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**

```
┌──(groot㉿GROOT)-[~/os]
└─$ ls
1.txt  2.txt  e3.py  p1.py  p2.py  __pycache__

┌──(groot㉿GROOT)-[~/os]
└─$ ls -l
total 24
-rw-r--r-- 1 groot groot   11 Oct 23 16:37 1.txt
-rw-r--r-- 1 groot groot   11 Oct 23 16:40 2.txt
-rw-r--r-- 1 groot groot 1102 Oct 23 16:40 e3.py
-rw-r--r-- 1 groot groot  152 Oct 23 16:50 p1.py
-rw-r--r-- 1 groot groot  109 Oct 23 16:50 p2.py
drwxr-xr-x 2 groot groot 4096 Oct 23 16:51 __pycache__

┌──(groot㉿GROOT)-[~/os]
└─$ ls -a
.  ..  1.txt  2.txt  e3.py  p1.py  p2.py  __pycache__

┌──(groot㉿GROOT)-[~/os]
└─$
```

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

```
┌──(groot㉿GROOT)-[~/os]
└─$ cat 1.txt
Hello World
Exercise Title: Times New Roman 14
Headings: Times New Roman Bold 12
Use Bullets
Contents: Times New Roman 12

┌──(groot㉿GROOT)-[~/os]
└─$ cat -n 1.txt
     1  Hello World
     2  Exercise Title: Times New Roman 14
     3  Headings: Times New Roman Bold 12
     4  Use Bullets
     5  Contents: Times New Roman 12
```

3. ps - Process Status

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

```
groot@GROOT: ~/os

┌──(groot㉿GROOT)-[~/os]
└─$ ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0   2324  1500 ?        Sl   16:22   0:00 /init
root           4  0.1  0.0   2368    68 ?        Sl   16:22   0:02 plan9 --c
root           8  0.0  0.0   2344   108 ?        S    16:22   0:00 /init
root        2846  0.0  0.0   2328   104 ?        Ss   16:35   0:00 /init
root        2847  0.0  0.0   2344   108 ?        S    16:35   0:00 /init
groot       2848  0.0  0.1   7156  3984 pts/2    Ss   16:35   0:00 -bash
root        4820  0.0  0.1  11384  3924 ?        S    16:44   0:00 /usr/sbin
xrdp        4823  0.0  0.0   9560  2320 ?        S    16:44   0:00 /usr/sbin
root        4831  0.0  0.0  11384  3636 ?        S    16:44   0:00 /usr/sbin
groot       4832  0.0  0.1  12520  7316 ?        S    16:44   0:00 xterm
groot       4833  0.0  2.2 330308 81980 ?        Sl   16:44   0:00 /usr/lib/
groot       4836  0.0  0.1  87648  4020 ?        Sl   16:44   0:00 /usr/sbin
```

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

```
┌──(groot㉿GROOT)-[~/os]
└─$ cp 1.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"

```
┌──(groot㉿GROOT)-[~/os]
└─$ echo Hello World
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

```
┌──(groot㉿GROOT)-[~/os]
└─$ cmp 1.txt newfile.txt
1.txt newfile.txt differ: byte 98, line 5
```

7. pwd - Print Working Directory

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

```
┌──(groot㉿GROOT)-[~/os]
└─$ pwd
/home/groot/os
```

8. rm - Remove Files and Directories

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

```
┌──(groot㉿GROOT)-[~/os]
└─$ ls
1.txt  2.txt  e3.py  newfile.txt  p1.py  p2.py  __pycache__

┌──(groot㉿GROOT)-[~/os]
└─$ rm 2.txt

┌──(groot㉿GROOT)-[~/os]
└─$ ls
1.txt  e3.py  newfile.txt  p1.py  p2.py  __pycache__
```

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

```
┌──(groot㉿GROOT)-[~/os]
└─$ ls
1.txt  e3.py  newfile.txt  p1.py  p2.py  __pycache__

┌──(groot㉿GROOT)-[~/os]
└─$ mv 1.txt old.txt

┌──(groot㉿GROOT)-[~/os]
└─$ ls
e3.py  newfile.txt  old.txt  p1.py  p2.py  __pycache__
```

10. touch - Create Empty Files

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

```
┌──(groot㉿GROOT)-[~/os]
└─$ ls
e3.py  newfile.txt  old.txt  p1.py  p2.py  __pycache__

┌──(groot㉿GROOT)-[~/os]
└─$ touch new.txt

┌──(groot㉿GROOT)-[~/os]
└─$ ls
e3.py  newfile.txt  new.txt  old.txt  p1.py  p2.py  __pycache__
```

