Ex. No: 1

Date:

**BASIC LINUX COMMANDS**

**Problem Statement:**

To execute basic Linux commands.

**Problem Description:**

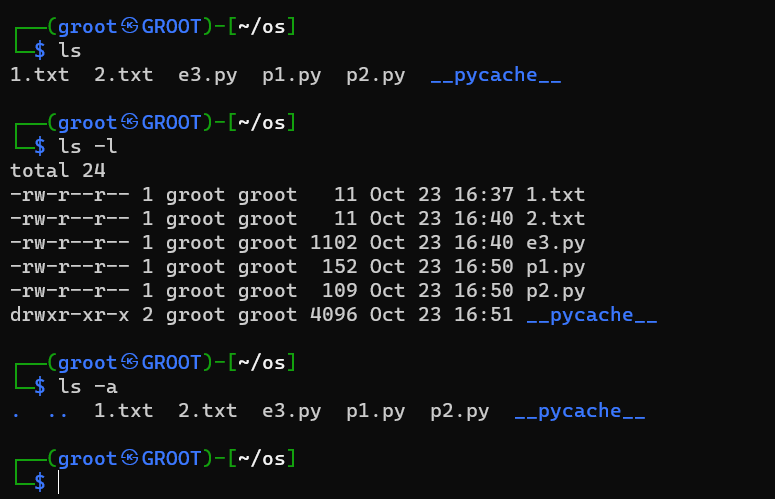
To execute all the basic Linux commands with various options:

**1. ls - List Files**

* Description: Lists files and directories in the current directory.
* Synopsis: ls [options] [files or directories]
* Options:

1. -l: Long format, providing detailed information about files.
2. -a: Include hidden files (those starting with a dot).
3. -t: Sort by modification time.

* Syntax: **ls -l, ls -a, ls -t**

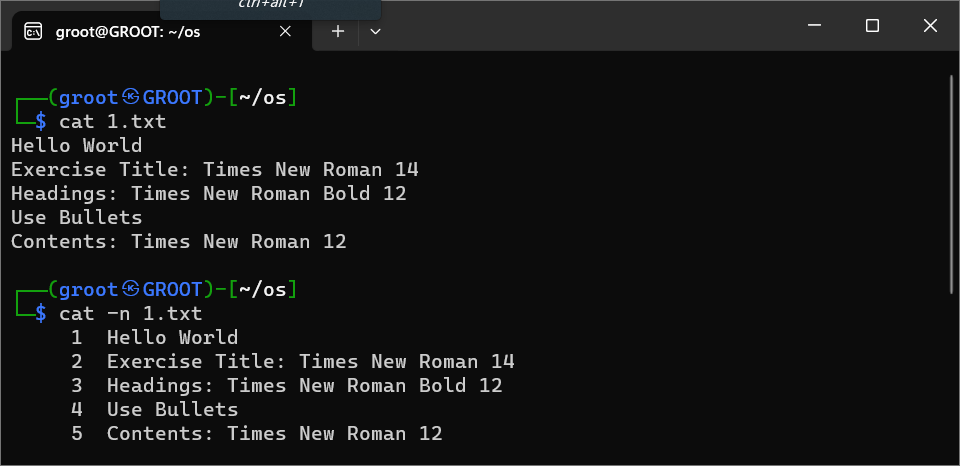


**2. cat** - Concatenate and Display Files

* Description: Displays the contents of one or more files.
* Synopsis: cat [options] [files]
* Options:

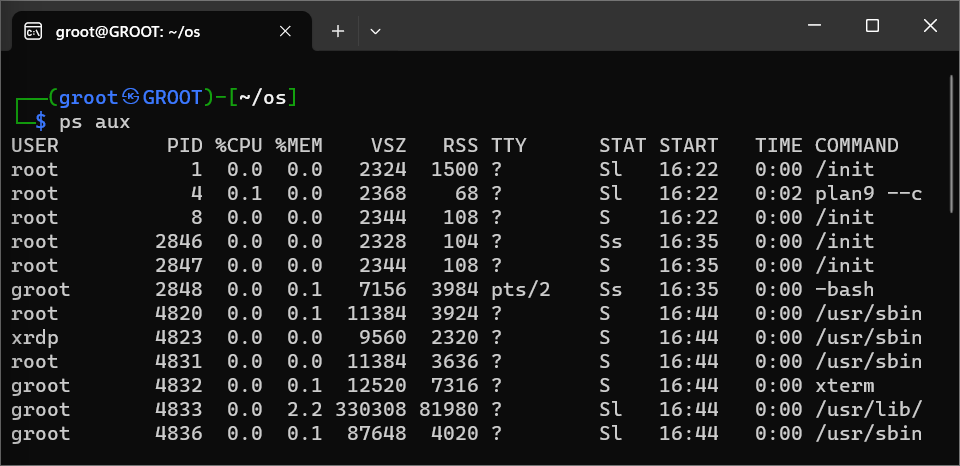
1. -n: Number lines when displaying the file.
2. -E: Show a "$" at the end of each line.
3. -s: Squeeze multiple blank lines into one

* Example: cat filename.txt

****

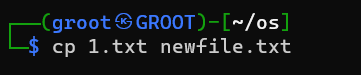
**3. ps - Process Status**

* Description: Displays information about running processes.
* Synopsis: ps [options]
* Example: ps aux



**4. cp - Copy Files and Directories**

* Description: Copies files or directories from one location to another.
* Synopsis: cp [options] source destination
* Example: cp file.txt newfile.txt



Newfile.txt will be created with all the contents in oslab.txt

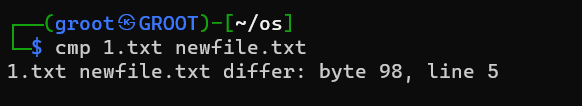
**5. echo - Print Text**

* Description: Prints text to the terminal.
* Synopsis: echo [options] [text]
* Example: echo "Hello, World"



**6.** **cmp - Compare Two Files**

* Description: Compares two files byte by byte and displays the first differing byte's offset.
* Synopsis: cmp [options] file1 file2
* Example: cmp file1.txt file2.txt



