EXPERIMENT – 01

AIM: *TO STUDY AND EXECUTE COMMANDS IN UNIX.*

FILE RELATED COMMANDS:

1. **PWD COMMAND**: Prints the current working directory path.

```
┌──(praveen㉿kali)-[~]
└─$ pwd
/home/praveen
```

2. **CD COMMAND**: Changes the current working directory.

```
┌──(praveen㉿kali)-[~]
└─$ cd Desktop
```

3. **LS COMMAND**: Lists files and directories in the current directory.

```
┌──(praveen㉿kali)-[~/Desktop]
└─$ ls -l
total 16
drwxr-xr-x 2 praveen praveen 4096 Sep 29 10:06 CET
-rwxr-xr-x 1 praveen praveen  558 Oct  4  2023 project_01_information_gathering_tool.py
-rwxr-xr-x 1 praveen praveen 1202 Oct  4  2023 project_02_network_and_port_scanner.py
-rwxr-xr-x 1 praveen praveen  328 Oct  6  2023 project_02_port_scanner.py
```

4. **RM COMMAND**: Removes files from the current directory.

```
┌──(praveen㉿kali)-[~]
└─$ rm short.py
```

5. **MV COMMAND**: Moves or renames files or directories.

```
┌──(praveen㉿kali)-[~]
└─$ mv cbit Desktop

┌──(praveen㉿kali)-[~]
└─$ cd Desktop

┌──(praveen㉿kali)-[~/Desktop]
└─$ ls -l
total 20
drwxr-xr-x 2 praveen praveen 4096 Sep 29 10:06 CET
drwxr-xr-x 2 praveen praveen 4096 Sep 29 10:11 cbit
-rwxr-xr-x 1 praveen praveen  558 Oct  4  2023 project_01_information_gathering_tool.py
-rwxr-xr-x 1 praveen praveen 1202 Oct  4  2023 project_02_network_and_port_scanner.py
-rwxr-xr-x 1 praveen praveen  328 Oct  6  2023 project_02_port_scanner.py
```

6. **CAT COMMAND**: Concatenates and displays the contents of files.

```
┌──(praveen㉿kali)-[~/Desktop]
└─$ cat >hello.txt
Good Mornning

┌──(praveen㉿kali)-[~/Desktop]
└─$ cat hello.txt
Good Mornning

┌──(praveen㉿kali)-[~/Desktop]
└─$ cat >hello.txt
I am a Student

┌──(praveen㉿kali)-[~/Desktop]
└─$ cat hello.txt
I am a Student
```

7. **CMP COMMAND**: Compares two files byte by byte.

```
┌──(praveen㉿kali)-[~/Desktop]
└─$ cmp hello.txt
cmp OS.txt
hello.txt - differ: byte 1, line 1
```

8. **CP COMMAND**: Copies files or directories.

```
┌──(praveen㉿kali)-[~/Desktop]
└─$ cp OS.txt OS1.txt
```

9. **ECHO COMMAND**: Displays a string of text on the terminal.

```
┌──(praveen㉿kali)-[~/Desktop]
└─$ echo "Hello"
Hello
```

10. **MKDIR COMMAND**: Creates a new directory.

```
┌──(praveen㉿kali)-[~/Desktop]
└─$ mkdir sys.py
```

11. **PASTE COMMAND**: Merges lines of multiple files.

```
┌──(praveen㉿kali)-[~/Desktop]
└─$ paste hello.txt OS.txt
I am a Student  OS lab Internal is on Wednesday.
```

12. **RMDIR COMMAND**: Removes empty directories.

```
┌──(praveen㉿kali)-[~/Desktop]
└─$ rmdir sys.py
```

13. **HEAD COMMAND**: Displays the first part of a file.

```
┌──(praveen㉿kali)-[~/Desktop]
└─$ head states.txt
Andhra Pradesh
Arunachal Pradesh
Assam
Bihar
Chhattisgarh
Goa
Gujarat
Haryana
Himachal Pradesh
Jammu and Kashmir
```

14. **TAIL COMMAND**: Displays the last part of a file.

```
┌──(praveen㉿kali)-[~/Desktop]
└─$ tail states.txt
Odisha
Punjab
Rajasthan
Sikkim
Tamil Nadu
Telangana
Tripura
Uttar Pradesh
Uttarakhand
West Bengal
```

15. **DATE COMMAND**: Displays or sets the system date and time.

```
┌──(praveen㉿kali)-[~/Desktop]
└─$ date
Sun Sep 29 11:06:25 IST 2024
```

16. **GREP COMMAND**: Searches for patterns in files.

```
┌──(praveen㉿kali)-[~/Desktop]
└─$ grep "operating" OS1.txt
An operating system brings powerful benefits to computer software and software development. Without an operating system, every applicati
functionality of the underlying computer, such as disk storage, network interfaces and so on. Considering the vast array of underlying ha
 impractical.
As long as each application accesses the same resources and services in the same way, that system software -- the operating system -- ca
d to develop and debug an application, while ensuring that users can control, configure and manage the system hardware through a common a
```

17. **TOUCH COMMAND**: Changes file timestamps or creates an empty file.

```
┌──(praveen㉿kali)-[~/Desktop]
└─$ touch OS1.txt
```

18. **CHMOD COMMAND**: Changes file permissions.

```
┌──(praveen㉿kali)-[~/Desktop]
└─$ chmod 755 OS1.txt
```

19. **MAN COMMAND**: Displays the manual pages for commands.

```
┌──(praveen㉿kali)-[~/Desktop]
└─$ man ls
```

```
LS(1)                                                                  User Commands

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List information about the FILEs (the current directory by default).  Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.

       Mandatory arguments to long options are mandatory for short options too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..

       --author
              with -l, print the author of each file

       -b, --escape
              print C-style escapes for nongraphic characters

       --block-size=SIZE
              with -l, scale sizes by SIZE when printing them; e.g., '--block-size=M'; see SIZE format below

       -B, --ignore-backups
              do not list implied entries ending with ~

       -c     with -lt: sort by, and show, ctime (time of last modification of file status information); with -l: show ctime and sort by name; otherwise: sort by ctime, newest first

       -C     list entries by columns

       --color[=WHEN]
              color the output WHEN; more info below

       -d, --directory
              list directories themselves, not their contents

       -D, --dired
              generate output designed for Emacs' dired mode

       -f     list all entries in directory order

       -F, --classify[=WHEN]
Manual page ls(1) line 1 (press h for help or q to quit)
```

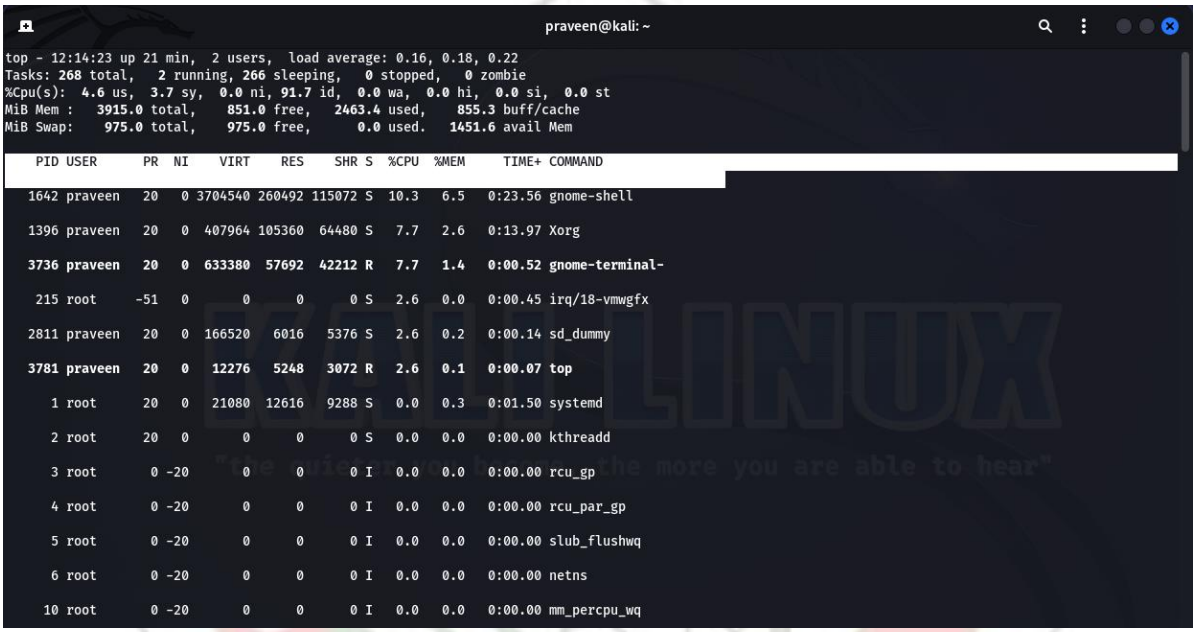
20. **CLEAR COMMAND**: Clears the terminal screen.

```
root        5371       2  0 13:43 ?        00:00:00 [kworker/2:3H-ttm]
root        5372       2  0 13:43 ?        00:00:00 [kworker/2:4H-ttm]
root        5373       2  0 13:44 ?        00:00:00 [kworker/1:2-ata_sff]
praveen     5378    1299  3 13:44 ?        00:00:01 /usr/bin/eog /home/pi
root        5405       2  0 13:44 ?        00:00:00 [kworker/u6:3-events_
praveen     5416    4144  0 13:44 pts/0    00:00:00 ps -ef

┌──(praveen㉿kali)-[~]
└─$ clear
```

PROCESS RELATED COMMANDS:

1. **TOP COMMAND**: Displays a real-time view of active processes, including CPU and memory usage.

```
┌──(praveen㉿kali)-[~]
└─$ top
```



2. **PS COMMAND:** Shows a snapshot of the current processes in the system.

ps: Displays processes running in the current shell.

```
┌──(praveen㉿kali)-[~]
└─$ ps
    PID TTY          TIME CMD
   4144 pts/0    00:00:00 zsh
   4163 pts/0    00:00:00 ps
```

ps -aux: Shows detailed information about all processes for all users.

```
┌──(praveen㉿kali)-[~]
└─$ ps -aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.1  0.3  21080 12616 ?        Ss   11:53   0:01 /sbin/init sp
root           2  0.0  0.0      0     0 ?        S    11:53   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   11:53   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   11:53   0:00 [rcu_par_gp]
root           5  0.0  0.0      0     0 ?        I<   11:53   0:00 [slub_flushwq
root           6  0.0  0.0      0     0 ?        I<   11:53   0:00 [netns]
root          10  0.0  0.0      0     0 ?        I<   11:53   0:00 [mm_percpu_wq
root          11  0.0  0.0      0     0 ?        I    11:53   0:00 [rcu_tasks_kt
root          12  0.0  0.0      0     0 ?        I    11:53   0:00 [rcu_tasks_ru
root          13  0.0  0.0      0     0 ?        I    11:53   0:00 [rcu_tasks_tr
root          14  0.0  0.0      0     0 ?        S    11:53   0:00 [ksoftirqd/0]
root          15  0.0  0.0      0     0 ?        I    11:53   0:00 [rcu_preempt]
root          16  0.0  0.0      0     0 ?        S    11:53   0:00 [migration/0]
root          17  0.0  0.0      0     0 ?        S    11:53   0:00 [idle_inject/
root          19  0.0  0.0      0     0 ?        S    11:53   0:00 [cpuhp/0]
root          20  0.0  0.0      0     0 ?        S    11:53   0:00 [cpuhp/1]
root          21  0.0  0.0      0     0 ?        S    11:53   0:00 [idle_inject/
root          22  0.0  0.0      0     0 ?        S    11:53   0:00 [migration/1]
root          23  0.0  0.0      0     0 ?        S    11:53   0:00 [ksoftirqd/1]
root          26  0.0  0.0      0     0 ?        S    11:53   0:00 [cpuhp/2]
root          27  0.0  0.0      0     0 ?        S    11:53   0:00 [idle_inject/
root          28  0.0  0.0      0     0 ?        S    11:53   0:00 [migration/2]
```

ps -ef: Lists all processes in a detailed format with additional fields.

```
┌──(praveen㉿kali)-[~]
└─$ ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 12:40 ?        00:00:02 /sbin/init splash
root           2       0  0 12:40 ?        00:00:00 [kthreadd]
root           3       2  0 12:40 ?        00:00:00 [rcu_gp]
root           4       2  0 12:40 ?        00:00:00 [rcu_par_gp]
root           5       2  0 12:40 ?        00:00:00 [slub_flushwq]
```