11. `chmod` - Change File Permissions

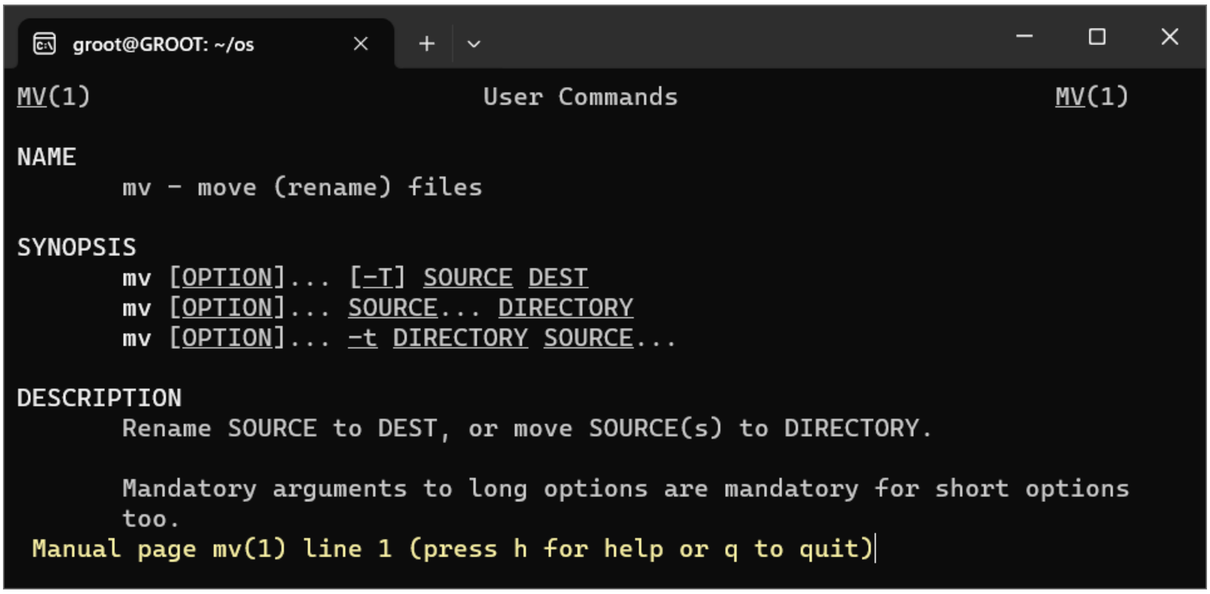
- Description: Modifies file permissions (read, write, execute) for users, groups, and others.
- Synopsis: `chmod [options] mode file`
- Example: chmod 500 newfile.txt

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

```
┌──(groot㉿GROOT)-[~/os]
└─$ more old.txt
Hello World
Exercise Title: Times New Roman 14
Headings: Times New Roman Bold 12
Use Bullets
```

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

```
Hello World
Exercise Title: Times New Roman 14
Headings: Times New Roman Bold 12
Use Bullets

old.txt (END)
```

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

```
┌──(groot㉿GROOT)-[~/os]
└─$ grep "H" old.txt
Hello World
Headings: Times New Roman Bold 12
```

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

```
┌──(groot㉿GROOT)-[~/os]
└─$ head old.txt
Hello World
Exercise Title: Times New Roman 14
Headings: Times New Roman Bold 12
```

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

```
┌──(groot㉿GROOT)-[~/os]
└─$ tail old.txt
Hello World
Exercise Title: Times New Roman 14
Headings: Times New Roman Bold 12
Use Bullets
```

19. sort - Sort Lines in Text Files

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

```
┌──(groot㉿GROOT)-[~/os]
└─$ sort old.txt

Exercise Title: Times New Roman 14
Headings: Times New Roman Bold 12
Hello World
Use Bullets
```

20. whoami - Display Current User

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

```
┌──(groot㉿GROOT)-[~/os]
└─$ whoami
groot
```

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:

```
┌──(groot㉿GROOT)-[~/os]
└─$ python3 2a.py
Parent Process - PID: 5077
Parent Process - Forking a Child Process...
Parent Process - Waiting for the child process to complete...
Child Process - PID: 5078
Child Process - Parent PID: 5077
Child Process - Exiting
Parent Process - Child Process has exited with status 0
```

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:

```
┌──(groot㉿GROOT)-[~/os]
└─$ python3 2b.py
Read data: This is a sample text.
```

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:

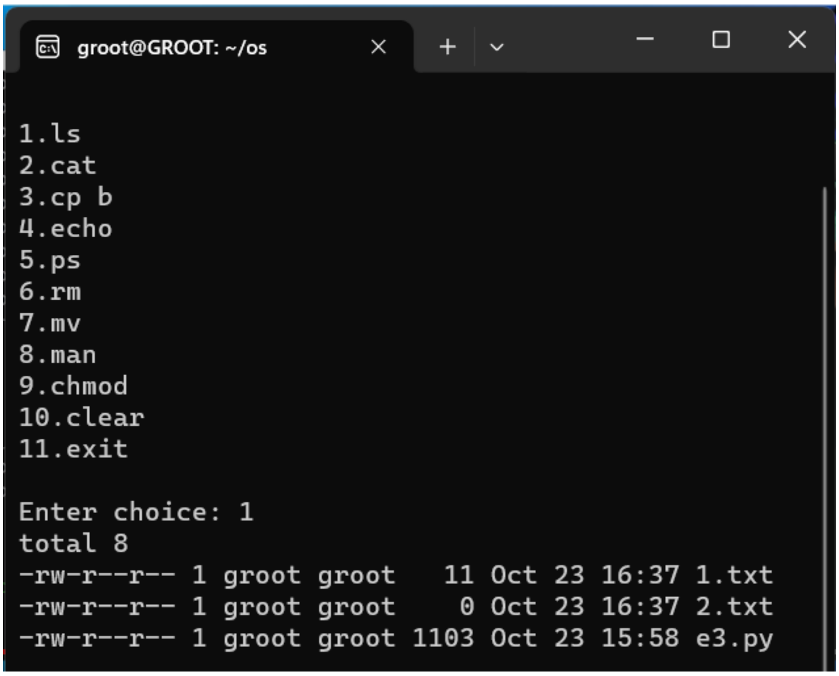
1. **ls:** List files and directories.
2. **cat:** Concatenate and display file contents.
3. **cp:** Copy files and directories.
4. **echo:** Display text.
5. **ps:** List running processes.
6. **rm:** Remove files and directories.
7. **mv:** Move or rename files and directories.
8. **man**: View manual 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','/u
sr/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:



```
groot@GROOT: ~/os

1.ls
2.cat
3.cp b
4.echo
5.ps
6.rm
7.mv
8.man
9.chmod
10.clear
11.exit

Enter choice: 1
total 8
-rw-r--r-- 1 groot groot   11 Oct 23 16:37 1.txt
-rw-r--r-- 1 groot groot    0 Oct 23 16:37 2.txt
-rw-r--r-- 1 groot groot 1103 Oct 23 15:58 e3.py
```

```
1.ls
2.cat
3.cp b
4.echo
5.ps
6.rm
7.mv
8.man
9.chmod
10.clear
11.exit

Enter choice: 2
Hello World
```

```
1.ls
2.cat
3.cp
4.echo
5.ps
6.rm
7.mv
8.man
9.chmod
10.clear
11.exit

Enter choice: 4
Hello World
```

```
1.ls
2.cat
3.cp
4.echo
5.ps
6.rm
7.mv
8.man
9.chmod
10.clear
11.exit

Enter choice: 5
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND

root           1  0.0  0.0   2324  1500 ?        Sl   16:22   0:00 /init
root           4  0.0  0.0   2508    68 ?        Sl   16:22   0:00 plan9 -
-control-socket 5 --log-level 4 --servroot           7  0.0  0.0   2328
104 ?        Ss   16:22   0:00 /init
root           8  0.0  0.0   2344   108 ?        S    16:22   0:00 /init
```

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:

```
Process 1; Input: 2
Process 2; 4
Process 1; Input: 3
Process 2; 9
Process 1; Input: 4
Process 2; 16
Process 1; Input: 5
Process 2; 25
Process 1; Input: 6
Process 2; 36
Process 1; Input: 7
Process 2; 49
Process 1; Input: 8
Process 2; 64
```

Result:

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

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)
```

```
┌──(groot㉿GROOT)-[~/os/4]
└─$ python3 4.py
Option 1 - List files in long format:
total 16
-rw-r--r-- 1 groot groot 1723 Oct 23 17:27 4.py
-rw-r--r-- 1 groot groot   99 Oct 23 17:25 destination.txt
-rw-r--r-- 1 groot groot   99 Oct 23 17:00 newfile.txt
-rw-r--r-- 1 groot groot    0 Oct 23 17:24 New Text Document.txt
-rw-r--r-- 1 groot groot   99 Oct 23 17:00 source.txt

Option 2 - List all files (including hidden):
.
..
4.py
destination.txt
newfile.txt
New Text Document.txt
source.txt

File 'source.txt' copied to 'destination.txt'
Option 1 - Display file contents:
Hello World
Exercise Title: Times New Roman 14
Headings: Times New Roman Bold 12
Use Bullets


Option 2 - Display file contents with line numbers:
     1  Hello World
     2  Exercise Title: Times New Roman 14
     3  Headings: Times New Roman Bold 12
     4  Use Bullets
     5

