the CPU scheduling algorithm **Shortest Job First (SJF) (Non-preemption)**.

**DESCRIPTION**: To calculate the average waiting time in the Shortest Job First algorithm, the processes are sorted by their burst time in ascending order. Then, the waiting time of each process is calculated by summing up the burst times of all processes that arrive before that process.

**ALGORITHM**:

1. Start the process
2. Accept the number of processes in the ready queue
3. For each process in the ready queue, assign the process ID and accept the CPU burst time
4. Sort the ready queue by burst time in ascending order
5. Set the waiting time of the first process as '0' and its turnaround time as its burst time
6. For each process in the ready queue, calculate:
   * Waiting time(n) = Waiting time(n-1) + Burst time(n-1)
   * Turnaround time(n) = Waiting time(n) + Burst time(n)
7. Calculate:
   * Average waiting time = Total waiting time / Number of processes
   * Average turnaround time = Total turnaround time / Number of processes
8. Stop the process

**Program:**

#include <stdio.h>

#include <stdlib.h>

**void** main() {

**int** p[20], bt[20], wt[20], tat[20], i, k, n, temp;

**float** wtavg, tatavg;

system("clear");

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++)

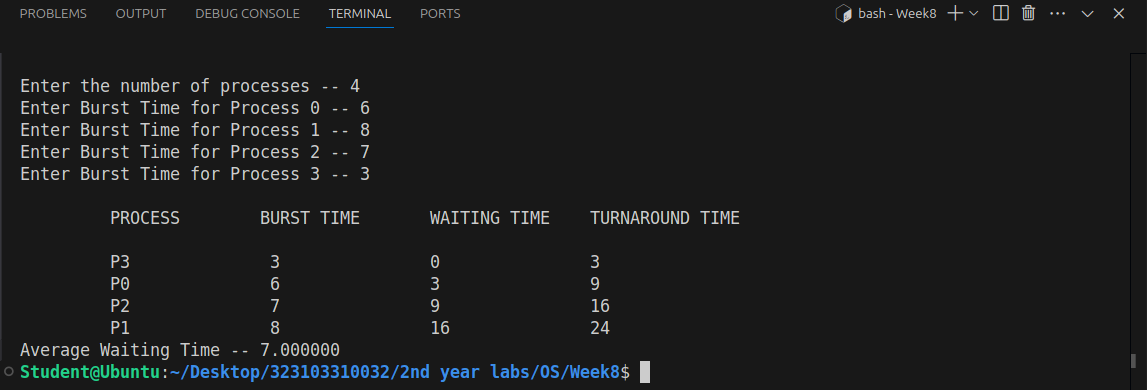
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);

printf("\nAverage Waiting Time -- %f", wtavg / n);

printf("\nAverage Turnaround Time -- %f", tatavg / n);

getchar();

}

**Output:**

**C) ROUND ROBIN:**

**AIM**: To simulate the CPU scheduling algorithm **Round Robin**.

**DESCRIPTION**: The goal is to calculate the average waiting time. A time slice is used where each process gets executed within that time-slice. If the process doesn’t finish in one time slice, it goes back to the queue. We check if the burst time is less than or equal to the time slice and calculate the turnaround time and waiting time accordingly.

**ALGORITHM**:

1. Start the process
2. Accept the number of processes in the ready queue and the time quantum (time slice)
3. For each process in the ready queue, assign the process ID and accept the CPU burst time
4. Calculate the number of time slices for each process:
   * Time slices for process(n) = burst time(n) / time slice
5. If burst time is less than the time slice, then number of time slices = 1.
6. Treat the ready queue as a circular queue:
   * Waiting time for process(n) = waiting time of process(n-1) + burst time of process(n-1) + the time difference in getting the CPU
   * Turnaround time for process(n) = waiting time of process(n) + burst time of process(n) + the time difference in getting the CPU
7. Calculate:
   * Average waiting time = Total waiting time / Number of processes
   * Average turnaround time = Total turnaround time / Number of processes
8. Stop the process