3. **KILL COMMAND**: Sends a signal to terminate or control a process.
   SIGKILL (-9): Forcefully terminates a process without cleanup.

```
┌──(praveen㉿kali)-[~]
└─$ kill -9 1234
```

   SIGTERM (-15): Gracefully terminates a process, allowing cleanup.

```
┌──(praveen㉿kali)-[~]
└─$ kill -15 1432
```

4. **NICE COMMAND:** Starts a process with a modified priority (niceness value), where a higher value reduces its priority.

```
┌──(praveen㉿kali)-[~]
└─$ nice -n 10 project1.py
nice: ‘project1.py’: No such file or directory
```

NETWORK RELATED COMMANDS:

1. **PING COMMAND**: Tests network connectivity by sending ICMP echo requests and measuring response times.

```
┌──(praveen㉿kali)-[~]
└─$ ping facebook.com
PING facebook.com (163.70.140.35) 56(84) bytes of data.
64 bytes from edge-star-mini-shv-01-hyd1.facebook.com (163.70.140.35): icmp_seq=1 ttl=55 time=5.14 ms
64 bytes from edge-star-mini-shv-01-hyd1.facebook.com (163.70.140.35): icmp_seq=2 ttl=55 time=6.13 ms
64 bytes from edge-star-mini-shv-01-hyd1.facebook.com (163.70.140.35): icmp_seq=3 ttl=55 time=9.49 ms
64 bytes from edge-star-mini-shv-01-hyd1.facebook.com (163.70.140.35): icmp_seq=4 ttl=55 time=4.58 ms
64 bytes from edge-star-mini-shv-01-hyd1.facebook.com (163.70.140.35): icmp_seq=5 ttl=55 time=5.72 ms
64 bytes from edge-star-mini-shv-01-hyd1.facebook.com (163.70.140.35): icmp_seq=6 ttl=55 time=4.70 ms
64 bytes from edge-star-mini-shv-01-hyd1.facebook.com (163.70.140.35): icmp_seq=7 ttl=55 time=4.66 ms
64 bytes from edge-star-mini-shv-01-hyd1.facebook.com (163.70.140.35): icmp_seq=8 ttl=55 time=9.09 ms
64 bytes from edge-star-mini-shv-01-hyd1.facebook.com (163.70.140.35): icmp_seq=9 ttl=55 time=20.7 ms
64 bytes from edge-star-mini-shv-01-hyd1.facebook.com (163.70.140.35): icmp_seq=10 ttl=55 time=9.53 ms
64 bytes from edge-star-mini-shv-01-hyd1.facebook.com (163.70.140.35): icmp_seq=11 ttl=55 time=5.39 ms
64 bytes from edge-star-mini-shv-01-hyd1.facebook.com (163.70.140.35): icmp_seq=12 ttl=55 time=7.05 ms
64 bytes from edge-star-mini-shv-01-hyd1.facebook.com (163.70.140.35): icmp_seq=13 ttl=55 time=8.11 ms
64 bytes from edge-star-mini-shv-01-hyd1.facebook.com (163.70.140.35): icmp_seq=14 ttl=55 time=6.01 ms
64 bytes from edge-star-mini-shv-01-hyd1.facebook.com (163.70.140.35): icmp_seq=15 ttl=55 time=6.89 ms
```

2. **IFCONFIG COMMAND**: Displays and configures network interface parameters on a system.

```
┌──(praveen㉿kali)-[~]
└─$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.8  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::a00:27ff:fec7:b004  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:c7:b0:04  txqueuelen 1000  (Ethernet)
        RX packets 23672  bytes 21730678 (20.7 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 13771  bytes 4784402 (4.5 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 26  bytes 1540 (1.5 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 26  bytes 1540 (1.5 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

3. **NETSTAT COMMAND**: Displays active network connections, routing tables, and interface statistics.

```
┌──(praveen㉿kali)-[~]
└─$ netstat -at
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 kali.bbrouter:37510     a23-65-124-99.dep:https ESTABLISHED
tcp        0      0 kali.bbrouter:53800     93.243.107.34.bc.:https ESTABLISHED
tcp        0      0 kali.bbrouter:47482     bom12s21-in-f5.1e:https ESTABLISHED
tcp        0      0 kali.bbrouter:60488     bom12s20-in-f10.1:https ESTABLISHED
```

4. **NSLOOKUP COMMAND**: Queries DNS to retrieve domain name and IP address information.

```
┌──(praveen㉿kali)-[~]
└─$ nslookup www.cbit.ac.in
Server:         192.168.1.1
Address:        192.168.1.1#53

Non-authoritative answer:
Name:   www.cbit.ac.in
Address: 3.111.165.12
```

5. **TELNET COMMAND**: Establishes a connection to a remote server for communication or testing routes.

```
┌──(praveen㉿kali)-[~]
└─$ telnet google.com 80
Trying 142.251.42.46...
Connected to google.com.
Escape character is '^]'.
```

6. **TRACEROUTE COMMAND**: Traces the path packets take to reach a network host, showing each hop.

```
┌──(praveen㉿kali)-[~]
└─$ traceroute facebook.com
traceroute to facebook.com (163.70.140.35), 30 hops max, 60 byte packets
 1  EARTH-1010.bbrouter (192.168.1.1)  7.475 ms  7.430 ms  7.412 ms
 2  103.44.2.93 (103.44.2.93)  7.590 ms  7.698 ms  7.681 ms
 3  * * *
 4  * * *
 5  * * *
 6  ae1.pr02.hyd1.tfbnw.net (157.240.82.174)  9.853 ms  4.302 ms  4.257 ms
 7  po202.asw04.hyd1.tfbnw.net (129.134.96.248)  4.232 ms  4.213 ms po202.asw01.hyd1.tfbnw.net
 8  psw01.hyd1.tfbnw.net (129.134.115.158)  4.178 ms psw04.hyd1.tfbnw.net (129.134.115.155)  4
 9  157.240.38.83 (157.240.38.83)  7.156 ms 173.252.67.165 (173.252.67.165)  7.103 ms 173.252.
```

STUDY ABOUT UNIX VI EDITORS AND ITS FEATURES

DESCRIPTION:

The **vi editor** is a powerful text editor that comes pre-installed on most UNIX and Linux systems. It is used primarily for editing plain text files and is known for being lightweight and efficient. The **vi** editor operates in three different modes, which make it versatile for various text manipulation tasks:

MODES OF VI EDITOR:

1. **Command Mode**: This is the default mode when vi is launched. It allows the user to navigate within the file, delete text, copy and paste, and switch to other modes.
2. **Insert Mode**: In this mode, users can input or edit text. It can be accessed from command mode by pressing i, I, a, A, etc.
3. **Last Line Mode (Ex Mode)**: This mode is used to execute more complex commands, such as saving the file, quitting the editor, searching, and performing other file operations. It is accessed by typing : in command mode.

KEY FEATURES OF VI EDITOR:

1. **Lightweight and Fast**: vi is a very lightweight editor, making it efficient even on older systems.
2. **Cross-Platform**: Works on a wide range of UNIX-based systems and Linux distributions.
3. **No GUI Required**: vi is a terminal-based editor, making it useful for servers or environments without a graphical user interface (GUI).
4. **Syntax Highlighting**: In modern versions, such as vim (an extended version of vi), syntax highlighting is available for easier code editing.
5. **Search and Replace**: vi offers powerful search (/searchterm) and replace (:%s/old/new/g) capabilities across files.
6. **Multiple Buffers**: vi supports editing multiple files simultaneously by using buffers, allowing users to switch between them.

EXPERIMENT – 02

AIM: *DEMONSTARTION OF FILE RELATED SYSTEM CALLS SUCH AS CREATE, OPEN, CLOSE, READ, WRITE, LSEEK.*

DESCRIPTION:

1. Create: Creates a new file with the specified name. If the file exists, it will be truncated.

- **Syntax**: int creat(const char *pathname, mode_t mode);
- **Parameters**:
  - pathname: The name (path) of the file to be created.
  - mode: The file permissions (e.g., read, write, execute) for the new file.
- **Return**: Returns a file descriptor on success or -1 on failure.

2. Open: Opens an existing file and returns a file descriptor for subsequent operations.

- **Syntax**: int open(const char *pathname, int flags, mode_t mode);
- **Parameters**:
  - pathname: The name (path) of the file to be opened.
  - flags: Specifies how the file is to be opened (e.g., O_RDONLY for read-only, O_WRONLY for write-only).
  - mode: File permissions (used only when creating a new file).
- **Return**: Returns a file descriptor on success or -1 on failure.

3. Close: Closes an open file descriptor, freeing associated system resources.

- **Syntax**: int close(int fd);
- **Parameters**:
  - fd: The file descriptor of the file to close.
- **Return**: Returns 0 on success or -1 on failure.

4. Read: Reads data from a file descriptor into a buffer in memory.

- **Syntax**: ssize_t read(int fd, void *buf, size_t count);
- **Parameters**:
  - fd: The file descriptor from which to read.
  - buf: A buffer to store the data being read.
  - count: The number of bytes to read.
- **Return**: Returns the number of bytes read or -1 on failure.

5. Write: Writes data from a buffer in memory to a file.

- **Syntax**: ssize_t write(int fd, const void *buf, size_t count);
- **Parameters**:
  - fd: The file descriptor to which to write.
  - buf: A buffer containing the data to be written.
  - count: The number of bytes to write.
- **Return**: Returns the number of bytes written or -1 on failure.

6. Lseek: Changes the current file offset (position) for reading or writing.

- **Syntax**: off_t lseek(int fd, off_t offset, int whence);
- **Parameters**:
  - fd: The file descriptor of the file to reposition.
  - offset: The new position relative to the location specified by whence.
  - whence: Specifies the reference point for the offset (e.g., SEEK_SET, SEEK_CUR, SEEK_END).
- **Return**: Returns the new file offset or -1 on failure.

PROGRAMS:

Program 1: WAP to copy the content from one file to another withoutdeleting the content of second file.

CODE:

program1.c

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

void main() {
    char b[100];

    int fd = open("file1.txt", O_RDONLY);
    int fd1 = open("file3.txt", O_WRONLY | O_APPEND);

    int n = read(fd, b, 100);

    write(fd1, b, n);

    close(fd);
    close(fd1);
}
```

OUTPUT:

file1.txt

```
Hello, CBITians..
```

file3.txt

```
We are Legends
Hello, CBITians..
```

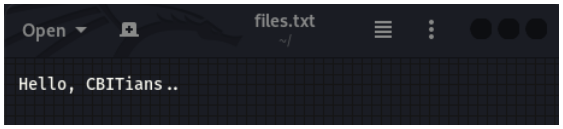
Program 2: WAP to read from one file and write into another file.