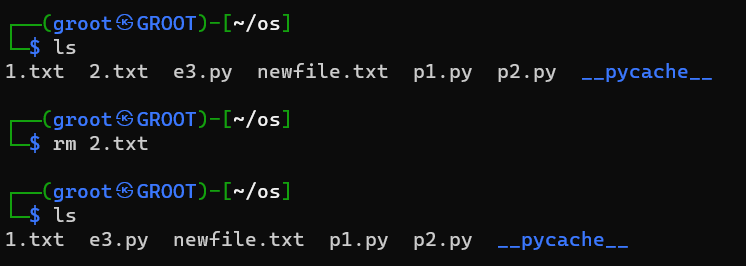
**7. pwd - Print Working Directory**

* Description: Displays the current working directory's absolute path.
* Synopsis: pwd
* Example: pwd



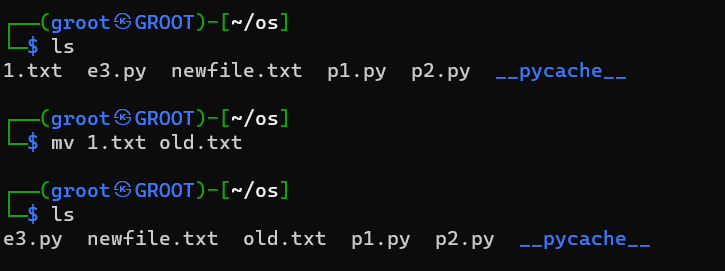
**8. rm - Remove Files and Directories**

* Description: Deletes files and directories.
* Synopsis: rm [options] [files or directories]
* Example: rm file.txt

****

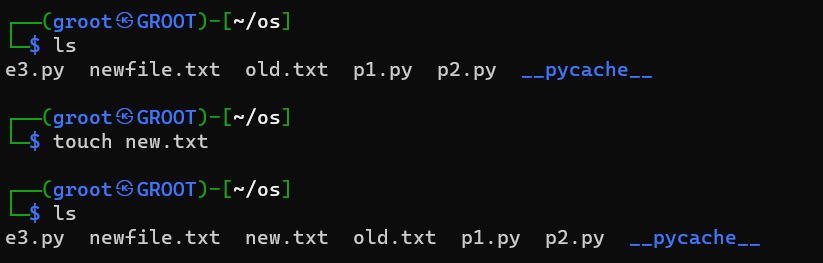
**9. mv - Move or Rename Files and Directories**

* Description: Moves or renames files and directories.
* Synopsis: mv [options] source destination
* Example: mv oldfile.txt newfile.txt



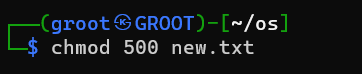
**10. touch - Create Empty Files**

* Description: Creates empty files or updates access and modification timestamps.
* Synopsis: touch [options] [files]
* Example: touch newfile2.txt



**11. chmod - Change File Permissions**

* Description: Modifies file permissions (read, write, execute) for users, groups, and others.
* Synopsis: chmod [options] mode file
* Example: chmod 500 newfile.txt

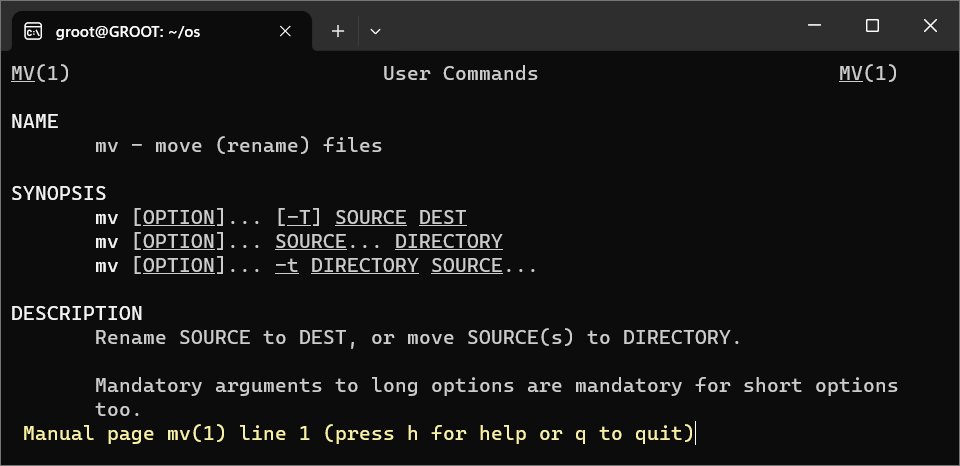
****

**12. clear - Clear the Terminal**

* Description: Clears the terminal screen.
* Synopsis: clear
* Example clear

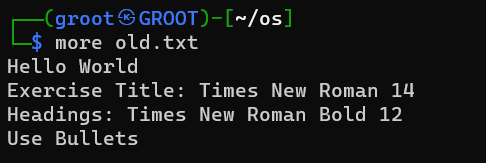
**13. man - Manual Pages**

* Description: Displays the manual page for a given command or topic.
* Synopsis: man [command or topic]
* Example: man mv



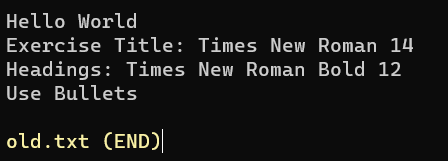
**14. more - View Text Files Page by Page**

* Description: Allows you to view the contents of a text file one page at a time.
* Synopsis: more [options] file
* Example: more tewfile.txt



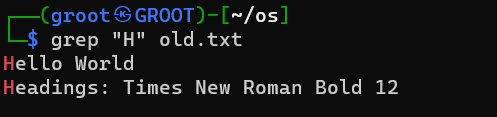
**15. less - View Text Files Page by Page (with backward navigation)**

* Description: Similar to more, but allows backward navigation through the text.
* Synopsis: less [options] file
* Example: less newfil2.txt



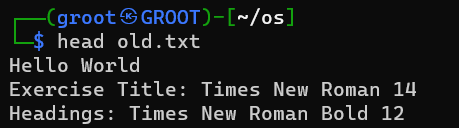
**16. grep - Search Text**

* Description: Searches for a pattern or text within one or more files.
* Synopsis: grep [options] pattern [files]
* Example: grep “A” newfile.txt



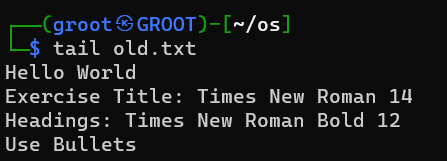
**17. head - Display the Beginning of Files**