**Program:**

#include <stdio.h>

#include<stdlib.h>

**void** main() {

**int** i, j, n, bu[10], wa[10], tat[10], t, ct[10], max;

**float** awt = 0, att = 0, temp = 0;

system("clear");

printf("Enter the no of processes -- ");

scanf("%d", &n);

**for**(i = 0; i < n; i++) {

printf("\nEnter Burst Time for process %d -- ", i+1);

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

ct[i] = bu[i];

}

printf("\nEnter the size of time slice -- ");

scanf("%d", &t);

max = bu[0];

**for**(i = 1; i < n; i++) {

**if**(max < bu[i]) max = bu[i];

}

**for**(j = 0; j < (max/t) + 1; j++) {

**for**(i = 0; i < n; i++) {

**if**(bu[i] != 0) {

**if**(bu[i] <= t) {

tat[i] = temp + bu[i];

temp = temp + bu[i];

bu[i] = 0;

} **else** {

bu[i] = bu[i] - t;

temp = temp + t;

}

}

}

}

**for**(i = 0; i < n; i++) {

wa[i] = tat[i] - ct[i];

att += tat[i];

awt += wa[i];

}

printf("\nThe Average Turnaround time is -- %f", att / n);

printf("\nThe Average Waiting time is -- %f", awt / n);

printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURNAROUND TIME\n");

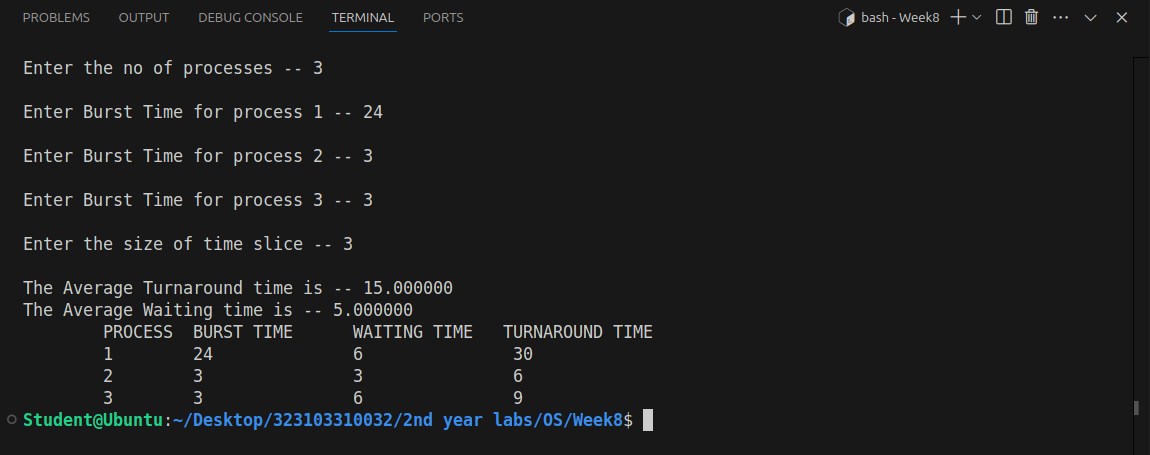
**for**(i = 0; i < n; i++)

printf("\t%d \t %d \t\t %d \t\t %d \n", i+1, ct[i], wa[i], tat[i]);

getchar();

}

**Output:**



**D) PRIORITY:**

**AIM**: To write a C program to simulate the CPU scheduling **Priority algorithm**.

**DESCRIPTION**: The priority algorithm calculates the average waiting time by sorting the processes based on their priorities. The waiting time for each process is obtained by summing up the burst times of all the previous processes.