CODE:

program2.c

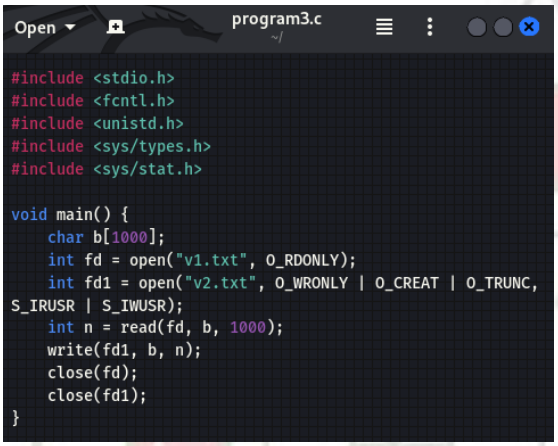
```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

void main() {
    char b[100];
    int fd = open("file1.txt", O_RDONLY);
    int fd1 = open("files.txt", O_WRONLY | O_CREAT |
O_TRUNC, S_IRUSR | S_IWUSR);
    int n = read(fd, b, 100);
    write(fd1, b, n);
    close(fd);
    close(fd1);
}
```

OUTPUT:

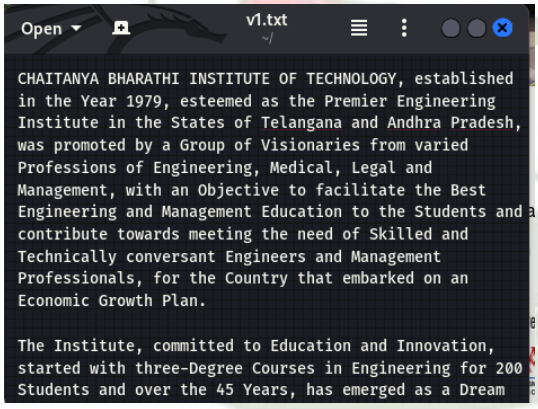


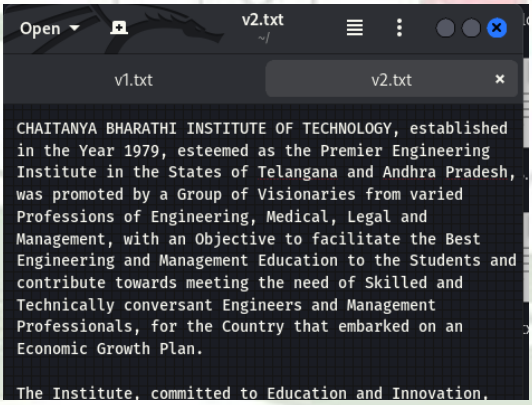
Program 3: WAP to copy the entire content from one file to another (here,content is more than 100 characters).

CODE:



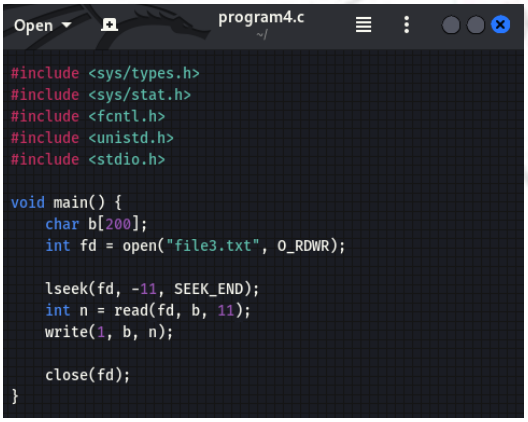
OUTPUT:



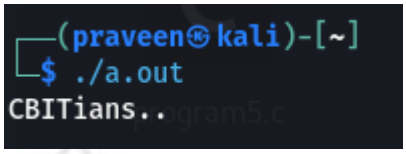


Program 4: WAP to print last 5 characters of a file onto the terminal screen.

CODE:

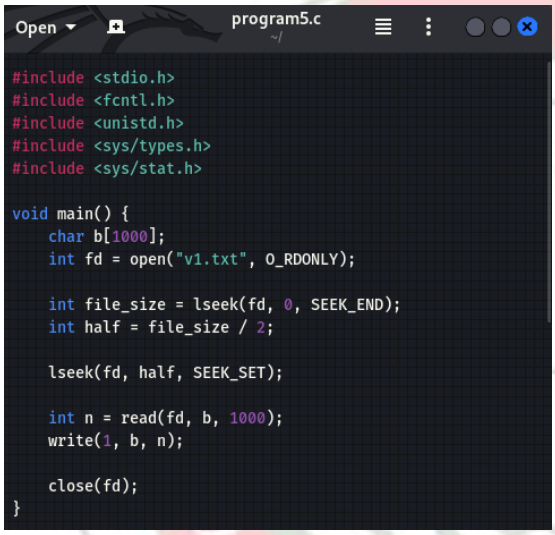


OUTPUT:

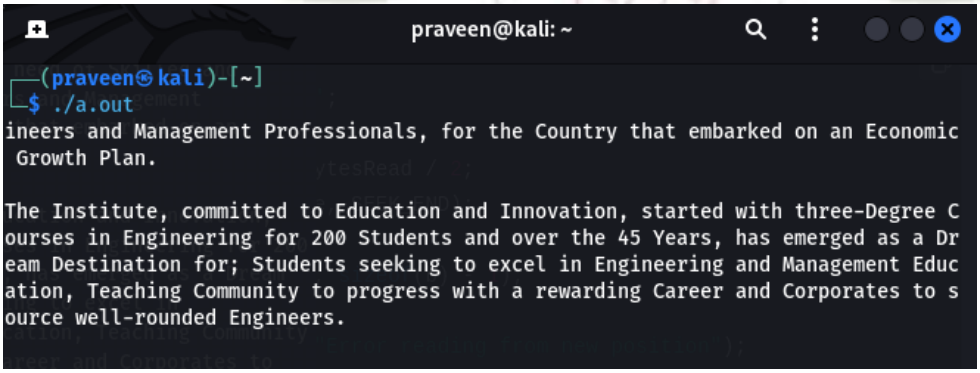


Program 5: WAP to print second half of the file.

CODE:



OUTPUT:



Program 6: WAP to print characters from 10th to 20th position of a file.

CODE:

program6.c

```c
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

void main() {
    char b[11];
    int fd = open("v1.txt", O_RDONLY);

    lseek(fd, 9, SEEK_SET);

    int n = read(fd, b, 10);
    write(1, b, n);

    close(fd);
}
```

OUTPUT:

v1.txt

/home/praveen/v1.txt

CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY, es[illegible]r 1979, esteemed as the Premier Engineering Institute in the States of Telangana and Andhra Pradesh, was promoted by a Group of Visionaries from varied Professions of Engineering, Medical, Legal and Management, with an Objective to facilitate the Best Engineering and Management Education to the Students and contribute towards meeting the need of Skilled and Technically conversant Engineers and Management Professionals, for the Country that embarked on an Economic Growth Plan.

The Institute, committed to Education and Innovation, started with three-Degree Courses in Engineering for 200 Students and over the 45 Years, has emerged as a Dream Destination for; Students seeking to excel in Engineering and Management Education, Teaching Community to progress with a rewarding Career and Corporates to source well-rounded Engineers.

```
┌──(praveen㉿kali)-[~]
└─$ cc program6.c
┌──(praveen㉿kali)-[~]
└─$ ./a.out
BHARATHI
```

EXPERIMENT – 03

AIM: *DEMONSTARTION OF PROCESS RELATED SYSTEM CALLS SUCH AS FORK, GETPID, GETPPID, WAIT, EXEC etc.*

DESCRIPTION:

Process-related system calls in Linux manage process creation, control, andsynchronization:

1. fork()

- **Header Files**: #include <unistd.h>
- **Syntax**: pid_t fork(void);
- **Explanation**: Creates a new process by duplicating the calling process. The new process (child) gets a unique process ID, sharing most of the parent's resources. It returns 0 to the child process and the child's PID to the parent. On failure, it returns -1.

2. getpid()

- **Header Files**: #include <unistd.h>
- **Syntax**: pid_t getpid(void);
- **Explanation**: Returns the process ID (PID) of the calling process. Useful for identifying the process, especially in multi-process applications.

3. getppid()

- **Header Files**: #include <unistd.h>
- **Syntax**: pid_t getppid(void);
- **Explanation**: Returns the parent process ID (PPID) of the calling process, useful for tracking process lineage.

4. exec() Family

- **Header Files**: #include <unistd.h>
- **Syntax**: int execve(const char *pathname, char *const argv[], char *const envp[]);
- **Explanation**: Replaces the current process image with a new process image specified by the pathname. The argv argument holds pointers to the command's arguments, and envp holds pointers to environment variables. If successful, exec does not return; otherwise, it returns -1.

5. wait()

- **Header Files**: #include <sys/types.h>, #include <sys/wait.h>
- **Syntax**: pid_t wait(int *status);
- **Explanation**: Suspends execution of the calling process until one of its child processes terminates. It returns the PID of the terminated child and stores the child's termination status in the variable pointed to by status. Returns -1 if no child processes exist.

1. Program to demonstrate fork system call

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    int a = 2;
    pid_t pid;
    pid = fork();

    printf("%d\n", pid);

    if (pid < 0) {
        printf("FORK FAILED\n");
    } else if (pid == 0) {
        printf("CHILD PROCESS \t a is: ");
        printf("%d\n", ++a);
    } else {
        printf("PARENT PROCESS \t a is: ");
        printf("%d\n", --a);
    }

    printf("EXITING WITH X = %d\n", a);

    return 0;
}
```

OUTPUT:

```
9675
PARENT PROCESS    a is: 1
EXITING WITH X = 1
0
CHILD PROCESS     a is: 3
EXITING WITH X = 3
```

2. Program to demonstrate getpid(),getppid() system calls

Without wait()

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    int a = 2;
    pid_t pid;
    pid = fork();

    printf("%d\n", pid);

    if (pid < 0) {
        printf("FORK FAILED\n");
    } else if (pid == 0) {
        printf("CHILD PROCESS \t a is: %d\n", ++a);
        printf("I AM THE CHILD AND MY PROCESS ID IS %d\n", getpid());
        printf("I AM THE CHILD AND MY PARENT'S PROCESS ID IS %d\n", getppid());
    } else {
        printf("PARENT PROCESS \t a is: %d\n", --a);
        printf("I AM THE PARENT AND MY PROCESS ID IS %d\n", getpid());
        printf("I AM THE PARENT AND MY CHILD'S PROCESS ID IS %d\n", pid);
    }

    printf("EXITING WITH X = %d\n", a);

    return 0;
}
```

OUTPUT:

```
9785
PARENT PROCESS    a is: 1
I AM THE PARENT AND MY PROCESS ID IS 9784
I AM THE PARENT AND MY CHILD'S PROCESS ID IS 9785
EXITING WITH X = 1
0
CHILD PROCESS     a is: 3
I AM THE CHILD AND MY PROCESS ID IS 9785
I AM THE CHILD AND MY PARENT'S PROCESS ID IS 1299
EXITING WITH X = 3
```

With Wait()

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();

    if (pid < 0) { // Error in creating a child
        printf("Child cannot be created\n");
    }
    else if (pid > 0) { // Parent section
        printf("Process id of child = %d\n", pid);
        printf("Process id of parent = %d\n", getpid());

        int status;
        pid_t child_pid = wait(&status); // Parent waits for the child

        if (child_pid > 0) {
            if (WIFEXITED(status)) {
                printf("Child process (PID: %d) exited with status %d\n", child_pid, WEXITSTATUS(status));
            } else {
                printf("Child process (PID: %d) did not terminate normally\n", child_pid);
            }
        } else {
            perror("wait failed");
        }
    }
    else { // Child section
        sleep(1); // Simulate some work
        printf("Process id of child = %d\n", getpid());
        printf("Process id of parent = %d\n", getppid());
    }

    return 6;
}
```

OUTPUT:

```
Process id of child = 9850
Process id of parent = 9849
Process id of child = 9850
Process id of parent = 9849
Child process (PID: 9850) exited with status 6
```

3. Write a Program to demonstrate the concept of Orphan and Zombie Process

ORPHAN CODE:

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();

    if (pid < 0) {  // Error in creating a child
        printf("Child cannot be created\n");
    }
    else if (pid > 0) {  // Parent section
        printf("Process id of child = %d\n", pid);
        printf("Process id of parent = %d\n", getpid());
    }
    else {  // Child section
        sleep(1);  // Simulate delay to show the child becoming an orphan
        printf("Process id of child = %d\n", getpid());
        printf("Process id of parent = %d\n", getppid());  // Shows parent's process ID
    }

    return 0;
}
```

Sleep():

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();

    if (pid < 0) {  // Error in creating a child
        printf("Child cannot be created\n");
    }
    else if (pid > 0) {  // Parent section
        sleep(1);  // Sleep to ensure child doesn't become orphan
        printf("Process id of child = %d\n", pid);  // pid holds child's PID
        printf("Process id of parent = %d\n", getpid());
    }
    else {  // Child section
        printf("Process id of child = %d\n", getpid());
        printf("Process id of parent = %d\n", getppid());
    }

    return 0;
}
```

OUTPUT:

```
┌──(praveen㉿kali)-[~]
└─$ ./a.out
Process id of child = 9915
Process id of parent = 9914

┌──(praveen㉿kali)-[~]
└─$ Process id of child = 9915
Process id of parent = 1299
```

OUTPUT:

```
Process id of child = 10012
Process id of parent = 10011
Process id of child = 10012
Process id of parent = 10011
```

4. Program to demonstrate exec System Call

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
    int pid = fork();

    if (pid < 0) {
        printf("Fork failed\n");
        return 1;
    }
    else if (pid == 0) {
        execl("/bin/ls", "ls", NULL);
        perror("execl failed");
        exit(1);
    }
    else {
        wait(NULL);
        printf("Child process complete\n");
    }

    return 0;
}
```

OUTPUT:

```
Desktop    Public     file1.txt  getpid_withwait.c  program4.c  v2.txt
Documents  Templates  file3.txt  orphan.c           program5.c
Downloads  Videos     files.txt  program1.c         program6.c
Music      a.out      fork.c     program2.c         sleep.c
Pictures   exec.c     getpid.c   program3.c         v1.txt
Child process complete
```

EXPERIMENT – 04

AIM: *Implement CPU Scheduling Algorithms (a) Round Robin (b) SJF (c) FCFS.*

DESCRIPTION:

Round Robin (RR) scheduling allocates CPU time to each process in acircular order. Each process gets a fixed time quantum to execute beforebeing moved to the back of the queue. This method ensures fairness andresponsiveness but can lead to higher average turnaround times if the quantum is not well-tuned.
Shortest Job First (SJF) scheduling selects the process with the smallestexecution time for the CPU next. It can be either preemptive (Shortest Remaining Time First) or non-preemptive. SJF minimizes average waitingtime and turnaround time but may lead to process starvation if short jobs keep arriving.
First Come First Serve (FCFS) scheduling processes tasks in the order they arrive. The first process in the queue is executed to completion beforemoving to the next. While simple and easy to implement, FCFS can lead tothe "convoy effect," where longer processes delay the execution of shorter ones.

