SRI CHANDRASEKHARENDRA SARASWATHI VISWA MAHAVIDYALAYA

(UNIVERSITY ESTABLISHED UNDER SECTION 3 OF UGC ACT 1956)
ENATHUR,KANCHIPURAM – 631561

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

OPERATING SYSTEM FOR ENGINEERS LAB RECORD

Name : G.Rohini RamaLakshmi
Reg.No : 11239A032
Class : 3rd year
Course : BE-CSE
Faculty in charge : Dr.T.Lakshmibai

SRI CHANDRASEKHARENDRA SARASWATHI VISWA MAHAVIDYALAYA

(UNIVERSITY ESTABLISHED UNDER SECTION 3 OF UGC ACT 1956)
ENATHUR,KANCHIPURAM – 631561

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

BONAFIDE CERTIFICATE

This is to certify that this is the bonafide record of work done by Mr/Mrs. G.Rohini Ramalakshmi with Reg.no 11239A032 of II-B.E-CSE in the OPERATING SYSTEM FOR ENGINEERS during the academic year 2025-2026

Station: **Enathur**
Date :

Staff-in-charge **Head of the department**

Submitted for the practical examination held on_____

Examiner-1 **Examiner-2**

SL.NO	Program Name	Date	Signature
1	Basic UNIX and LINUX Commands	21/08/25	
2	Shell Programming	28/08/25	
3	GREP, SED, AWK Experiments	04/09/25	
4	System Calls	11/09/25	
5	Process Management A.fork() B.exec()	18/09/25	
6	CPU Scheduling Algorithm A.FIFO B.Round Robin	18/09/25	
7	Banker's Algorithm for Deadlock Avoidance	16/10/25	
8	PROCESS SYNCHRONIZATION **Write a program to solve the Producer Consumer Problem using Semaphores**	16/10/25	
9	PROCESS SYNCHRONIZATION **Write a program to implement Dining Philosophers problem using Threads and Semaphores.**	30/10/25	

Experiment 1: Basic UNIX and LINUX Commands

Introduction:

UNIX and Linux are multi- user, multitasking operating systems widely used in servers, development, and research. They provide a powerful command- line interface (CLI) that enables users to efficiently manage files, processes, and system resources. This experiment introduces fundamental commands to familiarize students with the UNIX/Linux environment and its practical applications.

Aim:

To learn and execute basic UNIX/Linux commands for file management, process monitoring, text handling, and permissions through real- life use case scenarios.

Algorithm / Procedure:

1.Log in to the UNIX/Linux terminal.
2.Use basic commands such as pwd, ls, mkdir, cd to navigate the file system.
3.Create, copy, move, and delete files as per the use case scenario.
4.Use ps, top, kill, nice, and renice commands to monitor and manage processes.
5.Use text processing commands like cat, grep, wc, sort, uniq, cut, and tail.
6.Modify file permissions using chmod, chown, and umask.
7.Record the commands executed, their outputs, and observations in the observation table.

1A. File and Directory Management in Linux

Use Case Scenario:

You are managing project files for multiple teams in your company.

Commands:

- **pwd**

pwd stands for Print Working Directory.
It shows you the full absolute path of the directory you are currently in.

Syntax:

pwd

Example:

```
administrator@administrator-VirtualBox:~$ pwd
/home/administrator
```

- **mkdir**
  mkdir stands for Make Directory.
  It is used to create new directories (folders) in Linux/Unix systems.

Syntax:

mkdir directory_name

Example:

```
administrator@administrator-VirtualBox:~$ mkdir os
```

IS -l

ls → lists files and directories.

-l → long listing format (gives detailed info).

Syntax:

ls -l

Example:

```
administrator@administrator-VirtualBox:~$ ls -l
total 252
-rw-rw-r-- 1 administrator administrator    86 Oct 13  2016 a.c
-rw-rw-r-- 1 administrator administrator    82 Oct 13  2016 a.cpp
-rw-rw-r-- 1 administrator administrator    71 Nov  3  2017 a_file
-rw-rw-r-- 1 administrator administrator    71 Sep 15  2017 a_file~
drwxrwxr-x 2 administrator administrator  4096 Oct 25  2019 akash1
-rw-rw-r-- 1 administrator administrator   111 Oct 25  2019 akash1.sh
-rw-rw-r-- 1 administrator administrator   115 Oct 25  2019 akash1.sh~
-rw-rw-r-- 1 administrator administrator   345 Oct 13  2017 array2.sh
-rw-rw-r-- 1 administrator administrator   273 Nov  3  2017 array.sh~
-rw-rw-r-- 1 administrator administrator   189 Nov  3  2017 big.sh
-rw-rw-r-- 1 administrator administrator   224 Oct 29  2019 charishma.txt
```

ls -a

ls → lists files and directories.

-a → all files, including hidden ones (. and ..).

If no directory is given, it lists the current directory.