**ALGORITHM**:

1. Start the process
2. Accept the number of processes in the ready queue
3. For each process in the ready queue, assign the process ID and accept the CPU burst time
4. Sort the ready queue according to priority numbers.
5. Set the waiting time of the first process as '0' and its burst time as its turnaround time
6. Arrange the processes based on their priority
7. For each process in the ready queue, calculate:
   * Waiting time(n) = Waiting time(n-1) + Burst time(n-1)
   * Turnaround time(n) = Waiting time(n) + Burst time(n)
8. Calculate:
   * Average waiting time = Total waiting time / Number of processes
   * Average turnaround time = Total turnaround time / Number of processes
9. Stop the process

**Program:**

#include <stdio.h>

#include <stdlib.h>

**void** main() {

**int** p[20], bt[20], pri[20], wt[20], tat[20], i, k, n, temp;

**float** wtavg, tatavg;

system("clear");

printf("Enter the number of processes --- ");

scanf("%d", &n);

**for**(i = 0; i < n; i++) {

p[i] = i;

printf("Enter the Burst Time & Priority of Process %d --- ", i);

scanf("%d %d", &bt[i], &pri[i]);

}

**for**(i = 0; i < n; i++) {

**for**(k = i + 1; k < n; k++) {

**if**(pri[i] > pri[k]) {

temp = p[i];

p[i] = p[k];

p[k] = temp;

temp = bt[i];

bt[i] = bt[k];

bt[k] = temp;

temp = pri[i];

pri[i] = pri[k];

pri[k] = temp;

}

}

}

wtavg = wt[0] = 0;

tatavg = tat[0] = 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("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND TIME");

**for**(i = 0; i < n; i++)

printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d", p[i], pri[i], bt[i], wt[i], tat[i]);

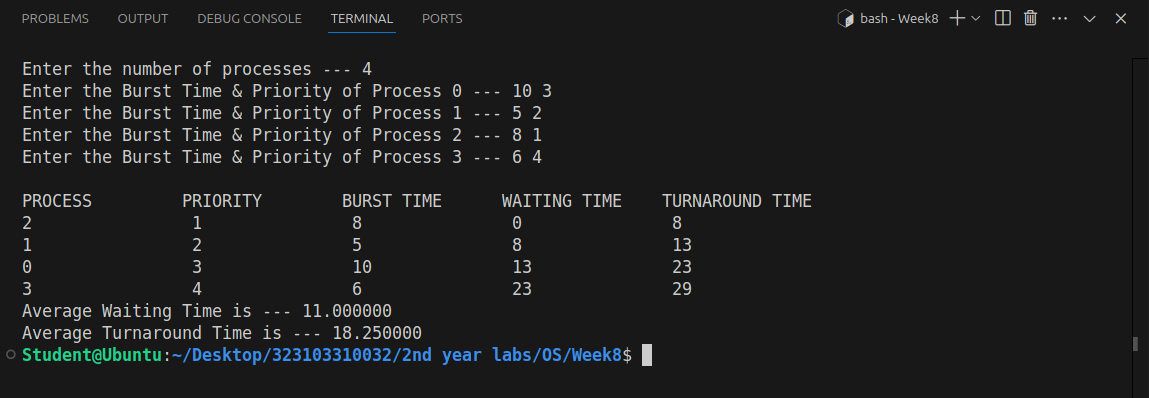
printf("\nAverage Waiting Time is --- %f", wtavg / n);

printf("\nAverage Turnaround Time is --- %f\n", tatavg / n);

getchar();

}

**Output:**



**WEEK-9**

**AIM:** To write a C program to simulate the producer-consumer problem using semaphores.

**DESCRIPTION:**

The **Producer-Consumer problem** is a classic synchronization problem that involves two processes: the producer and the consumer, which share a common, fixed-size buffer. The producer produces items that are placed into the buffer, and the consumer removes (consumes) these items from the buffer. The key issue is to ensure synchronization between the producer and consumer, such that:

1. The producer does not overwrite the buffer when it is full.
2. The consumer does not attempt to consume an item that has not yet been produced.

A solution is often implemented using shared memory and semaphores for synchronization to control access to the buffer. The buffer in this case is implemented as an array, and the producer and consumer processes must use semaphores to ensure they do not step on each other’s toes when accessing the buffer.

**PROBLEM DESCRIPTION:**

1. There is a fixed-size buffer.
2. The producer produces items and the consumer consumes them.
3. To allow both processes to run concurrently, the producer and consumer must be synchronized to prevent conflicts over the shared buffer.
4. The program uses semaphores to synchronize the actions of the producer and consumer, ensuring that the consumer waits for the producer to produce an item before consuming it and that the producer waits if the buffer is full.