ALGORITHM:

1. FCFS Scheduling

1. Input the processes along with their burst time (bt).
2. Find waiting time (wt) for all processes.
3. As first process that comes need not to wait so
   a. waiting time for process 1 will be 0 i.e. wt[0] = 0.
4. Find waiting time for all other processes i.e. for
   a. process i -> wt[i] = bt[i-1] + wt[i-1] .
5. Find turnaround time = waiting_time + burst_time for all processes.
6. Find average waiting time = total_waiting_time / no_of_processes.
7. Similarly, find average turnaround time = total_turn_around_time / no_of_processes.

2. SJF Scheduling

1. Sort all processes according to the arrival time.
2. Select the process that has the minimum arrival time and minimum burst time.
3. After the completion of the process, create a pool of processes that arrive afterward until the completion of the previous process.
4. Select the process among the pool that has the minimum burst time.

5. Completion Time: Time at which the process completes its execution.
6. Turnaround Time: The time difference between completion time and arrival time.
    - Turnaround Time=Completion Time−Arrival Time
      Time}Turnaround Time=Completion Time−Arrival Time
7. Waiting Time (W.T): The time difference between turnaround time and burst time.
    - Waiting Time=Turnaround Time−Burst Time

3. RR Scheduling

1. Initialize the queue.
2. Set the time quantum.
3. Enqueue all processes with their burst times.
4. While the queue is not empty:
    - current_process = Dequeue()
    - If burst_time[current_process] <= time_quantum:
        - Execute current_process for burst_time[current_process].
        - Remove current_process from the queue.
    - Else:
        - Execute current_process for time_quantum.
        - Update burst_time[current_process] = burst_time[current_process] – time_quantum
    - Enqueue current_process at the end of the queue.
5. End while.

PROGRAMS:

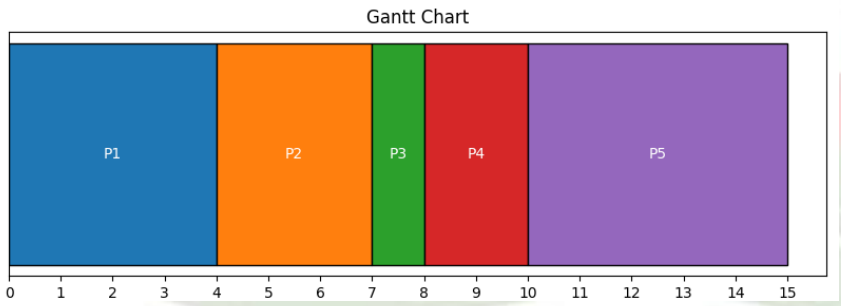
1. FCFS

```
import matplotlib.pyplot as plt

print("FIRST COME FIRST SERVE SCHEDULING")
n = int(input("Enter number of processes: "))
processes = {}

for i in range(n):
    key = "P" + str(i + 1)
    arrival_time = int(input(f"Enter arrival time of process {key}: "))
    burst_time = int(input(f"Enter burst time of process {key}: "))
    processes[key] = [arrival_time, burst_time]

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

ET = []
TAT = []
WT = []

for i in range(len(processes)):
    if i == 0:
        ET.append(processes[list(processes.keys())[i]][0] + processes[list(processes.keys())[i]][1])
    else:
        ET.append(ET[i - 1] + processes[list(processes.keys())[i]][1])

TAT = [ET[i] - processes[list(processes.keys())[i]][0] for i in range(len(processes))]
WT = [TAT[i] - processes[list(processes.keys())[i]][1] for i in range(len(processes))]

RT = []
for i in range(len(processes)):
    if i == 0:
        RT.append(0)
    else:
        RT.append(ET[i - 1] - processes[list(processes.keys())[i]][0])

avg_WT = sum(WT) / n
avg_TAT = sum(TAT) / n
total_time = ET[-1]
throughput = n / total_time

print("Process | Arrival | Burst | Exit | Turn Around | Wait | Response")
for i in range(n):
    process_key = list(processes.keys())[i]
    print(f"  {process_key}     |    {processes[process_key][0]}     |  {processes[process_key][1]}  | {ET[i]}  |   {TAT[i]}      | {WT[i]}  |   {RT[i]}")
```

```
print("Average Waiting Time:", avg_WT)
print("Average Turnaround Time:", avg_TAT)
print("Throughput:", throughput, "processes/unit time")

plt.figure(figsize=(10, 3))
current_time = 0
for i in range(n):
    plt.barh(y=0, width=processes[list(processes.keys())[i]][1], left=current_time, edgecolor='black')
    plt.text(x=current_time + processes[list(processes.keys())[i]][1] / 2,
             y=0,
             s=list(processes.keys())[i],
             ha='center',
             va='center',
             color='white')
    current_time += processes[list(processes.keys())[i]][1]

plt.yticks([])
plt.xticks(range(current_time + 1))
plt.xlabel("Time")
plt.title("Gantt Chart")
plt.show()
```

OUTPUT:

```
FIRST COME FIRST SERVE SCHEDULING
Enter number of processes: 5
Enter arrival time of process P1: 0
Enter burst time of process P1: 4
Enter arrival time of process P2: 1
Enter burst time of process P2: 3
Enter arrival time of process P3: 2
Enter burst time of process P3: 1
Enter arrival time of process P4: 3
Enter burst time of process P4: 2
Enter arrival time of process P5: 4
Enter burst time of process P5: 5
Process | Arrival | Burst | Exit | Turn Around | Wait | Response
  P1    |    0    |  4    | 4    |    4        | 0    |    0
  P2    |    1    |  3    | 7    |    6        | 3    |    3
  P3    |    2    |  1    | 8    |    6        | 5    |    5
  P4    |    3    |  2    | 10   |    7        | 5    |    5
  P5    |    4    |  5    | 15   |    11       | 6    |    6
Average Waiting Time: 3.8
Average Turnaround Time: 6.8
Throughput: 0.3333333333333333 processes/unit time
```



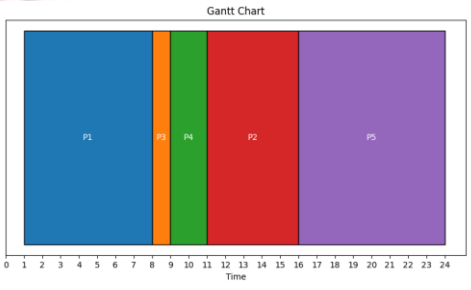
2. SJF

```
import matplotlib.pyplot as plt

def sjf_scheduling():
    print("SHORTEST JOB FIRST SCHEDULING")
    n = int(input("Enter number of processes: "))
    processes = []

    for i in range(n):
        process_id = i + 1
        arrival_time = int(input(f"Enter arrival time of process P{process_id}: "))
        burst_time = int(input(f"Enter burst time of process P{process_id}: "))
        processes.append([process_id, arrival_time, burst_time])

    processes.sort(key=lambda x: x[1])
    waiting_time = [0] * n
    turnaround_time = [0] * n
    completion_time = [0] * n
    response_time = [0] * n

    total_waiting_time = 0
    total_turnaround_time = 0
    total_response_time = 0
    current_time = 0
    completed_processes = 0
    completed = [False] * n
    gantt_chart = []
    gantt_times = []
```

```
    while completed_processes < n:
        available_processes = [p for p in processes if p[1] <= current_time and not completed[p[0] - 1]]

        if available_processes:
            available_processes.sort(key=lambda x: x[2])
            process = available_processes[0]
            process_id = process[0]
            arrival_time = process[1]
            burst_time = process[2]

            if response_time[process_id - 1] == 0:
                response_time[process_id - 1] = current_time - arrival_time
            start_time = current_time
            completion_time[process_id - 1] = start_time + burst_time
            waiting_time[process_id - 1] = start_time - arrival_time
            turnaround_time[process_id - 1] = waiting_time[process_id - 1] + burst_time
            current_time += burst_time
            completed_processes += 1
            completed[process_id - 1] = True
            gantt_chart.append(f"P{process_id}")
            gantt_times.append((start_time, current_time))
        else:
            gantt_chart.append("IDLE")
            gantt_times.append((current_time, current_time + 1))
            current_time += 1

    avg_waiting_time = sum(waiting_time) / n
    avg_turnaround_time = sum(turnaround_time) / n
    avg_response_time = sum(response_time) / n
    total_time = completion_time[-1]
    throughput = n / total_time if total_time > 0 else 0

    print("\nProcess | Arrival Time | Burst Time | Waiting Time | Turnaround Time")
    for i in range(n):
        print(f" P{processes[i][0]} | {processes[i][1]} | {processes[i][2]} | {waiting_time[i]} | {turnaround_time[i]}")

    print(f"\nAverage Waiting Time: {avg_waiting_time:.2f}")
    print(f"Average Turnaround Time: {avg_turnaround_time:.2f}")
    print(f"Average Response Time: {avg_response_time:.2f}")
    print(f"Throughput: {throughput:.2f} processes/unit time")
    print("\nGantt Chart:")

    plt.figure(figsize=(10, 3))
    for i in range(len(gantt_chart)):
        if gantt_chart[i] != "IDLE":
            plt.barh(y=0, width=gantt_times[i][1] - gantt_times[i][0], left=gantt_times[i][0], edgecolor='black')
            plt.text(x=gantt_times[i][0] + (gantt_times[i][1] - gantt_times[i][0]) / 2,
                     y=0, s=gantt_chart[i], ha='center', va='center', color='white')

    plt.yticks([])
    plt.xticks(range(int(current_time) + 1))
    plt.xlabel("Time")
    plt.title("Gantt Chart")
    plt.show()

sjf_scheduling()
```

OUTPUT:

```
SHORTEST JOB FIRST SCHEDULING
Enter number of processes: 5
Enter arrival time of process P1: 1
Enter burst time of process P1: 7
Enter arrival time of process P2: 2
Enter burst time of process P2: 5
Enter arrival time of process P3: 3
Enter burst time of process P3: 1
Enter arrival time of process P4: 4
Enter burst time of process P4: 2
Enter arrival time of process P5: 5
Enter burst time of process P5: 8

Process | Arrival Time | Burst Time | Waiting Time | Turnaround Time
 P1 | 1 | 7 | 0 | 7
 P2 | 2 | 5 | 9 | 14
 P3 | 3 | 1 | 5 | 6
 P4 | 4 | 2 | 5 | 7
 P5 | 5 | 8 | 11 | 19

Average Waiting Time: 6.00
Average Turnaround Time: 10.60
Average Response Time: 6.00
Throughput: 0.21 processes/unit time
```