* Description: Displays the first few lines of a text file.
* Synopsis: head [options] [files]
* Example: head newfile2.txt



**18. tail - Display the End of Files**

* Description: Displays the last few lines of a text file.
* Synopsis: tail [options] [files]
* Example: tail newfile2.txt



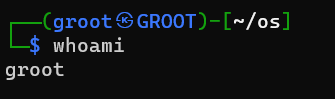
**19. sort - Sort Lines in Text Files**

* Description: Sorts the lines in a text file.
* Synopsis: sort [options] [files]
* Example: sort newfile2.txt



**20. whoami - Display Current User**

* Description: Displays the username of the current user.
* Synopsis: whoami
* Example: whoami



**Result:**

Thus, all the basic linux commands were executed successfully

Ex. No: 2

Date:

**SYSTEM CALLS PROGRAMMING**

**Problem Statement:**

Create a simple Python program that uses system calls for process management. The program should demonstrate the use of fork(), getpid(), getppid(), sleep(), exit().

**Problem Description:**

This Python program illustrates process management using system calls like fork(), getpid(), getppid(), sleep(), and exit(). It initiates with displaying the parent process's PID, then creates a child process, demonstrating both the child's PID and a simulated delay. Following this, the child process exits gracefully. Meanwhile, the parent process waits for the child process to complete and displays the child's exit status, offering a succinct demonstration of process control.

**Synopsis:**

1. **fork():** Creates a new process by duplicating the current one.
2. **getpid():** Retrieves the Process ID (PID) of the current process.
3. **getppid():** Retrieves the PID of the parent process.
4. **sleep():** Delays process execution for a specified time.
5. **exit():** Terminates the process, providing an exit status.

**Code 2a:**

import os

import time

def child\_process():

    print("Child Process - PID:", os.getpid())

    print("Child Process - Parent PID:", os.getppid())

    time.sleep(2)

    print("Child Process - Exiting")

    os.\_exit(0)

def main():

    print("Parent Process - PID:", os.getpid())

    print("Parent Process - Forking a Child Process...")

    child\_pid = os.fork()

    if child\_pid == 0:

        child\_process()

    else:

        print("Parent Process - Waiting for the child process to complete...")

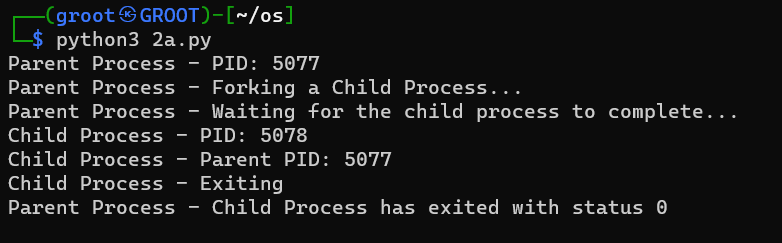
        \_, status = os.wait()

        print("Parent Process - Child Process has exited with status", status)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Output:**

****

**Problem Statement:**

Create a simple Python program that uses system calls for file operations. The program should demonstrate the use of read(), write(), and close() system calls.

**Problem Description:**

This Python program highlights file operations using system calls like read(), write(), and close(). It starts by opening a file for read and write access, then writes a sample text to the file and repositions the file cursor. Subsequently, it reads the data from the file, prints it to the console, and closes the file. This concise program effectively showcases the core file operation system calls.

1. **Synopsis:**
2. **read():** Reads data from a file descriptor.
3. **write():** Writes data to a file descriptor.
4. **close():** Closes a file descriptor, releasing associated resources.

**Code 2b:**

import os

def file\_operations(filename):

    try:

        fd = os.open(filename, os.O\_RDWR | os.O\_CREAT)

        if fd:

            os.write(fd, b"This is a sample text.")

            os.lseek(fd, 0, os.SEEK\_SET)

            data = os.read(fd, 1024)

            print("Read data:", data.decode())

            os.close(fd)

        else:

            print("Error opening file")

    except OSError as e:

        print("File operation error:", e)

def main():

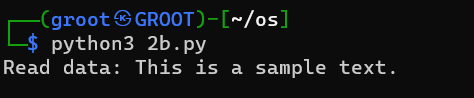
    filename = "sample.txt"

    file\_operations(filename)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Output:**

****

**Result:**

Thus, system calls for process management and file management were executed successfully

Ex No: 3

Date:

**SYSTEM CALLS PROGRAMMING**

**Problem Statement:**

Develop a menu-driven program using the exec() system call to execute Linux commands, including ls, cat, cp, echo, ps, rm, mv, man, chmod, and clear.

**Problem Description:**

Design a menu-driven program using the execl() system call to execute Linux commands like ls, cat, cp, echo, ps, rm, mv, man, chmod, and clear, providing an interactive interface for users to conveniently run these commands. Ensure secure handling of user input to prevent vulnerabilities like command injection.

**Synopsis:**

1. **ls:** List files and directories.
2. **cat:** Concatenate and display file contents.
3. **cp:** Copy files and directories.
4. **echo:** Display text.
5. **ps:** List running processes.
6. **rm:** Remove files and directories.
7. **mv:** Move or rename files and directories.
8. **man**: View manual pages.
9. **chmod:** Change file permissions.
10. **clear:** Clear the terminal screen.