**Program:** Producer-consumer problem without using semaphores.

#include <stdio.h>

#include<stdlib.h>

**void** main() {

system("clear");

**int** buffer[10], bufsize, in, out, produce, consume, choice = 0;

in = 0;

out = 0;

bufsize = 10;

**while** (choice != 3) {

printf("\n1. Produce \t 2. Consume \t3. Exit");

printf("\nEnter your choice: ");

scanf("%d", &choice);

**switch**(choice) {

**case** 1:

**if** ((in + 1) % bufsize == out)

printf("\nBuffer is Full");

**else** {

printf("\nEnter the value: ");

scanf("%d", &produce);

buffer[in] = produce;

in = (in + 1) % bufsize;

}

**break**;

**case** 2:

**if** (in == out)

printf("\nBuffer is Empty");

**else** {

consume = buffer[out];

printf("\nThe consumed value is %d", consume);

out = (out + 1) % bufsize;

}

**break**;

**case** 3:

printf("\nExiting...\n");

**break**;

default:

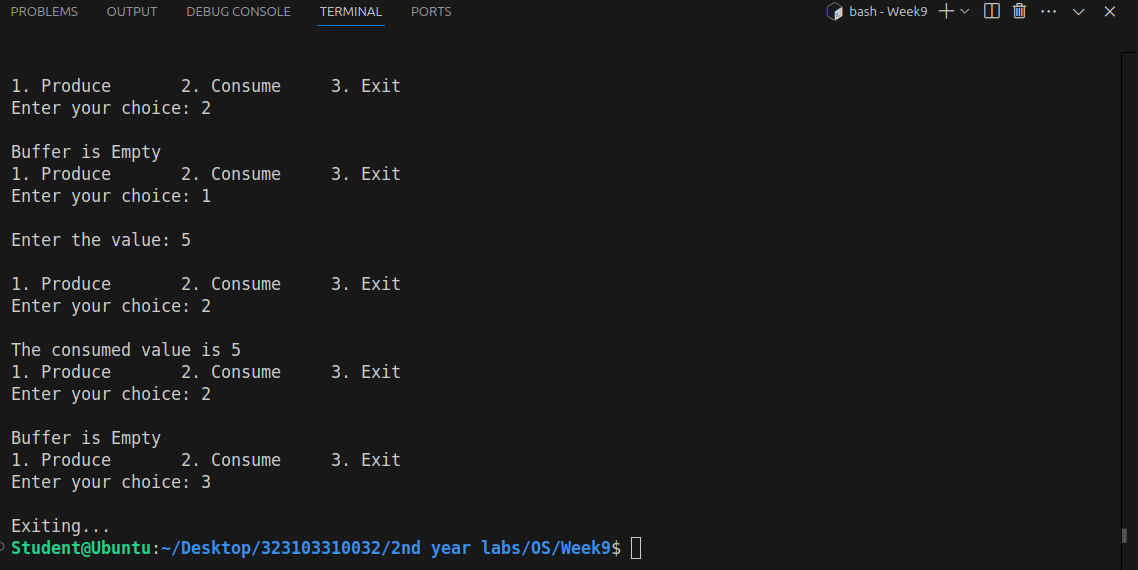
printf("\nInvalid choice!");

}

}

}

**Output:**



**Program:** Producer-consumer problem using semaphores.

#include <stdio.h>

#include <stdlib.h>

#include <semaphore.h>

#define BUF\_SIZE 10

**int** buffer[BUF\_SIZE], in = 0, out = 0;

**sem\_t** empty, full, mutex;

**void** producer() {

**int** produce;

printf("\nEnter the value to produce: ");

scanf("%d", &produce);

sem\_wait(&empty);

sem\_wait(&mutex);

buffer[in] = produce;

in = (in + 1) % BUF\_SIZE;

printf("\nProduced value: %d", produce);

sem\_post(&mutex);

sem\_post(&full);