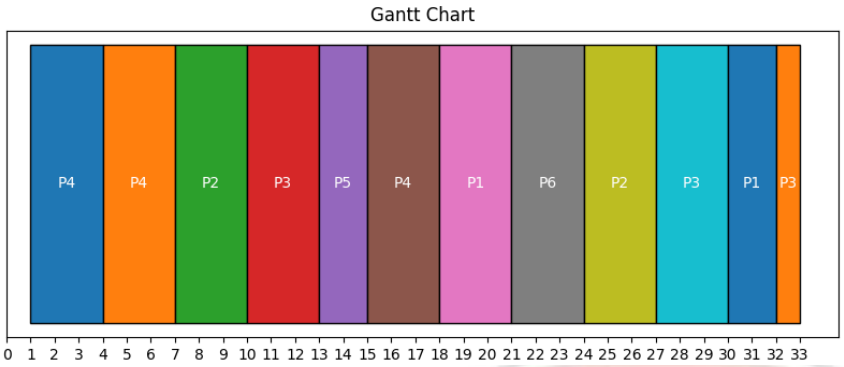


3. RR

```
import matplotlib.pyplot as plt
from collections import deque

def round_robin(processes, burst_time, arrival_time, quantum):
    n = len(processes)
    remaining_burst_time = burst_time[:]
    waiting_time = [0] * n
    turnaround_time = [0] * n
    response_time = [0] * n
    gantt_chart = []
    t = 0
    process_queue = deque()
    process_start_time = [-1] * n

    while True:
        for i in range(n):
            if arrival_time[i] <= t and remaining_burst_time[i] > 0 and i not in process_queue:
                process_queue.append(i)

        if process_queue:
            i = process_queue.popleft()
            if process_start_time[i] == -1:
                process_start_time[i] = t
            start_time = t
            burst = min(quantum, remaining_burst_time[i])
            t += burst
            remaining_burst_time[i] -= burst

            if remaining_burst_time[i] == 0:
                waiting_time[i] = t - burst_time[i] - arrival_time[i]
                turnaround_time[i] = t - arrival_time[i]
            else:
                process_queue.append(i)
            gantt_chart.append((processes[i], start_time, t))
        else:
            next_arrival = min([arrival_time[i] for i in range(n) if arrival_time[i] > t], default=None)
            if next_arrival is not None:
                t = next_arrival
            else:
                break

    avg_waiting_time = sum(waiting_time) / n
    avg_turnaround_time = sum(turnaround_time) / n
    avg_response_time = sum(response_time) / n
    total_time = max(t, max([arrival_time[i] + burst_time[i] for i in range(n)]))
    throughput = n / total_time if total_time > 0 else 0

    print("Gantt Chart:")
    plt.figure(figsize=(10, 3))
    for process, start, end in gantt_chart:
        plt.barh(y=0, width=end - start, left=start, edgecolor='black')
        plt.text(x=start + (end - start) / 2, y=0, s=f"P{process}", ha='center', va='center', color='white')

    plt.yticks([])
    plt.xticks(range(int(t) + 1))
    plt.xlabel("Time")
    plt.title("Gantt Chart")
    plt.show()

    print("\nProcess\tArrival Time\tBurst Time\tWaiting Time\tTurnaround Time\tResponse Time")
    for i in range(n):
        print(f"P{processes[i]}\t{arrival_time[i]}\t\t{burst_time[i]}\t\t{waiting_time[i]}\t\t{turnaround_time[i]}\t\t{response_time[i]}")

    print(f"\nAverage Waiting Time: {avg_waiting_time:.2f}")
    print(f"Average Turnaround Time: {avg_turnaround_time:.2f}")
    print(f"Average Response Time: {avg_response_time:.2f}")
    print(f"Throughput: {throughput:.2f} processes/unit time")

n = int(input("Enter the number of processes: "))
processes = []
burst_time = []
arrival_time = []

for i in range(n):
    process_id = i + 1
    processes.append(process_id)
    bt = int(input(f"Enter the burst time for process P{process_id}: "))
    burst_time.append(bt)
    at = int(input(f"Enter the arrival time for process P{process_id}: "))
    arrival_time.append(at)

quantum = int(input("Enter the time quantum: "))
round_robin(processes, burst_time, arrival_time, quantum)
```

OUTPUT:

```
Enter the number of processes: 6
Enter the burst time for process P1: 5
Enter the arrival time for process P1: 5
Enter the burst time for process P2: 6
Enter the arrival time for process P2: 4
Enter the burst time for process P3: 7
Enter the arrival time for process P3: 3
Enter the burst time for process P4: 9
Enter the arrival time for process P4: 1
Enter the burst time for process P5: 2
Enter the arrival time for process P5: 2
Enter the burst time for process P6: 3
Enter the arrival time for process P6: 6
Enter the time quantum: 3
```



Process	Arrival Time	Burst Time	Waiting Time	Turnaround Time	Response Time
P1	5	5	22	27	22
P2	4	6	17	23	17
P3	3	7	23	30	23
P4	1	9	8	17	8
P5	2	2	11	13	11
P6	6	3	15	18	15

Average Waiting Time: 16.00
Average Turnaround Time: 21.33
Average Response Time: 16.00
Throughput: 0.18 processes/unit time

OUTPUT ANALYSIS:

1. First Come First Serve (FCFS)

Time Complexity: O(n)

Processes are executed in the order they arrive.

Space Complexity: O(n)

Requires space to store process information, such as arrival time, burst time, completion time, waiting time, and turnaround time (TAT).

2. Shortest Job First (SJF)

Time Complexity: O(n log n)

Sorting processes by burst time takes O(n log n).

Space Complexity: O(n)

Space is needed to store process details and the sorted list.

3. Round Robin (RR)

Time Complexity: O(n * q)

Each process may be processed multiple times (up to n times), and each time quantum qqq may require O(n) operations. Thus, the complexity is O(n * q), where qqq is the time quantum.

Space Complexity: O(n)

Uses a queue to manage processes, requiring space proportional to the number of processes.

EXPERIMENT – 05

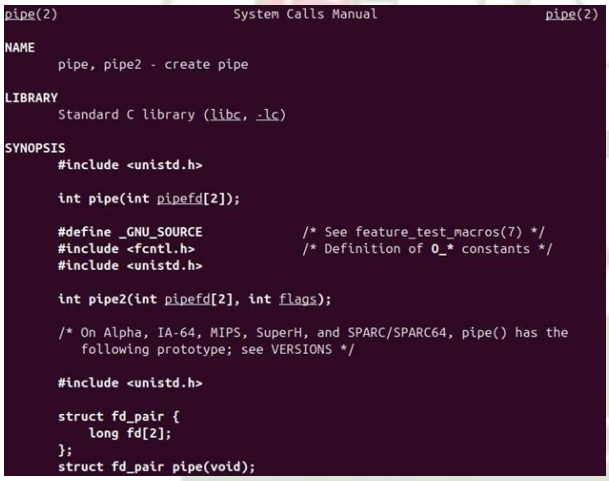
AIM: *DEMONSTARTION OF IPC MECHANISMS SUCH AS PIPE and SHARED MEMORY.*

DESCRITION:

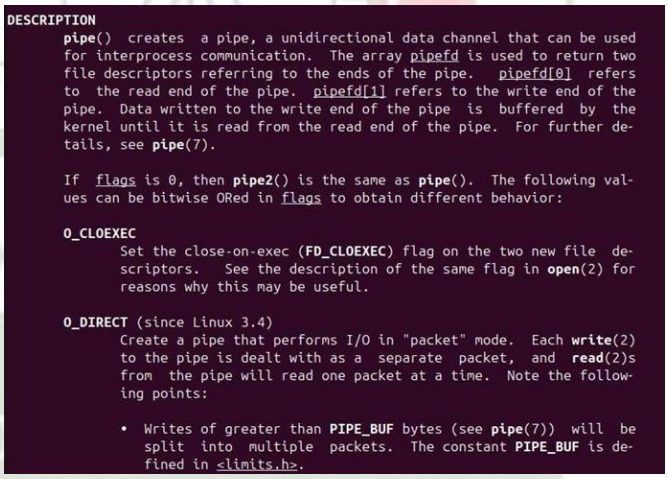
Inter-Process Communication (IPC) mechanisms, like **pipes** and **sharedmemory**, enable data exchange between processes.

- **Pipes** provide a unidirectional communication channel, allowing oneprocess to write and another to read data sequentially.
- **Shared memory** enables multiple processes to access common memory space, facilitating fast data exchange by minimizing copying.

Pipe()

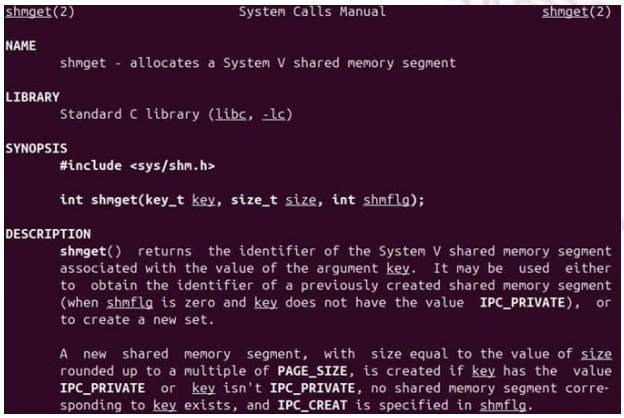


```
pipe(2)                    System Calls Manual                    pipe(2)

NAME
       pipe, pipe2 - create pipe

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <unistd.h>

       int pipe(int pipefd[2]);

       #define _GNU_SOURCE             /* See feature_test_macros(7) */
       #include <fcntl.h>              /* Definition of O_* constants */
       #include <unistd.h>

       int pipe2(int pipefd[2], int flags);

       /* On Alpha, IA-64, MIPS, SuperH, and SPARC/SPARC64, pipe() has the
          following prototype; see VERSIONS */

       #include <unistd.h>

       struct fd_pair {
           long fd[2];
       };
       struct fd_pair pipe(void);
```

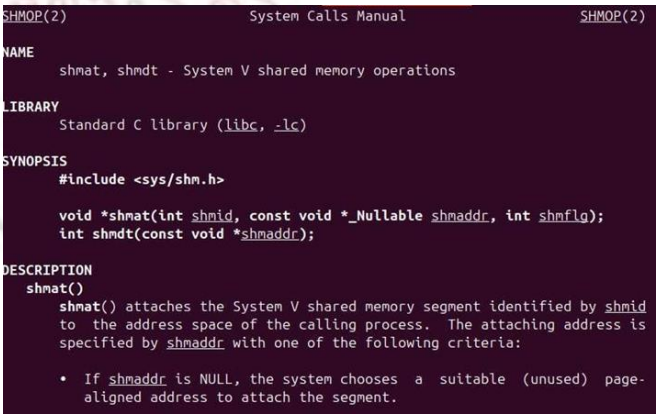


```
DESCRIPTION
       pipe()  creates  a pipe, a unidirectional data channel that can be used
       for interprocess communication.  The array pipefd is used to return two
       file descriptors referring to the ends of the pipe.  pipefd[0]  refers
       to  the read end of the pipe.  pipefd[1] refers to the write end of the
       pipe.  Data written to the write end of the pipe  is  buffered  by  the
       kernel until it is read from the read end of the pipe.  For further de-
       tails, see pipe(7).

       If  flags is 0, then pipe2() is the same as pipe().  The following val-
       ues can be bitwise ORed in flags to obtain different behavior:

       O_CLOEXEC
              Set the close-on-exec (FD_CLOEXEC) flag on the two new file  de-
              scriptors.   See the description of the same flag in open(2) for
              reasons why this may be useful.

       O_DIRECT (since Linux 3.4)
              Create a pipe that performs I/O in "packet" mode.  Each write(2)
              to the pipe is dealt with as a  separate  packet,  and  read(2)s
              from  the pipe will read one packet at a time.  Note the follow-
              ing points:

              •  Writes of greater than PIPE_BUF bytes (see pipe(7))  will  be
                 split  into  multiple  packets.  The constant PIPE_BUF is de-
                 fined in <limits.h>.
```

Shared Memory



```
shmget(2)                  System Calls Manual                  shmget(2)

NAME
       shmget - allocates a System V shared memory segment

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <sys/shm.h>

       int shmget(key_t key, size_t size, int shmflg);

DESCRIPTION
       shmget()  returns  the identifier of the System V shared memory segment
       associated with the value of the argument key.  It may be  used  either
       to  obtain the identifier of a previously created shared memory segment
       (when shmflg is zero and key does not have the value  IPC_PRIVATE),  or
       to create a new set.

       A  new  shared  memory  segment,  with  size equal to the value of size
       rounded up to a multiple of PAGE_SIZE, is created if key has the  value
       IPC_PRIVATE  or  key isn't IPC_PRIVATE, no shared memory segment corre-
       sponding to key exists, and IPC_CREAT is specified in shmflg.
```



```
SHMOP(2)                   System Calls Manual                   SHMOP(2)

NAME
       shmat, shmdt - System V shared memory operations

LIBRARY
       Standard C library (libc, -lc)

SYNOPSIS
       #include <sys/shm.h>

       void *shmat(int shmid, const void *_Nullable shmaddr, int shmflg);
       int shmdt(const void *shmaddr);

DESCRIPTION
   shmat()
       shmat() attaches the System V shared memory segment identified by shmid
       to  the address space of the calling process.  The attaching address is
       specified by shmaddr with one of the following criteria:

       •  If shmaddr is NULL, the system chooses  a  suitable  (unused)  page-
          aligned address to attach the segment.
```

shmdt()

shmdt() detaches the shared memory segment located at the address specified by shmaddr from the address space of the calling process. The to-be-detached segment must be currently attached with shmaddr equal to the value returned by the attaching **shmat**() call.