**Code:**

import os

command\_path =['/bin/ls','/bin/cat','/bin/cp','/bin/echo','/bin/ps','/bin/rm','/bin/mv','/usr/bin/man','/bin/chmod','/usr/bin/clear']

print('''

1.ls

2.cat

3.cp b

4.echo

5.ps

6.rm

7.mv

8.man

9.chmod

10.clear

11.exit

'''

)

x=int(input("enter choice"))

while x!=11:

    i=x

    if i==11:

        break

    elif i==1:

        os.execl(command\_path[0], 'ls', '-l')

    elif i==2:

        os.execl(command\_path[1], 'cat', '1.txt')

    elif i==3:

        os.execl(command\_path[2], 'cp ', '1.txt' '2.txt')

    elif i==4:

        os.execl(command\_path[3], 'echo', 'Hello World')

    elif i==5:

        os.execl(command\_path[4], 'ps', '-aux')

    elif i==6:

        os.execl(command\_path[5], 'rm', '2.txt')

    elif i==7:

        os.execl(command\_path[6], 'mv', '1.txt','2.txt')

    elif i==8:

        os.execl(command\_path[7], 'man', 'ls')

    elif i==9:

        os.execl(command\_path[8], 'chmod', '755','2.txt')

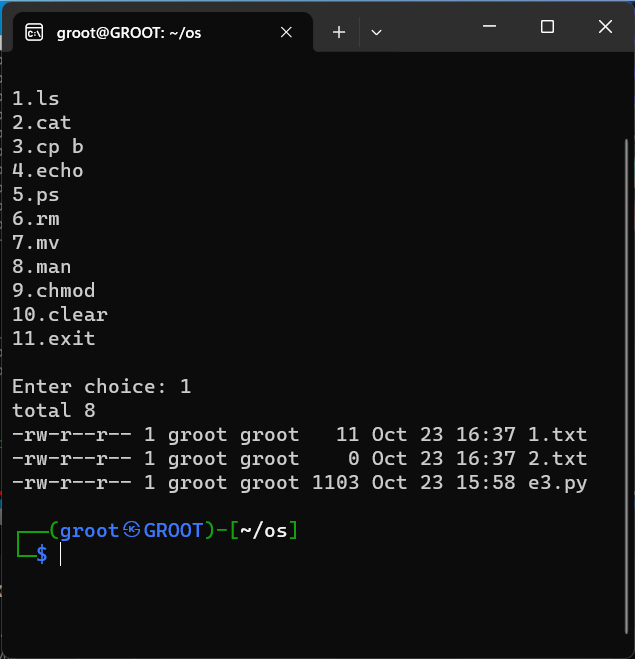
    elif i==10:

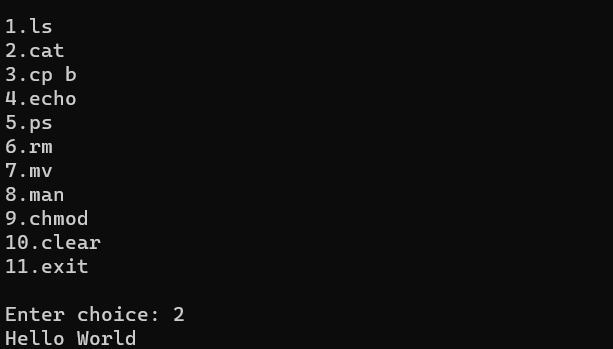
        os.execl(command\_path[9], 'clear')

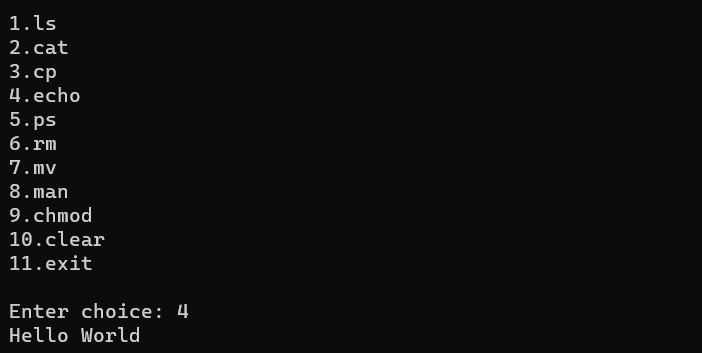
    else:

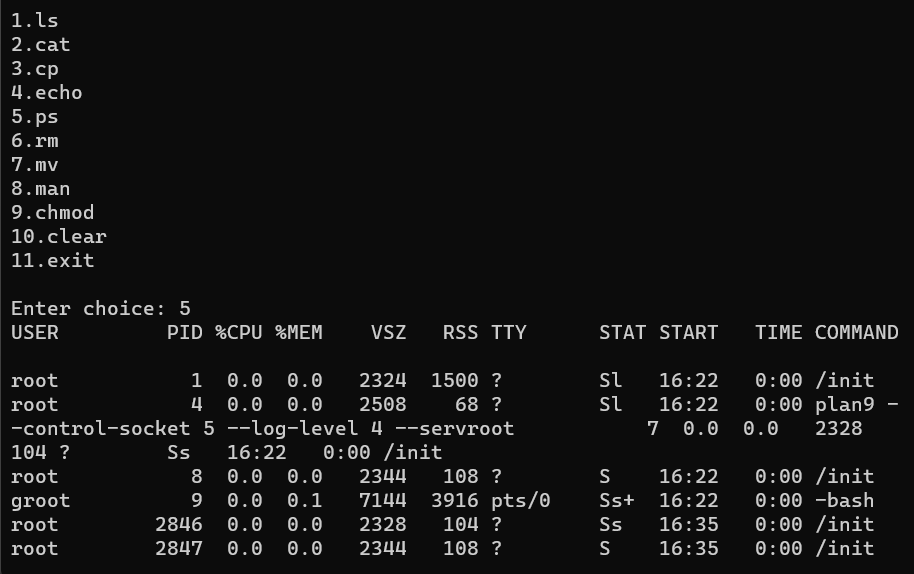
        print("enter a valid choice")

**Output:**

****

****

****

****

**Problem Statement:**

Use exec() system call for process 1 to call process 2 and process 2 to call process 1 to create an infinite loop.

**Problem Description:**

This exercise demonstrates an infinite loop created between two processes, Process 1 and Process 2, through the use of the `exec()` system call. Process 1 invokes Process 2 with `exec()`, and Process 2 subsequently triggers Process 1 using the same system call, resulting in an endlessly repeating execution pattern.