File 'source.txt' moved/renamed to 'newfile.txt'
Option 1 - Search for 'example' in the file:
Hello World

Option 2 - Search for 'example' (case-insensitive) in the file:
Hello World
```

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:

1. 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).
2. Sort Processes by Arrival Time
3. 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.
4. 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:

```
Enter the no.of proceses:5
Enter the arrival time of processor 1: 4
Enter the burst time of processor 1: 5
Enter the arrival time of processor 2: 6
Enter the burst time of processor 2: 4
Enter the arrival time of processor 3: 0
Enter the burst time of processor 3: 3
Enter the arrival time of processor 4: 6
Enter the burst time of processor 4: 2
Enter the arrival time of processor 5: 5
Enter the burst time of processor 5: 4

PID  AT  BT
P1    4   5
P2    6   4
P3    0   3
P4    6   2
P5    5   4

Enter the no.of overhead unit: 0

GANTT CHART

|___P3|*|_____P1||____P5||____P2||__P4|
```

```
PID    AT        BT      CT       TT           WT
-------------------------------------------------
P1        4       5       9        5           0
P2        6       4       17        11           7
P3        0       3       3        3           0
P4        6       2       19        13           11
P5        5       4       13        8           4
Average Waiting Time:  4.4
Average Turnaround Time:  8.0
```

```
Enter the no.of proceses:6
Enter the arrival time of processor 1: 0
Enter the burst time of processor 1: 3
Enter the arrival time of processor 2: 1
Enter the burst time of processor 2: 2
Enter the arrival time of processor 3: 2
Enter the burst time of processor 3: 1
Enter the arrival time of processor 4: 3
Enter the burst time of processor 4: 4
Enter the arrival time of processor 5: 4
Enter the burst time of processor 5: 5
Enter the arrival time of processor 6: 5
Enter the burst time of processor 6: 2

PID  AT  BT
P1    0   3
P2    1   2
P3    2   1
P4    3   4
P5    4   5
P6    5   2

Enter the no.of overhead unit: 1

GANTT CHART

|___P1|*|__P2|*|_P3|*|____P4|*|_____P5|*|__P6|
```

```
PID    AT        BT       CT        TT          WT
---------------------------------------------------
P1        0       3        3         3          0
P2        1       2        5         4          2
P3        2       1        6         4          3
P4        3       4        10         7          3
P5        4       5        15         11          6
P6        5       2        17         12          10
Average Waiting Time:  4.0
Average Turnaround Time:  6.833333333333333
```

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:

```
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5
```

```
Produced: 6
Consumed: 6
Produced: 7
Consumed: 7
Produced: 8
Consumed: 8
Produced: 9
Consumed: 9
Produced: 10
Consumed: 10
```

```
Produced: 11
Consumed: 11
Produced: 12
Consumed: 12
Produced: 13
Consumed: 13
Produced: 14
Consumed: 14
Produced: 15
Consumed: 15
```

```
Produced: 16
Consumed: 16
Produced: 17
Consumed: 17
Produced: 18
Consumed: 18
Produced: 19
Consumed: 19
Produced: 20
Consumed: 20
```

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

```
┌──(groot㉿GROOT)-[~/os]
└─$ python3 pip1.py
Child received: Hello from Parent!
```

Two pipes with two parent and children

```
┌──(groot㉿GROOT)-[~/os]
└─$ python3 pip2.py
Hello from Parent 2 to Child 11!
Hello from Parent 1 to Child 21!
```

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:

```
┌──(groot㉿GROOT)-[~/os]
└─$ python3 server.py
Server data in memory:
Server data in memory: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Server data in memory: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Server data in memory: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Server data in memory: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Server data in memory: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Server data in memory: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Server data in memory: exit
```

```
┌──(groot㉿GROOT)-[~/os]
└─$ python3 client.py
Server says: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Enter 'exit' to quit: exit
```

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:

```
Philosopher 0 is thinking
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 0 is thinking
Philosopher 1 is thinking
Philosopher 0 is thinking
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 0 is thinking
Philosopher 1 is thinking

Philosopher 3 is thinking
Philosopher 2 is thinking
Philosopher 0 is thinking
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 4 is thinking
Philosopher 3 is thinking
Philosopher 3 is thinking
```

```
Philosopher 0 is eating
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking

Philosopher 0 is eating
Philosopher 1 is hungry
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking

Philosopher 0 is eating
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking

Philosopher 0 is eating
Philosopher 1 is thinking
Philosopher 2 is eating
Philosopher 3 is thinking
Philosopher 4 is hungry
Philosopher 0 is eating
Philosopher 1 is thinking
Philosopher 2 is eating
```

Result:

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