On a successful **shmdt**() call, the system updates the members of the shmid_ds structure associated with the shared memory segment as follows:

- shm_dtime is set to the current time.
- shm_lpid is set to the process-ID of the calling process.
- shm_nattch is decremented by one. If it becomes 0 and the segment is marked for deletion, the segment is deleted.

1. Program to Demonstrate Ordinary Pipe IPC Mechanism

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
    int fd[2];
    if (pipe(fd) == -1) {
        perror("Pipe creation failed");
        exit(EXIT_FAILURE);
    }
    // Fork a child process
    pid_t pid = fork();
    if (pid == -1) {
        perror("Fork failed");
        exit(EXIT_FAILURE);
    }
    if (pid == 0) {
        // Child process
        close(fd[0]);  // Close the read end of the pipe
        char message[] = "Hello from the child process!\n";
        write(fd[1], message, sizeof(message));
        close(fd[1]);  // Close the write end of the pipe in the child
    } else {
        // Parent process
        close(fd[1]);  // Close the write end of the pipe
        char buffer[100];
        read(fd[0], buffer, sizeof(buffer));
        printf("Parent received: %s", buffer);
        close(fd[0]);  // Close the read end of the pipe in the parent
    }
    return 0;
}
```

OUTPUT:

```
cbit@cbit-VirtualBox:~$ gedit pipec.c
cbit@cbit-VirtualBox:~$ cc pipec.c
cbit@cbit-VirtualBox:~$ ./a.out
Parent received: Hello from the child process!
cbit@cbit-VirtualBox:~$
```

2. Program to Demonstrate Shared Memory IPC Mechanism.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#define SHM_SIZE 1024  // Size of the shared memory
int main() {
    key_t key = ftok("shmfile", 65);  // Generate unique key
    int shmid = shmget(key, SHM_SIZE, 0666 | IPC_CREAT);  // Create shared memory
    if (shmid == -1) {
        perror("shmget failed");
        exit(1);
    }
    char *shared_memory = (char*) shmat(shmid, NULL, 0);  // Attach to shared memory
    if (fork() == 0) {  // Child Process
        sleep(1);  // Ensuring parent writes first
        printf("Child reads: %s\n", shared_memory);  // Read data from shared memory
        shmdt(shared_memory);  // Detach from shared memory
    } else {  // Parent Process
        strcpy(shared_memory, "Hello from parent!");  // Write data to shared memory
        printf("Parent wrote to shared memory.\n");
        wait(NULL);  // Wait for child to finish
        shmctl(shmid, IPC_RMID, NULL);  // Destroy shared memory
    }
    return 0;
}
```

OUTPUT:

Parent Process: Creates a shared memory segment and writes data to it.

Child Process: Reads the data from the shared memory.

```
Parent wrote to shared memory.
Child reads: Hello from parent!
```

EXPERIMENT – 06

AIM: *IMPLEMENATION OF SOCKET COMMANDS SUCH AS SOCKET, SEND, RECV, BIND, LISTEN, ACCEPT, CONNECT.*

DESCRITION:

1. **socket()**: Initializes a new socket instance for network communication, specifying the address family and type (e.g., TCP/UDP).
2. **send()**: Sends data to a connected socket in a TCP connection, used for transmitting messages between client and server.
3. **recv()**: Receives data from a connected socket in a TCP connection, waiting for incoming data from the peer.
4. **bind()**: Assigns a specific local IP address and port to a socket, preparing it for listening to incoming connections.
5. **listen()**: Puts the socket in a passive mode, allowing it to accept incoming connection requests from clients.
6. **accept()**: Accepts an incoming connection on a listening socket, creating a new socket to handle client communication.
7. **connect()**: Initiates a connection to a remote server, allowing the client to establish a communication channel.

socket()

```
SOCKET(2)                  Linux Programmer's Manual                  SOCKET(2)

NAME
       socket - create an endpoint for communication

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

       int socket(int domain, int type, int protocol);

DESCRIPTION
       socket()  creates  an endpoint for communication and returns a file de-
       scriptor that refers to that endpoint.  The file descriptor returned by
       a  successful call will be the lowest-numbered file descriptor not cur-
       rently open for the process.

       The domain argument specifies a communication domain; this selects  the
       protocol  family  which will be used for communication.  These families
       are defined in <sys/socket.h>.  The formats currently understood by the
       Linux kernel include:

       Name         Purpose                                    Man page
       AF_UNIX      Local communication                        unix(7)
       AF_LOCAL     Synonym for AF_UNIX
       AF_INET      IPv4 Internet protocols                    ip(7)
       AF_AX25      Amateur radio AX.25 protocol               ax25(4)
       AF_IPX       IPX - Novell protocols
       AF_APPLETALK AppleTalk                                  ddp(7)
       AF_X25       ITU-T X.25 / ISO-8208 protocol             x25(7)
       AF_INET6     IPv6 Internet protocols                    ipv6(7)
       AF_DECnet    DECet protocol sockets
       AF_KEY       Key  management protocol, originally de-
                    veloped for usage with IPsec
       AF_NETLINK   Kernel user interface device               netlink(7)
       AF_PACKET    Low-level packet interface                 packet(7)
       AF_RDS       Reliable Datagram Sockets (RDS) protocol   rds(7)
                                                               rds-rdma(7)
```

recv()

```
RECV(2)                    Linux Programmer's Manual                    RECV(2)

NAME
       recv, recvfrom, recvmsg - receive a message from a socket

SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>

       ssize_t recv(int sockfd, void *buf, size_t len, int flags);

       ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                        struct sockaddr *src_addr, socklen_t *addrlen);

       ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);

DESCRIPTION
       The  recv(),  recvfrom(),  and recvmsg() calls are used to receive mes-
       sages from a socket.  They may be used to receive data on both  connec-
       tionless  and  connection-oriented  sockets.  This page first describes
       common features of all three system calls, and then describes the  dif-
       ferences between the calls.

       The  only  difference  between  recv()  and  read(2) is the presence of
       flags.  With a zero flags argument, recv() is generally  equivalent  to
       read(2) (but see NOTES).  Also, the following call

           recv(sockfd, buf, len, flags);

       is equivalent to

           recvfrom(sockfd, buf, len, flags, NULL, NULL);
```

listen()

```
LISTEN(2)                                  Linux Programmer's Manual                                  LISTEN(2)

NAME
       listen - listen for connections on a socket

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

       int listen(int sockfd, int backlog);

DESCRIPTION
       listen() marks the socket referred to by sockfd as a passive socket, that is, as a socket that will be used to accept in-
       coming connection requests using accept(2).

       The sockfd argument is a file descriptor that refers to a socket of type SOCK_STREAM or SOCK_SEQPACKET.

       The backlog argument defines the maximum length to which the queue of pending connections for sockfd may grow.  If a con-
       nection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED or, if
       the underlying protocol supports retransmission, the request may be ignored so that a later reattempt at connection  suc-
       ceeds.

RETURN VALUE
       On success, zero is returned.  On error, -1 is returned, and errno is set appropriately.
```

bind()

```
LISTEN(2)                                  Linux Programmer's Manual                                  LISTEN(2)

NAME
       listen - listen for connections on a socket

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

       int listen(int sockfd, int backlog);

DESCRIPTION
       listen() marks the socket referred to by sockfd as a passive socket, that is, as a socket that will be used to accept in-
       coming connection requests using accept(2).

       The sockfd argument is a file descriptor that refers to a socket of type SOCK_STREAM or SOCK_SEQPACKET.

       The backlog argument defines the maximum length to which the queue of pending connections for sockfd may grow.  If a con-
       nection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED or, if
       the underlying protocol supports retransmission, the request may be ignored so that a later reattempt at connection  suc-
       ceeds.

RETURN VALUE
       On success, zero is returned.  On error, -1 is returned, and errno is set appropriately.
```

accept(

```
ACCEPT(2)                                  Linux Programmer's Manual                                  ACCEPT(2)

NAME
       accept, accept4 - accept a connection on a socket

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

       int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

       #define _GNU_SOURCE             /* See feature_test_macros(7) */
       #include <sys/socket.h>

       int accept4(int sockfd, struct sockaddr *addr,
                   socklen_t *addrlen, int flags);

DESCRIPTION
       The accept() system call is used with connection-based socket types (SOCK_STREAM, SOCK_SEQPACKET).  It extracts the first
       connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected  socket,
       and returns a new file descriptor referring to that socket.  The newly created socket is not in the listening state.  The
       original socket sockfd is unaffected by this call.
```

connect()

```
CONNECT(2)                                 Linux Programmer's Manual                                 CONNECT(2)

NAME
       connect - initiate a connection on a socket

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

       int connect(int sockfd, const struct sockaddr *addr,
                   socklen_t addrlen);

DESCRIPTION
       The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr.
       The addrlen argument specifies the size of addr.  The format of the address in addr is determined by the address space of
       the socket sockfd; see socket(2) for further details.

       If the socket sockfd is of type SOCK_DGRAM, then addr is the address to which datagrams are sent by default, and the only
       address from which datagrams are received.  If the socket is of type SOCK_STREAM or SOCK_SEQPACKET, this call attempts to
       make a connection to the socket that is bound to the address specified by addr.
```

Program to demonstrate Socket Programming.