**Synopsis:**

**exec():** replaces the current process with a new one, loading and executing a different program.

**Code:**

**Process 1:**

def main():

    i=int(input("Process 1; Input: "))

    if i!=0:

        import p2

        p2.main(i)

if \_\_name\_\_ == '\_\_main\_\_':

    exec("main()")

**Process 2:**

def main(i):

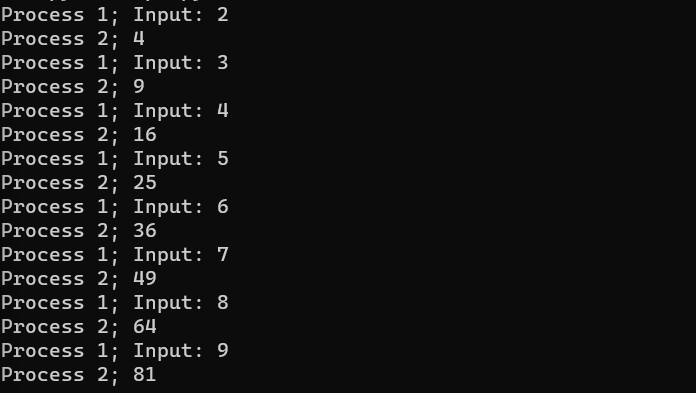
    if i!=0:

        import p1

        print("Process 2;",(i)\*(i))

        exec('p1.main()')

**Output:**

****

Ex. No: 4

Date:

**SIMULATION OF LINUX COMMANDS**

**Problem Statement:**

To simulate ls, cp, cat, mv, grep Linux commands using subprocess.

**Problem Description:**

Simulate Linux commands like ls, cp, cat, mv, and grep using Python's subprocess module, allowing you to execute these commands from within a Python script, mimicking their behavior and functionality on a Linux system.

**Synopsis:**

* **ls:** list files and directories
* **cp:** copy files and directories
* **cat:** concatenate and display file contents
* **mv:** move or rename files and directories
* **grep:** Search for patterns in text within files

**Code:**

import subprocess

# Command 1: ls (List files in a directory)

option1 = subprocess.run(["ls", "-l"], stdout=subprocess.PIPE, text=True)

option2 = subprocess.run(["ls", "-a"], stdout=subprocess.PIPE, text=True)

print("Option 1 - List files in long format:")

print(option1.stdout)

print("Option 2 - List all files (including hidden):")

print(option2.stdout)

# Command 2: cp (Copy a file)

source\_file = "source.txt"

destination\_file = "destination.txt"

subprocess.run(["cp", source\_file, destination\_file])

print(f"File '{source\_file}' copied to '{destination\_file}'")

# Command 3: cat (Display file contents)

file\_to\_display = "source.txt"

option1 = subprocess.run(["cat", file\_to\_display], stdout=subprocess.PIPE, text=True)

option2 = subprocess.run(["cat", "-n", file\_to\_display], stdout=subprocess.PIPE, text=True)

print("Option 1 - Display file contents:")

print(option1.stdout)

print("Option 2 - Display file contents with line numbers:")

print(option2.stdout)

# Command 4: mv (Move or rename a file)

source\_file = "source.txt"

destination\_file = "newfile.txt"

subprocess.run(["mv", source\_file, destination\_file])

print(f"File '{source\_file}' moved/renamed to '{destination\_file}'")

# Command 5: grep (Search for a pattern in a file)

file\_to\_search = "newfile.txt"

pattern = "Hello"

option1 = subprocess.run(["grep", pattern, file\_to\_search], stdout=subprocess.PIPE, text=True)

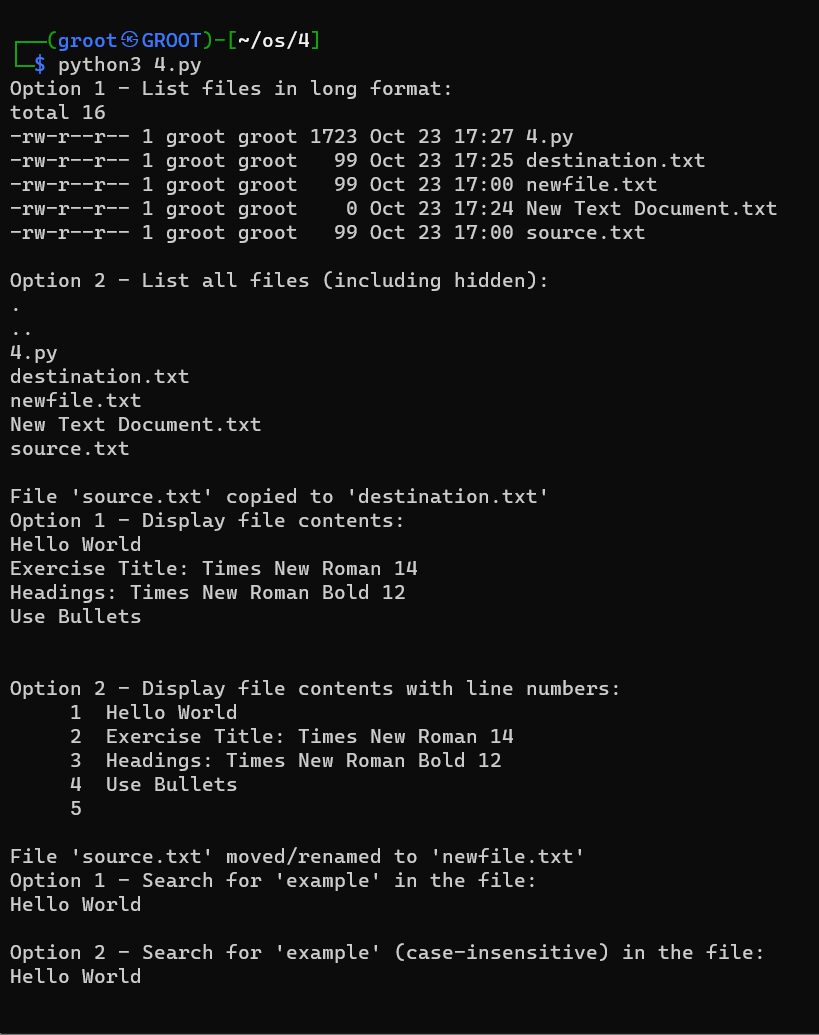
option2 = subprocess.run(["grep", "-i", pattern, file\_to\_search], stdout=subprocess.PIPE, text=True)

print("Option 1 - Search for 'example' in the file:")

print(option1.stdout)

print("Option 2 - Search for 'example' (case-insensitive) in the file:")

print(option2.stdout)

****

**Result:**

Thus, all the Linux commands were simulated using subprocess from a python script.