}

**void** consumer() {

**int** consume;

sem\_wait(&full);

sem\_wait(&mutex);

consume = buffer[out];

out = (out + 1) % BUF\_SIZE;

printf("\nConsumed value: %d", consume);

sem\_post(&mutex);

sem\_post(&empty);

}

**int** main() {

system("clear");

**int** choice;

sem\_init(&empty, 0, BUF\_SIZE);

sem\_init(&full, 0, 0);

sem\_init(&mutex, 0, 1);

**while** (1) {

printf("\n1. Produce \t 2. Consume \t 3. Exit");

printf("\nEnter your choice: ");

scanf("%d", &choice);

**switch** (choice) {

**case** 1:

producer();

**break**;

**case** 2:

consumer();

**break**;

**case** 3:

printf("\nExiting...\n");

sem\_destroy(&empty);

sem\_destroy(&full);

sem\_destroy(&mutex);

**return** 0;

default:

printf("\nInvalid choice! Please try again.");

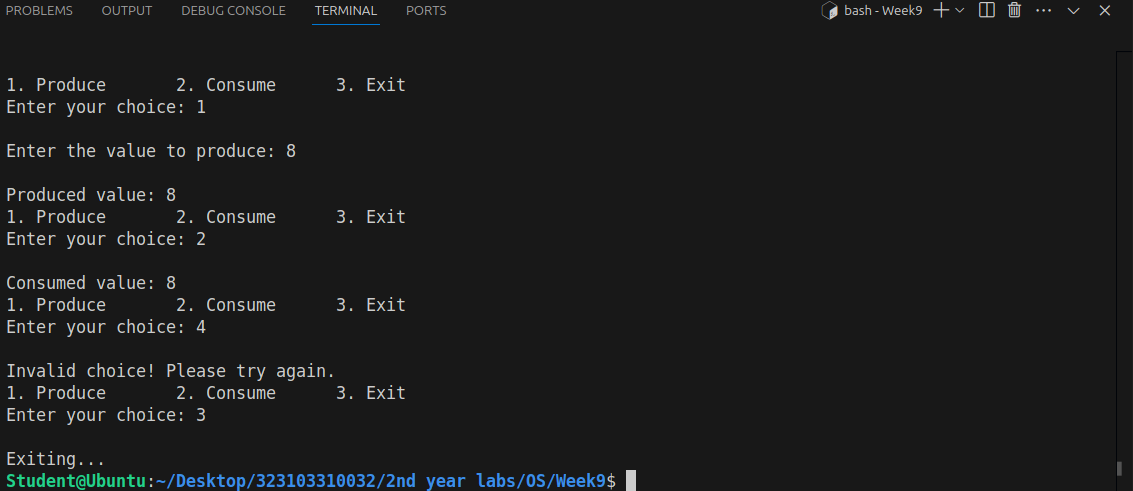
}

}

**return** 0;

}

**Output:**



**WEEK-10**

**AIM:** To implement Banker’s algorithm for the purpose of deadlock avoidance in a system with multiple processes and resources.

**DESCRIPTION:**

The Banker’s algorithm is a resource allocation and deadlock avoidance algorithm that tests for the safety of resource allocation by simulating possible allocations before actually assigning resources.

* It ensures that the system will remain in a **safe state** after allocating resources to a process.
* The algorithm requires three matrices: **Allocation**, **Maximum**, and **Need**.
* It also uses an **Available** vector to keep track of available resources.
* The algorithm works by checking if there exists a sequence of processes (Safe Sequence) that allows all processes to finish execution without leading to a deadlock.