Server

server.py > ...

```python
import socket
def start_server():
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_address = ('localhost', 1234)
    server_socket.bind(server_address)
    server_socket.listen(1)
    print("Server is listening on port 1234...")
    while True:
        client_socket, client_address = server_socket.accept()
        print(f"Connection from {client_address} has been established.")
        welcome_message = "Welcome to the server!"
        client_socket.sendall(welcome_message.encode('utf-8'))
        client_message = client_socket.recv(1024)
        print("Received from client:", client_message.decode('utf-8'))
        response_message = "Message received!"
        client_socket.sendall(response_message.encode('utf-8'))
        client_socket.close()
if __name__ == "__main__":
    start_server()
```

Client

client.py > ...

```python
import socket
def start_client():
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_address = ('localhost', 1234)
    client_socket.connect(server_address)
    data = client_socket.recv(1024)
    print("Received from server:", data.decode('utf-8'))
    message = "Hello, Server! This is the client."
    client_socket.sendall(message.encode('utf-8'))
    server_response = client_socket.recv(1024)
    print("Received from server:", server_response.decode('utf-8'))
    client_socket.close()
if __name__ == "__main__":
    start_client()
```

OUTPUT:

```
PS C:\Users\CBIT-CET\Documents\63> python server.py
Server is listening on port 1234...
PS C:\Users\CBIT-CET\Documents\63> python client.py
Received from server: Welcome to the server!
Received from server: Message received!
PS C:\Users\CBIT-CET\Documents\63> python server.py
Server is listening on port 1234...
Connection from ('127.0.0.1', 51820) has been established
Received from client: Hello, Server! This is the client.
```

EXPERIMENT – 07

AIM: *IMPLEMENT PAGE REPLACEMENT ALGORITHMS*
(a) *FIFO (b) LRU (c)OPTIMAL*

DESCRITION:

In operating systems, whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

(a) FIFO (FIRST IN FIRST OUT)

This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

```
def fifo_page_replacement(pages, frame_size):
    page_frame = []
    page_faults = 0
    hits = 0

    for page in pages:
        if page not in page_frame:
            page_faults += 1
            if len(page_frame) >= frame_size:
                page_frame.pop(0)
            page_frame.append(page)
        else:
            hits += 1

    total_references = len(pages)
    miss_ratio = page_faults / total_references
    hit_ratio = hits / total_references

    return page_faults, hits, miss_ratio, hit_ratio

pages_input = input("Enter the page references string (comma-separated): ")
pages = list(map(int, pages_input.split(',')))
frame_size = int(input("Enter the frame size: "))

page_faults, hits, miss_ratio, hit_ratio = fifo_page_replacement(pages, frame_size)

print("FIFO Page Faults:", page_faults)
print("Hits:", hits)
print("Miss Ratio:", miss_ratio)
print("Hit Ratio:", hit_ratio)
```

OUTPUT:

```
Enter the page references string (comma-separated): 4,7,6,1,7,6,1,2,7,2
Enter the frame size: 3
FIFO Page Faults: 6
Hits: 4
Miss Ratio: 0.6
Hit Ratio: 0.4
```

OUTPUT ANALYSIS:

FIFO does not consider the future need or recent usage of a page, which can result in high page faults, especially in scenarios with repetitive access patterns.

(b) LRU (LEAST RECENTLY USED)

In **L**east **R**ecently **U**sed (LRU) algorithm is a Greedy algorithm where the page to be replaced is least recently used. This algorithm is based on the strategy that whenever a page fault occurs, the least recently used page will be replaced with a new page. So, the page not utilized for the longest time in the memory (compared to all other pages) gets replaced.

```
def lru_page_replacement(pages, frame_size):
    page_frame = []
    page_faults = 0
    hits = 0

    for page in pages:
        if page in page_frame:
            hits += 1
            page_frame.remove(page)
            page_frame.append(page)
        else:
            page_faults += 1
            if len(page_frame) >= frame_size:
                page_frame.pop(0)
            page_frame.append(page)

    total_references = len(pages)
    miss_ratio = page_faults / total_references
    hit_ratio = hits / total_references

    return page_faults, hits, miss_ratio, hit_ratio

pages_input = input("Enter the page references string (comma-separated): ")
pages = list(map(int, pages_input.split(',')))
frame_size = int(input("Enter the frame size: "))

page_faults, hits, miss_ratio, hit_ratio = lru_page_replacement(pages, frame_size)

print("LRU Page Faults:", page_faults)
print("Hits:", hits)
print("Miss Ratio:", miss_ratio)
print("Hit Ratio:", hit_ratio)
```

OUTPUT:

```
Enter the page references string (comma-separated): 4,7,6,1,7,6,1,2,7,2
Enter the frame size: 3
LRU Page Faults: 6
Hits: 4
Miss Ratio: 0.6
Hit Ratio: 0.4
```

OUTPUT ANALYSIS:

LRU outperforms FIFO by considering the "recency" of page accesses, aiming to retain pages that have been used recently. This makes it a practical algorithm for reducing page faults in scenarios with frequent re-access patterns.

(c) OPTIMAL

In this algorithm, OS replaces the page that will not be used for the longest period of time in future.

```
def optimal_page_replacement(pages, frame_size):
    page_frame = []
    page_faults = 0
    hits = 0

    for i in range(len(pages)):
        page = pages[i]

        if page in page_frame:
            hits += 1
        else:
            page_faults += 1

            if len(page_frame) < frame_size:
                page_frame.append(page)
            else:
                future_uses = []

                for frame_page in page_frame:
                    if frame_page in pages[i+1:]:
                        future_uses.append(pages[i+1:].index(frame_page))
                    else:
                        future_uses.append(float('inf'))

                frame_to_replace = future_uses.index(max(future_uses))
                page_frame[frame_to_replace] = page

    total_references = len(pages)
    miss_ratio = page_faults / total_references
    hit_ratio = hits / total_references

    return page_faults, hits, miss_ratio, hit_ratio

pages_input = input("Enter the page references string (comma-separated): ")
pages = list(map(int, pages_input.split(',')))
frame_size = int(input("Enter the frame size: "))

page_faults, hits, miss_ratio, hit_ratio = optimal_page_replacement(pages, frame_size)

print("Optimal Page Faults:", page_faults)
print("Hits:", hits)
print("Miss Ratio:", miss_ratio)
print("Hit Ratio:", hit_ratio)
```

OUTPUT:

```
Enter the page references string (comma-separated): 4,7,6,1,7,6,1,2,7,2
Enter the frame size: 3
Optimal Page Faults: 5
Hits: 5
Miss Ratio: 0.5
Hit Ratio: 0.5
```

OUTPUT ANALYSIS:

The Optimal algorithm is the theoretical best solution, achieving the minimum possible page faults by replacing the page that will not be used for the longest time in the future.

EXPERIMENT – 08

AIM: *TO CREATE AND EXECUTE TWO THREADS THAT PERFORM DIFFERENT TASKS CONCURRENTLY*

DESCRITION:

This program creates two threads: one that prints numbers and another that prints letters. Each thread runs concurrently, allowing both tasks to execute in parallel.

CODE:

```
import threading
import time

# Task for the first thread: Counting numbers
def count_numbers():
    for i in range(1, 6):
        print(f"Count: {i}")
        time.sleep(1)  # Simulating a time-consuming task

# Task for the second thread: Printing letters
def print_letters():
    for letter in "ABCDE":
        print(f"Letter: {letter}")
        time.sleep(1)  # Simulating a time-consuming task

# Creating two threads for the tasks
thread1 = threading.Thread(target=count_numbers)
thread2 = threading.Thread(target=print_letters)

# Starting both threads
thread1.start()
thread2.start()

# Waiting for both threads to complete
thread1.join()
thread2.join()

print("Both tasks completed.")
```

OUTPUT:

```
Count: 1Letter: A

Count: 2Letter: B

Count: 3Letter: C

Count: 4Letter: D

Count: 5Letter: E

Both tasks completed.
```

OUTPUT ANALYSIS :

- **Concurrency:** The interleaved output demonstrates that both tasks run concurrently, even though Python's Global Interpreter Lock (GIL) serializes execution at the bytecode level. With I/O-bound or delay-based operations, threads can still appear to run in parallel.
- **Timing and Intervals:** Each thread has an independent 1-second interval, creating nearly simultaneous outputs in an alternating pattern, which can vary slightly based on thread scheduling by the OS.

This output analysis demonstrates concurrent behavior in multi-threaded Python Programs where each thread completes its task independently but simultaneously with another thread.

EXPERIMENT – 09

AIM: *IMPLEMENTATION OF CLASSICAL PROBLEMS FOR SYNCHRONIZATION (DINING PHILOSOPHER PROBLEM AND PRODUCER- CONSUMER PROBLEM.)*

DESCRITION:

1. **DINING PHILOSOPHER PROBLEM**

   The Dining Philosopher Problem involves five philosophers sitting at a table who either think or eat. There are five forks placed between them, and each philosopher needs two forks to eat. The challenge is to devise a synchronization mechanism to avoid deadlocks, ensuring no philosopher is indefinitely hungry.

2. **PRODUCER-CONSUMER PROBLEM**

   The Producer-Consumer Problem involves two types of threads: producers, which add items to a shared buffer, and consumers, which remove items. The problem is to make sure that:
   - Producers don't add items when the buffer is full.
   - Consumers don't remove items when the buffer is empty.

DINING PHILOSOPHER PROBLEM CODE:

```
import threading
import time
import random

NUM_PHILOSOPHERS = 5

forks = [threading.Semaphore(1) for _ in range(NUM_PHILOSOPHERS)]

def philosopher(id):
    left_fork = id
    right_fork = (id + 1) % NUM_PHILOSOPHERS

    while True:
        print(f"Philosopher {id} is thinking.")
        time.sleep(random.uniform(1, 3))

        print(f"Philosopher {id} is hungry.")

        with forks[left_fork]:
            with forks[right_fork]:
                print(f"Philosopher {id} is eating.")
                time.sleep(random.uniform(1, 2))

        print(f"Philosopher {id} finished eating and is thinking.")

philosophers = [threading.Thread(target=philosopher, args=(i,)) for i in range(NUM_PHILOSOPHERS)]
for p in philosophers:
    p.start()
for p in philosophers:
    p.join()
```

OUTPUT:

```
Philosopher 2 is thinking.
Philosopher 3 is hungry.
Philosopher 1 finished eating and is thinking.Philosopher 0 is eating.

Philosopher 2 is hungry.Philosopher 1 is thinking.

Philosopher 0 finished eating and is thinking.Philosopher 4 is eating.

Philosopher 0 is thinking.
Philosopher 1 is hungry.
Philosopher 4 finished eating and is thinking.Philosopher 3 is eating.

Philosopher 4 is thinking.
Philosopher 0 is hungry.
Philosopher 3 finished eating and is thinking.Philosopher 2 is eating.

Philosopher 3 is thinking.
Philosopher 4 is hungry.
Philosopher 2 finished eating and is thinking.Philosopher 1 is eating.

Philosopher 2 is thinking.
Philosopher 3 is hungry.
Philosopher 1 finished eating and is thinking.Philosopher 0 is eating.

Philosopher 1 is thinking.
Philosopher 2 is hungry.
Philosopher 0 finished eating and is thinking.Philosopher 4 is eating.

Philosopher 0 is thinking.
Philosopher 1 is hungry.
Philosopher 0 is hungry.
Philosopher 4 finished eating and is thinking.Philosopher 3 is eating.
```

PRODUCER CONSUMER PROBLEM CODE:

```python
import threading
import time
import random

buffer = []
BUFFER_SIZE = 5
buffer_lock = threading.Condition()

def producer():
    while True:
        item = random.randint(1, 100)
        with buffer_lock:
            while len(buffer) >= BUFFER_SIZE:
                print("Buffer full, producer is waiting.")
                buffer_lock.wait()
            buffer.append(item)
            print(f"Producer produced item: {item}")
            buffer_lock.notify()
        time.sleep(random.uniform(1, 2))

def consumer():
    while True:
        with buffer_lock:
            while not buffer:
                print("Buffer empty, consumer is waiting.")
                buffer_lock.wait()
            item = buffer.pop(0)
            print(f"Consumer consumed item: {item}")
            buffer_lock.notify()
        time.sleep(random.uniform(1, 3))

producers = [threading.Thread(target=producer) for _ in range(2)]
consumers = [threading.Thread(target=consumer) for _ in range(2)]

for p in producers + consumers:
    p.start()

for p in producers + consumers:
    p.join()
```

OUTPUT:

```
Producer produced item: 19
Producer produced item: 91
Consumer consumed item: 19
Consumer consumed item: 91
Producer produced item: 68
Producer produced item: 32
Consumer consumed item: 68
Producer produced item: 7
Consumer consumed item: 32
Producer produced item: 55
Producer produced item: 67
Producer produced item: 55
Consumer consumed item: 7
Producer produced item: 32
Consumer consumed item: 55
Producer produced item: 94
Producer produced item: 59
Consumer consumed item: 67
Producer produced item: 27
Buffer full, producer is waiting.
Consumer consumed item: 55
Producer produced item: 77
Buffer full, producer is waiting.
Consumer consumed item: 32
Producer produced item: 41
Buffer full, producer is waiting.
Consumer consumed item: 41
Producer produced item: 35
Buffer full, producer is waiting.
Consumer consumed item: 42
Producer produced item: 73
Consumer consumed item: 94
Consumer consumed item: 88
Producer produced item: 32
Consumer consumed item: 67
Producer produced item: 63
Producer produced item: 51
Consumer consumed item: 35
Producer produced item: 64
Consumer consumed item: 73
Producer produced item: 96
Consumer consumed item: 32
Producer produced item: 6
Consumer consumed item: 63
Producer produced item: 34
Buffer full, producer is waiting.
Consumer consumed item: 51
Producer produced item: 62
Buffer full, producer is waiting.
Consumer consumed item: 64
```

OUTPUT ANALYSIS :

The Dining Philosophers and Producer-Consumer problems demonstrate efficient synchronization by carefully coordinating access to shared resources, which avoids both deadlocks and race conditions. In both implementations, each thread or process waits only when necessary, allowing independent and concurrent operations that enhance performance and prevent blocking.