# Ex No. 5

# Date :

# IMPLEMENTATION OF FCFS CPU SCHEDULING ALGORITHM

# Problem Statement :

# Create a python program to implement FCFS CPU scheduling algorithm.

# Problem Description:

# FCFS is one of the simplest scheduling algorithms used by operating systems to manage the execution of processes. The primary goal is to develop a Python program that simulates the FCFS algorithm to manage a queue of processes and allocate CPU time to each process in the order they arrive.

# Algorithm:

# Create a list or data structure to hold information about each process. Each entry should include:

# Process ID (an identifier for the process).

# Arrival Time (the time at which the process arrives).

# Burst Time (the time required to complete the process).

# Sort Processes by Arrival Time

# Process Execution Loop:

# Iterate through the sorted list of processes.

# For each process:

# If the process has not yet arrived (its arrival time is greater than the current time), wait until it arrives.

# Update the current time to be the maximum of the process's arrival time and the current time (ensuring the current time moves forward).

# Execute the process for its burst time.

# Calculate the turnaround time for the process (turnaround time = completion time - arrival time).

# Calculate the waiting time for the process (waiting time = turnaround time - burst time).

# Add the waiting time to the total waiting time.

# Calculate and Display Metrics:

# Calculate the average waiting time (average waiting time = total waiting time / number of processes).

# Display the turnaround time, waiting time, and average waiting time for each process.

# CODE:

n=int(input("Enter the no.of proceses:"))

at=[]

bt=[]

pid=[]

for i in range(n):

  at.append(int(input(f"Enter the arrival time of processor {i+1}: ")))

  bt.append(int(input(f"Enter the burst time of processor {i+1}: ")))

  pid.append(f"P{i+1}")

print()

print("PID  AT  BT")

for i in range(n):

   print(f"P{i+1}   ", at[i], " ",bt[i])

d={}

for j in range(n):

  d[f"P{j+1}"]=[at[j],bt[j]]

print()

overhead=int(input("Enter the no.of overhead unit: "))

print()

d = sorted(d.items(), key=lambda item: item[1][0])

CT=[]

idle=0

st=""

for i in range(len(d)):

    if(i==0):

       v=d[i][1][1]

       CT.append(v)

       st+=("|"+"\_"\*v+str(d[i][0])+"|")

    elif CT[i-1]<d[i][1][0]:

       v1=CT[i-1] + d[i][1][1]

       idle+=((d[i][1][0]-CT[i-1])+overhead)

       CT.append(idle+ v1)

       st+=("\*"\*idle+"|")

       st+=("\_"\*(d[i][1][1])+str(d[i][0])+"|")

    else:

       v2=(CT[i-1] + d[i][1][1])

       CT.append(v2)

       st+=("\*"\*overhead+"|")

       st+=("\_"\*(d[i][1][1])+str(d[i][0])+"|")

TT = []

for i in range(len(d)):

    TT.append(CT[i] - d[i][1][0])

WT = []

for i in range(len(d)):

    WT.append(TT[i] - d[i][1][1])

AWT = 0

for i in WT:

    AWT +=i

AWT = (AWT/n)

ATT = 0

for i in TT:

    ATT +=i

ATT = (ATT/n)

print("GANTT CHART"+"\n")

print(st+"\n")

print("PID    AT        BT      CT       TT          WT   ")

print("---------------------------------------------------")

for p in pid:

 for i in range(len(d)):

   if p==d[i][0]:

      print(d[i][0],"      ",d[i][1][0],"     ",d[i][1][1],"     ",CT[i],"      ",TT[i],"        ",WT[i],"     ")

print("Average Waiting Time: ",AWT)

print("Average Turnaround Time: ",ATT)

# Output:

A screenshot of a computer

Description automatically generated

A black and white screen with numbers and text

Description automatically generated

A screenshot of a computer program

Description automatically generated

A black screen with white text

Description automatically generated

**Result :**

The FCFS (First-Come-First-Serve) CPU scheduling algorithm has been successfully executed for the provided set of processes. The turnaround times and waiting times for each process have been calculated, and the average waiting time has been determined.

Ex. No: 9

Date:

**IPC USING SEMAPHORES – PRODUCER, CONSUMER PROBLEM**

**Problem Statement:**

Implement a program to solve the Producer-Consumer problem using semaphores and IPC.

**Problem Description:**

The Producer-Consumer problem involves two types of processes, producers and consumers, who share a common, fixed-size buffer as a queue. Producers are responsible for producing items and adding them to the buffer, while consumers retrieve and consume items from the buffer. The challenge is to ensure that producers do not produce when the buffer is full, and consumers do not consume when the buffer is empty. Semaphores are used to synchronize access to the buffer and ensure that producers and consumers work together without conflicts.

**Algorithm:**

1. Initialize semaphores: empty, full, and mutex.
2. Create a shared buffer.
3. Implement the producer function to produce and add items to the buffer.
4. Implement the consumer function to consume items from the buffer.
5. Create producer and consumer threads.
6. Start the threads.
7. Wait for both threads to finish

**Code:**

import threading

import time

# Constants

BUFFER\_SIZE = 5

MAX\_NUMBER = 25

# Semaphores

empty = threading.Semaphore(BUFFER\_SIZE)

full = threading.Semaphore(0)

mutex = threading.Semaphore(1)

# Shared buffer

buffer = []

# Producer function

def producer():

    for item in range(1, MAX\_NUMBER + 1):  # Produce numbers from 1 to 25

        empty.acquire()  # Wait for an empty slot

        mutex.acquire()  # Obtain the mutex to access the buffer

        buffer.append(item)  # Add the item to the buffer

        print(f"Produced: {item}")

        mutex.release()  # Release the mutex

        full.release()  # Signal that the buffer is no longer empty

        time.sleep(0.1)  # Simulate some work

# Consumer function

def consumer():

    for \_ in range(MAX\_NUMBER):  # Consume a total of 25 items

        full.acquire()  # Wait for a full buffer

        mutex.acquire()  # Obtain the mutex to access the buffer

        item = buffer.pop(0)  # Consume the item from the buffer

        print(f"Consumed: {item}")

        mutex.release()  # Release the mutex

        empty.release()  # Signal that there's an empty slot in the buffer

        time.sleep(0.1)  # Simulate some work

# Create producer and consumer threads

producer\_thread = threading.Thread(target=producer)

consumer\_thread = threading.Thread(target=consumer)

# Start the threads

producer\_thread.start()

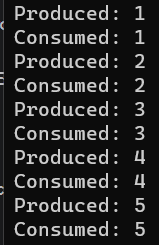
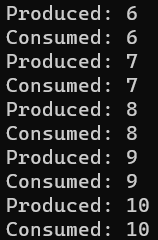
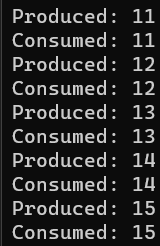
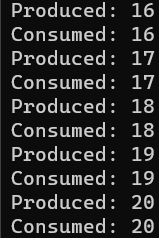
consumer\_thread.start()

# Wait for both threads to finish

producer\_thread.join()

consumer\_thread.join()

**Output:**

**   **

**Result:**

Thus, IPC using semaphores has been implemented successfully to solve producer consumer problem

Ex. No: 10

Date:

**IPC USING PIPES**

**Problem Statement:**

To implement inter process communication using pipes.