**Program:**

#include <stdio.h>

**int** main() {

**int** p, r, count = 0, i, j, alc[5][3], max[5][3], need[5][3], safe[5], available[3], done[5], terminate = 0;

printf("Enter the number of processes and resources: ");

scanf("%d %d", &p, &r);

printf("Enter allocation matrix (%dx%d):\n", p, r);

**for** (i = 0; i < p; i++) {

**for** (j = 0; j < r; j++) {

scanf("%d", &alc[i][j]);

}

}

printf("Enter maximum resource requirement matrix (%dx%d):\n", p, r);

**for** (i = 0; i < p; i++) {

**for** (j = 0; j < r; j++) {

scanf("%d", &max[i][j]);

}

}

printf("Enter available resources: ");

**for** (i = 0; i < r; i++) {

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

}

*// Calculate Need matrix*

printf("\nNeed matrix:\n");

**for** (i = 0; i < p; i++) {

**for** (j = 0; j < r; j++) {

need[i][j] = max[i][j] - alc[i][j];

printf("%d\t", need[i][j]);

}

printf("\n");

}

*// Initialize process completion status*

**for** (i = 0; i < p; i++) {

done[i] = 0;

}

*// Find safe sequence*

**while** (count < p) {

**for** (i = 0; i < p; i++) {

**if** (done[i] == 0) {

**for** (j = 0; j < r; j++) {

**if** (need[i][j] > available[j])

**break**;

}

**if** (j == r) { *// If all needs are met*

safe[count] = i;

done[i] = 1;

**for** (j = 0; j < r; j++) {

available[j] += alc[i][j]; *// Release resources*

}

count++;

terminate = 0;

} **else** {

terminate++;

}

}

}

**if** (terminate == (p - 1)) {

printf("Safe sequence does not exist.\n");

**break**;

}

}

*// If safe sequence exists, print it*

**if** (terminate != (p - 1)) {

printf("\nAvailable resources after completion:\n");

**for** (i = 0; i < r; i++) {

printf("%d\t", available[i]);

}

printf("\nSafe sequence:\n");

**for** (i = 0; i < p; i++) {

printf("P%d\t", safe[i]);

}

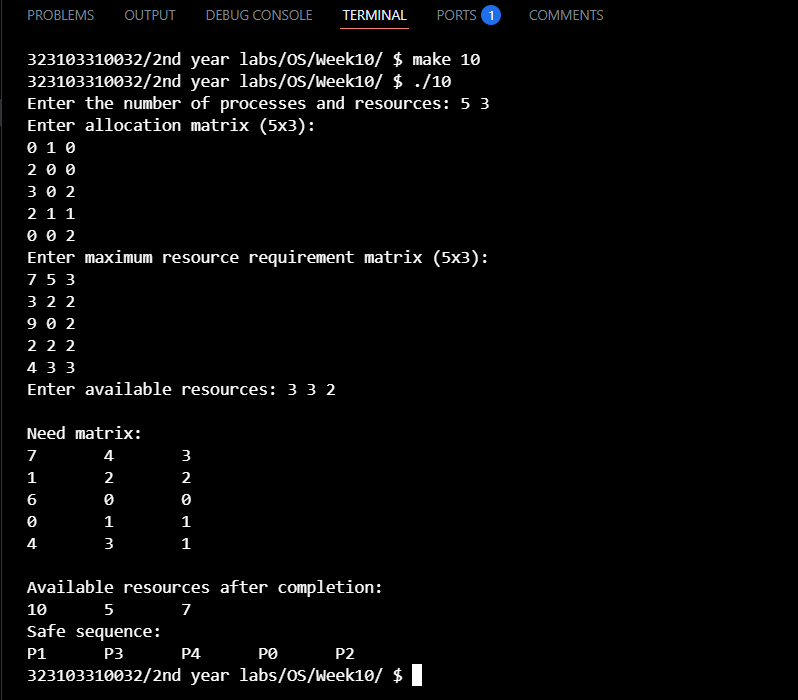
printf("\n");

}

**return** 0;

}

**Output:**

****

**WEEK-11**

**AIM:** Implement the MVT and MFT Memory Management techniques

**Program(MVT):**

#include <stdio.h>

**void** main() {

**int** ms, mp[10], i, temp, n = 0;

**char** ch = 'y';

printf("Enter the total memory available (in Bytes) ");

scanf("%d", &ms);

temp = ms;

**for** (i = 0; ch == 'y'; i++, n++) {

printf("Enter memory required for process %d (in Bytes) -- ", i + 1);