EXPERIMENT – 10

AIM: *IMPLEMENTION OF BANKERS ALGORITHM FOR DEADLOCK DETECTION AND AVOIDANCE*

DESCRITION:

The Banker's Algorithm is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra. It is used in operating systems to manage the allocation of resources to multiple processes in a way that ensures safe execution without deadlock. This algorithm is particularly useful in systems where resources are limited and can be allocated to processes at any point in time.

1. **Safe State**: A system is in a safe state if it can allocate resources to each process in some order and still avoid a deadlock. In other words, there exists at least one sequence that allows each process to complete without causing a deadlock.
2. **Resource Allocation**: The algorithm determines if the resources available are sufficient to fulfill a process's request. If so, resources are allocated, but only if the system remains in a safe state after the allocation.
3. **Need Matrix**: Represents the remaining resources that each process will require to complete its execution after a certain amount has already been allocated.
4. **Allocation Matrix**: Tracks the current resources assigned to each process.
5. **Available Vector**: Represents the resources that are available in the system.
6. **Request**: When a process requests resources, the algorithm checks whether allocating these resources will leave the system in a safe state.

Algorithm Steps

1. **Initialization**: Define the matrices (Allocation, Max, Need) and the Available vector. The Need matrix is calculated as:
   Need[i][j]=Max[i][j]−Allocation[i][j]
2. **Request Handling**:
   - If a process requests resources, check if the requested resources are less than or equal to the Need for that process.
   - Check if the requested resources are available.
3. **Safety Check**:
   - Temporarily allocate the requested resources to the process and update the Available and Need matrices.
   - Check if the system is in a safe state by trying to find a sequence in which all processes can execute without deadlock.
   - If a safe sequence exists, grant the request permanently. If not, revert to the previous state and deny the request.
4. **Deadlock Detection**:
   - The algorithm periodically checks if the system is in an unsafe state, which indicates a potential deadlock.

```python
def get_matrix(rows, cols, name):
    matrix = []
    print(f"Enter the {name} matrix:")
    for i in range(rows):
        row = list(map(int, input(f"Row {i + 1}: ").split()))
        while len(row) != cols:
            print(f"Please enter exactly {cols} integers.")
            row = list(map(int, input(f"Row {i + 1}: ").split()))
        matrix.append(row)
    return matrix

n = int(input("Enter the number of processes: "))
m = int(input("Enter the number of resources: "))

alloc = get_matrix(n, m, "Allocation")
max = get_matrix(n, m, "Maximum")

avail = list(map(int, input("Enter available resources (space-separated): ").split()))
while len(avail) != m:
    print(f"Please enter exactly {m} integers.")
    avail = list(map(int, input("Enter available resources (space-separated): ").split()))

f = [0] * n
ans = [0] * n
ind = 0

need = [[0 for _ in range(m)] for _ in range(n)]
for i in range(n):
    for j in range(m):
        need[i][j] = max[i][j] - alloc[i][j]

for _ in range(n):
    for i in range(n):
        if f[i] == 0:
            flag = 0
            for j in range(m):
                if need[i][j] > avail[j]:
                    flag = 1
                    break

            if flag == 0:
                ans[ind] = i
                ind += 1
                for y in range(m):
                    avail[y] += alloc[i][y]
                f[i] = 1

print("Following is the SAFE Sequence:")
for i in range(n - 1):
    print(" P", ans[i], " ->", sep="", end="")
print(" P", ans[n - 1], sep="")
```

OUTPUT:

```
Enter the number of processes: 5
Enter the number of resources: 3
Enter the Allocation matrix:
Row 1: 0 1 0
Row 2: 2 0 0
Row 3: 3 0 2
Row 4: 2 1 1
Row 5: 0 0 2
Enter the Maximum matrix:
Row 1: 7 5 3
Row 2: 3 2 2
Row 3: 9 0 2
Row 4: 2 2 2
Row 5: 4 3 3
Enter available resources (space-separated): 3 3 2
Following is the SAFE Sequence:
 P1 -> P3 -> P4 -> P0 -> P2
```

OUTPUT ANALYSIS :

Input Breakdown

1. Number of Processes: 5
2. Number of Resources: 3
3. Allocation Matrix: This matrix represents the current allocation of resources for each process.

P0: [0, 1, 0]
P1: [2, 0, 0]
P2: [3, 0, 2]
P3: [2, 1, 1]
P4: [0, 0, 2]

4. Maximum Matrix: This matrix shows the maximum resources each process might request.

P0: [7, 5, 3]
P1: [3, 2, 2]
P2: [9, 0, 2]
P3: [2, 2, 2]
P4: [4, 3, 3]

5. Available Resources: [3, 3, 2] This array shows the currently available resources of each type.

Safe Sequence Analysis

The safe sequence given by the algorithm is: P1 -> P3 -> P4 -> P0 -> P2
The safe sequence indicates the order in which processes can complete without causing a deadlock.
Here's the step-by-step reasoning for the algorithm finding this sequence:

- Step 1: Check for a process that can complete with the available resources [3, 3, 2].
  - P1 can complete because its need ([1, 2, 2]) can be satisfied with the available resources.
  - Execute P1, freeing up its allocated resources, resulting in new available resources: [5, 3, 2].
- Step 2: With resources [5, 3, 2], check the remaining processes.
  - P3 can complete because its need ([0, 1, 1]) is within the available resources.
  - Execute P3, freeing its allocated resources, resulting in new available resources: [7, 4, 3].
- Step 3: With resources [7, 4, 3], check the remaining processes.
  - P4 can complete with its need ([4, 3, 1]) being within the available resources.
  - Execute P4, freeing its allocated resources, resulting in new available resources: [7, 4, 5].
- Step 4: With resources [7, 4, 5], check the remaining processes.
  - P0 can complete as its need ([7, 4, 3]) is within the available resources.
  - Execute P0, freeing its allocated resources, resulting in new available resources: [7, 5, 5].
- Step 5: Finally, with resources [7, 5, 5], P2 can complete as its need ([6, 0, 0]) is within the available resources.
  - Execute P2, freeing its allocated resources.

Conclusion

The Banker's Algorithm has determined a safe sequence P1 -> P3 -> P4 -> P0 -> P2, meaning that this system state is safe and no deadlock will occur if processes follow this execution order. The allocation of resources is managed in a way that each process can complete safely and release its resources for other processes in the sequence, ensuring overall system stability.

EXPERIMENT – 11

AIM: *IMPLEMENTION OF LINKED, INDEXED AND CONTIGUOUS FILE ALLOCATION METHODS.*

DESCRITION:

1. Contiguous File Allocation

In Contiguous File Allocation, each file is stored in contiguous blocks on the disk. The file's metadata contains the starting block and the length (number of blocks) allocated to the file. This approach allows direct access but may lead to fragmentation.

2. Linked File Allocation

In Linked Allocation, each file is stored in non-contiguous blocks on the disk, with each block containing a pointer to the next block. This eliminates fragmentation but makes direct access more challenging.

3. Indexed File Allocation

In Indexed Allocation, each file has an index block containing pointers to all the disk blocks used by the file. This method allows direct access without requiring contiguous blocks, but each file needs an additional index block.

CODE:

1. Contiguous File Allocation

```
def main():
    n = int(input("Enter no. of files: "))
    b = [0] * 20
    sb = [0] * 20
    t = [0] * 20
    c = [[0] * 20 for _ in range(20)]

    for i in range(n):
        b[i] = int(input(f"Enter no. of blocks occupied by file-{i + 1}: "))
        sb[i] = int(input(f"Enter the starting block of file-{i + 1}: "))
        t[i] = sb[i]
        for j in range(b[i]):
            c[i][j] = sb[i]
            sb[i] += 1
```

```
    print("Filename\tStart block\tlength")
    for i in range(n):
        print(f"{i + 1}\t\t {t[i]} \t\t{b[i]}")

    x = int(input("Enter file name: "))
    print(f"File name is: {x}")
    print(f"length is: {b[x - 1]}")
    print("Blocks occupied: ", end="")
    for i in range(b[x - 1]):
        print(f"{c[x - 1][i]:4}", end="")
    print()

if __name__ == "__main__":
    main()
```

OUTPUT:

```
Enter no.of files: 2
Enter no. of blocks occupied by file-1: 4
Enter the starting block of file-1: 2
Enter no. of blocks occupied by file-2: 15
Enter the starting block of file-2: 4
Filename        Start block     length
1               2               4
2               4               15
Enter file name:2
File name is:2
length is:15
Blocks occupied:    4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
```

OUTPUT ANALYSIS :

By executing the above program, we have successfully implemented the Contiguous File Allocation Method

2. Indexed File Allocation

```
def main():
    n = int(input("Enter no. of files: "))
    b = [0] * 20
    sb = [0] * 20
    t = [0] * 20
    c = [[0] * 20 for _ in range(20)]

    for i in range(n):
        b[i] = int(input(f"Enter no. of blocks occupied by file-{i + 1}: "))
        sb[i] = int(input(f"Enter the starting block of file-{i + 1}: "))
        t[i] = sb[i]
        for j in range(b[i]):
```

```
            c[i][j] = sb[i]
            sb[i] += 1

        print("Filename\tStart block\tlength")
        for i in range(n):
            print(f"{i + 1}\t\t {t[i]} \t\t{b[i]}")

        x = int(input("Enter file name: "))
        print(f"File name is: {x}")
        print(f"length is: {b[x - 1]}")
        print("Blocks occupied: ", end="")
        for i in range(b[x - 1]):
            print(f"{c[x - 1][i]:4}", end="")
        print()

    if __name__ == "__main__":
        main()
```

OUTPUT:

```
Enter no. of files: 2
Enter starting block and size of file-1: 2 5
Enter blocks occupied by file-1: 10
enter blocks of file-1: 3 2 5 4 6 7 2 6 4 7
Enter starting block and size of file-2: 3 4
Enter blocks occupied by file-2: 5
enter blocks of file-2: 3 4 5 6 7

File     index  length
1       2       10
2       3       5

Enter file name: 2
File name is:2
Index is:3
Block occupied are:  3  4  5  6  7
```

OUTPUT ANALYSIS :

By executing the above program, we have successfully implemented the Indexed File Allocation Method.

3. Linked File Allocation

```
class File:
    def __init__(self):
        self.fname = ""
        self.start = 0
        self.size = 0
        self.block = [0] * 10
```

```
def main():
    files = []
    n = int(input("Enter no. of files: "))

    for i in range(n):
        file = File()
        file.fname = input("Enter file name: ")
        file.start = int(input("Enter starting block: "))
        file.block[0] = file.start
        file.size = int(input("Enter no. of blocks: "))

        print("Enter block numbers:")
        for j in range(1, file.size):
            file.block[j] = int(input())

        files.append(file)

    print("File\tstart\tsize\tblock")
    for file in files:
        print(f"{file.fname}\t{file.start}\t{file.size}\t", end="")
        for j in range(1, file.size - 1):
            print(f"{file.block[j]}--->", end="")
        print(file.block[file.size - 1])

if __name__ == "__main__":
    main()
```

OUTPUT:

```
Enter no. of files:2
Enter file name:os
Enter starting block:20
Enter no.of blocks:6
Enter block numbers:4 12 15 45 32 25
Enter file name:lab
Enter starting block:12
Enter no.of blocks:5
Enter block numbers:6 5 4 3 2
File    start   size    block
os      20      6       4--->12--->15--->45--->32--->25
lab     12      5       6--->5--->4--->3--->2
```

OUTPUT ANALYSIS :

By executing the above program, we have successfully implemented the Indexed File Allocation Method.