**Problem Description:**

To implement IPC using pipes a) one pipe, where parent process writes into the pipe and child reads from the pipe b) two pipes where child 1 reads from parent 2 and child 2 reads from parent 1.

**Algorithm:**

**One pipe with one parent and children**

1. Create a pipe
2. Create a child process pid using fork
3. Check if pid == 0 and close the write end of the pipe
4. Read the data from parent process using read
5. In parent process close the read end of the pipe
6. Write the data to the pipe
7. Wait for child process to finish

**Two pipe with parent and two children**

1. Create two pipes
2. Create a parent pid2 with fork
3. If pid2 = 0. Close the read end of pipe 1 and write end of pipe 2 in parent 2
4. Exit the process without wait
5. Write data to pipe2
6. If pid1 = 0. Close the read end of pipe 2 and write end of pipe 1 in parent 1
7. Write data to pipe 1
8. Exit the process
9. Print the data in the pipes without wait
10. Terminate.

**Code:**

**One pipe with one parent and children**

import os

# Create a pipe

pipe\_read, pipe\_write = os.pipe()

# Create a child process

pid = os.fork()

if pid == 0:

# This is the child process

    os.close(pipe\_write) # Close the write end of the pipe in the child

    child\_data = os.read(pipe\_read, 1024)

    print(f"Child received: {child\_data.decode()}")

else:

# This is the parent process

    os.close(pipe\_read) # Close the read end of the pipe in the parent

    data\_to\_send = "Hello from Parent!"

    os.write(pipe\_write, data\_to\_send.encode())

    os.wait() # Wait for the child process to finish

**Two pipes with two parent and children**

import os

# Create two pipes

pipe1\_read, pipe1\_write = os.pipe()

pipe2\_read, pipe2\_write = os.pipe()

# Create Parent 2

pid2 = os.fork()

if pid2 == 0:

# This is Parent 2

    os.close(pipe1\_read) # Close the read end of Pipe 1 in Parent 2

    os.close(pipe2\_write) # Close the write end of Pipe 2 in Parent 2

    data\_to\_send2 = "Hello from Parent 2 to Child 11!"

    os.write(pipe1\_write, data\_to\_send2.encode())

    os.\_exit(0) # Exit the child process without waiting

else:

# Create Parent 1

    pid1 = os.fork()

if pid1 == 0:

    # This is Parent 1

    os.close(pipe2\_read) # Close the read end of Pipe 2 in Parent 1

    os.close(pipe1\_write) # Close the write end of Pipe 1 in Parent 1

    data\_to\_send1 = "Hello from Parent 1 to Child 21!"

    os.write(pipe2\_write, data\_to\_send1.encode())

    os.\_exit(0) # Exit the child process without waiting

else:

    # This is the main parent process

    # Wait for both child processes to finish

    os.waitpid(pid1, 0)

    os.waitpid(pid2, 0)

    message\_from\_child1 = os.read(pipe1\_read, 1024).decode()

    message\_from\_child2 = os.read(pipe2\_read, 1024).decode()

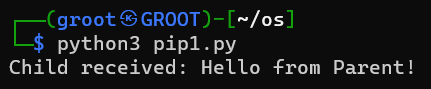
    # Print the messages

    print(message\_from\_child1)

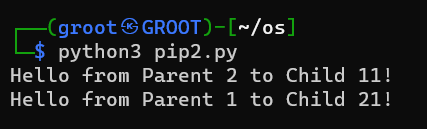
    print(message\_from\_child2)

**Output**

**One pipe with one parent and children**

****

**Two pipes with two parent and children**

****

**Result:**

Thus, inter process communication between processes using pipes has been executed successfully.

Ex. No: 11

Date:

**IPC USING SHARED MEMORY**

**Problem Statement:**

To implement inter process communication using shared memory.

**Problem Description:**

To create a client and server based inter process communication program where server writes A-Z to the shared memory. Client should read the shared memory and display it. When user enters exit in client both client and server should terminate.

**Algorithm:**

1. Create a shared memory with size 100 name sm2
2. Create a buffer shm\_server
3. Display the data in shared memory
4. If data in buffer = exit terminate
5. Write A-Z to the buffer
6. Create a shared memory client instance and buffer
7. Read the data from the shared memory
8. Ask user for input
9. Update the data to the memory
10. If data in buffer = exit terminate
11. Close and unlink the shared memory

**Code:**

**Server**

import time

import multiprocessing.shared\_memory as shared\_memory

# Server code

shm\_server = shared\_memory.SharedMemory(create=True, size=100, name='sm2')

buffer = shm\_server.buf

try:

    while True:

        # Display the data from shared memory

        server\_data = bytes(buffer[:100]).decode('utf-8')

        print("Server data in memory:", server\_data)

        # Check if the data in shared memory equals "exit" and terminate if true

        if buffer[:4] == b'exit':

            shm\_server.close()

            shm\_server.unlink()

            break

        message1 = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

        buffer[:100] = b'\x00' \* 100

        message\_bytes1 = message1.encode('utf-8')

        buffer[:len(message\_bytes1)] = message\_bytes1

        time.sleep(5)

finally:

    # Clean up

    shm\_server.close()

    shm\_server.unlink()

**Client:**

import multiprocessing.shared\_memory as shared\_memory

# Client code

shm\_client = shared\_memory.SharedMemory(name="sm2")

buffer = shm\_client.buf

try:

    while True:

        server\_data = bytes(buffer[:100]).decode('utf-8')

        print("Server says:", server\_data)

        if server\_data.strip() == 'exit':