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

**if** (mp[i] <= temp) {

printf("Memory is allocated for Process %d\n", i + 1);

temp = temp - mp[i];

} **else** {

printf("Memory is Full\n");

**break**;

}

printf("Do you want to continue(y/n) -- ");

scanf(" %c", &ch);

}

printf("Total Memory Available -- %d\n", ms);

printf("PROCESS\t\tMEMORY ALLOCATED\n");

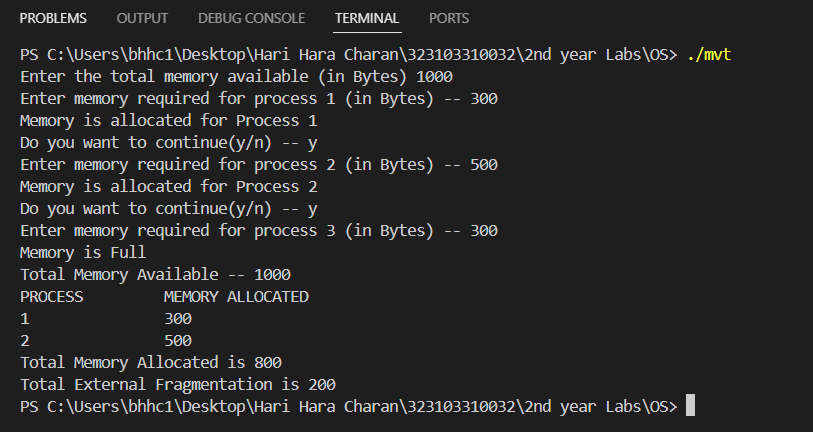
**for** (i = 0; i < n; i++)

printf("%d\t\t%d\n", i + 1, mp[i]);

printf("Total Memory Allocated is %d\n", ms - temp);

printf("Total External Fragmentation is %d\n", temp);

}

**  
Output:**

**Program(MFT):**

#include<stdio.h>

**int** main(){

**int** p, bs, nob, totalme;

printf("Total memory: ");

scanf("%d", &totalme);

printf("Enter no.of processes: ");

scanf("%d", &p);

printf("Enter size of each block: ");

scanf("%d", &bs);

nob = totalme / bs;

**int** process[p];

printf("Enter memory required for each process: ");

**for**(**int** i = 0; i < p; i++){

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

}

**int** tif = 0, tef = 0, cnt = 0;

tef = totalme - (bs \* nob);

printf("No. of Blocks available in memory %d\n", nob);

printf("PROCESS\tMEMORY REQUIRED\tALLOCATED\tINTERNAL FRAGMENTATION\n");

**int** i;

**for**(i = 0; i < p && cnt < nob; i++){

**if**(process[i] <= bs){

printf("%d\t%d\t\tYES\t\t%d", i + 1, process[i], bs - process[i]);

tif += bs - process[i];

cnt++;

} **else** {

printf("%d\t%d\t\tNO", i + 1, process[i]);

}

printf("\n");

}

**if**(i < p){

printf("Memory full, remaining processes cannot be accomodated\n");

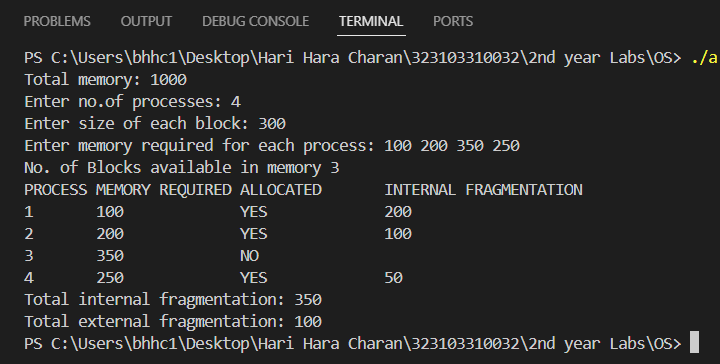
}

printf("Total internal fragmentation: %d\n", tif);

printf("Total external fragmentation: %d\n", tef);

**return** 0;

}



**Output:**