            # Set the "exit" flag in the shared memory to signal the server to terminate

            buffer[:4] = b'exit'

            break

        user\_input = input("Enter 'exit' to quit: ")

        buffer[:100] = b'\x00' \* 100

        message\_bytes = user\_input.encode('utf-8')

        buffer[:len(message\_bytes)] = message\_bytes

        if user\_input == 'exit':

            # Set the "exit" flag in the shared memory to signal the server to terminate

            buffer[:4] = b'exit'

            shm\_client.close()

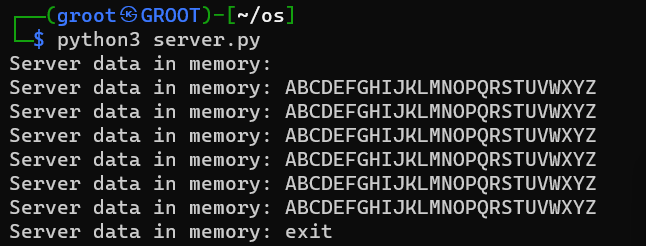
            shm\_client.unlink()

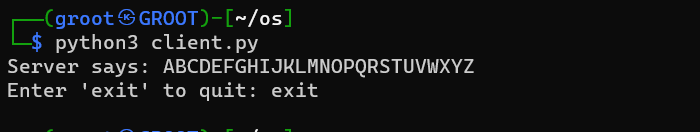
            break

finally:

    shm\_client.close()

**Output:**

****

****

**Result:**

Thus, IPC using shared memory has been implemented successfully.

Ex. No: 12

Date:

**SOLVING DINING PHILOSOPHER'S PROBLEM USING SEMAPHORES**

**Problem Statement:**

To solve dining philosopher’s problem using semaphores

**Problem Description:**

The Dining Philosophers Problem is a classic concurrency challenge where philosophers sit around a circular table, alternating between thinking and eating. To eat, a philosopher must use two adjacent chopsticks. The problem highlights the need for synchronization to prevent conflicts and deadlocks. Philosophers must follow rules: picking up available chopsticks and releasing them when finished. The objective is to devise a solution using semaphores or mutexes to enable philosophers to dine without contention or deadlock, emphasizing the importance of synchronization in concurrent systems.

**Algorithm:**

1. Initialize Semaphore class for chopstick management.
2. Create a list of chopsticks, each represented as a Semaphore.
3. Initialize philosopher\_status list for tracking philosopher states.
4. Define philosopher function to handle philosopher behavior.
5. Implement think function to simulate thinking.
6. Define dine function to manage dining behavior.
7. Use wait and signal to control access to chopsticks.
8. Print philosopher status using print\_status function.
9. In the main section, create philosopher threads and start them.
10. Join all philosopher threads to ensure completion of their tasks.

**Code:**

import threading

import time

class Semaphore:

    def \_\_init\_\_(self, initial\_value=1):

        self.value = initial\_value

        self.lock = threading.Lock()

    def wait(self):

        with self.lock:

            while self.value <= 0:

                pass

            self.value -= 1

    def signal(self):

        with self.lock:

            self.value += 1

NUM\_PHILOSOPHERS = 5

chopsticks = [Semaphore(1) for \_ in range(NUM\_PHILOSOPHERS)]

philosopher\_status = ["thinking"] \* NUM\_PHILOSOPHERS

cycles\_completed = 0

def philosopher(id):

    global cycles\_completed

    while cycles\_completed < 10:

        think(id)

        dine(id)

        cycles\_completed += 1

def think(id):

    philosopher\_status[id] = "thinking"

    print\_status()

    time.sleep(2)

def dine(id):

    left\_chopstick = id

    right\_chopstick = (id + 1) % NUM\_PHILOSOPHERS

    if chopsticks[left\_chopstick].value > 0 and chopsticks[right\_chopstick].value > 0:

        chopsticks[left\_chopstick].wait()

        chopsticks[right\_chopstick].wait()

        philosopher\_status[id] = "eating"

        print\_status()

        time.sleep(1)

        chopsticks[left\_chopstick].signal()

        chopsticks[right\_chopstick].signal()

    else:

        philosopher\_status[id] = "hungry"

        print\_status()

def print\_status():

    for i in range(NUM\_PHILOSOPHERS):

        print(f'Philosopher {i} is {philosopher\_status[i]}')

    print()

if \_\_name\_\_ == "\_\_main\_\_":

    philosophers = []

    for i in range(NUM\_PHILOSOPHERS):

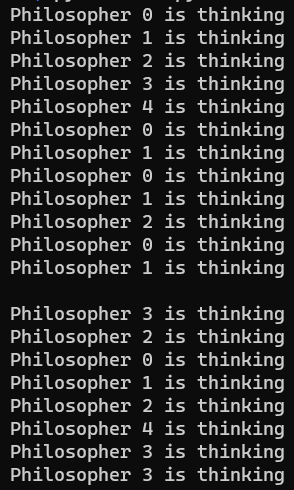
        philosopher\_thread = threading.Thread(target=philosopher, args=(i,))

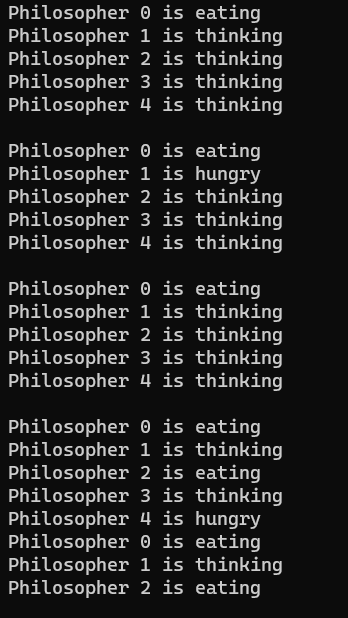
        philosopher\_thread.start()

        philosophers.append(philosopher\_thread)

    for philosopher\_thread in philosophers:

        philosopher\_thread.join()

**Output:**

****

**Result:**

Thus, the dining philosopher problem using semaphores has been implemented successfully.

**Result:**

Thus, the menu driven program using execl() and calling one process from another process using exec() was executed successfully.