Syntax:

ls -a

Example:

```
administrator@administrator-VirtualBox:~$ ls -a
.               display.sh        .gconf           password.sh~
..              display.sh~       .gksu.lock       pc.c
a.c             .dmrc             .gnome2          pc.c~
a.cpp           Documents         .gtk-bookmarks   Pictures
a_file          Downloads         .gvfs            .profile
a_file~         element.sh        .ICEauthority    Public
akash1          element.sh~       .local           .pulse
akash1.sh       examples.desktop  l.txt            .pulse-cookie
akash1.sh~      factorial.sh      machine.sh       replace.sh
array2.sh       factorial.sh~     machine.sh~      replace.sh~
array.sh~       factor.sh         .mission-control rr.c
.bash_history   factor.sh~        .mozilla         rr.c~
.bash_logout    facto.sh          Music            Templates
.bashrc         fact.sh           n1               test
big.sh          fact.sh~          new_directory    .thumbnails
.cache          fifo.c            op.txt           Videos
charishma.txt   fifo.c.save       op.txt.save      vsm
.config         [file]            os               .Xauthority
.dbus           first.c           pass.sh          .xsession-errors
Desktop         first.c~          pass.sh~         .xsession-errors.old
dir             .fontconfig       password.sh
```

cd

cd = change directory

It is used to move from your current working directory to another directory in the Linux filesystem.

By default, when you open a terminal, you are in your home directory (/home/username).

Syntax:

cd [file name]

Example:

```
administrator@administrator-VirtualBox:~$ cd os
```

Mkdir teamA/docs (nested directory)

mkdir = make directory.

teamA/docs = means create a directory called docs inside an existing directory teamA.

Syntax:

mkdir teamA/docs

Example:

```
administrator@administrator-VirtualBox:~/os$ mkdir os/teamA
```

Cd teamA/docs

cd = change directory

It is used to move from your current working directory to another directory in the Linux filesystem.

Syntax:

Cd os/teamA

Example:

```
administrator@administrator-VirtualBox:~/os$ cd os/teamA
```

Touch report1.txt notes.doc hidden.log

The touch command in Linux is mainly used to:
1. Create empty files
2. Update the timestamp (last modified/last accessed time) of existing files
It's one of the simplest ways to quickly create a new file.

Syntax:

touch [OPTION]... FILE...

OPTION → optional flags (for controlling timestamps, permissions, etc.)

FILE → one or more filenames

Common Examples

1. Create a single empty file

touch file1.txt

Creates file1.txt if it does not exist.

```
administrator@administrator-VirtualBox:~/osl$ touch report1.txt
```

2. Create multiple files at once

touch report1.txt notes.doc hidden.log

Creates all three files together.

```
administrator@administrator-VirtualBox:~/osl$ touch report1.txt notes.doc hidden
.log
```

cp report1.txt ../

cp → copy command

report1.txt → source file

../ → destination (parent directory)

This copies report1.txt into the parent directory without deleting the original.

```
administrator@administrator-VirtualBox:~/osl$ cp report1.txt ../
```

mv notes.doc ../

mv → move command

notes.doc → file to move

../ → destination (parent directory)

This moves notes.doc into the parent directory (removes it from current folder).

```
administrator@administrator-VirtualBox:~/osl$ mv notes.doc ../
```

rm hidden.log

rm → remove (delete)

hidden.log → file name

This permanently deletes the file hidden.log.
No recycle bin — once deleted, it's gone.

```
administrator@administrator-VirtualBox:~/osl$ rm hidden.log
```

find . -name "*.txt"

```
administrator@administrator-VirtualBox:~/os$ find . -name "*.txt"
```

find . -mtime -1

```
administrator@administrator-VirtualBox:~/os$ find . -mtime -1
.
```

1B. Monitoring and Managing Processes

Use Case Scenario:

You are a developer testing an application that spawns multiple processes.

ps -ef # view all processes

```
administrator@administrator-VirtualBox:~/os$ ps -ef
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 09:27 ?        00:00:00 /sbin/init
root           2     0  0 09:27 ?        00:00:00 [kthreadd]
root           3     2  0 09:27 ?        00:00:00 [ksoftirqd/0]
root           5     2  0 09:27 ?        00:00:00 [kworker/0:0H]
root           7     2  0 09:27 ?        00:00:00 [migration/0]
root           8     2  0 09:27 ?        00:00:00 [rcu_bh]
root           9     2  0 09:27 ?        00:00:00 [rcu_sched]
root          10     2  0 09:27 ?        00:00:00 [watchdog/0]
root          11     2  0 09:27 ?        00:00:00 [khelper]
root          12     2  0 09:27 ?        00:00:00 [kdevtmpfs]
root          13     2  0 09:27 ?        00:00:00 [netns]
root          14     2  0 09:27 ?        00:00:00 [writeback]
```

top # monitor processes in real-time

```
administrator@administrator-VirtualBox:~/os$ top

top - 09:39:13 up 11 min,  2 users,  load average: 0.00, 0.03, 0.05
Tasks: 146 total,   1 running, 145 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  0.3%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   5109448k total,   596044k used,  4513404k free,    44768k buffers
Swap:  2095100k total,        0k used,  2095100k free,   275152k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1095 root      20   0 94004  38m 9644 S  0.7  0.8   0:04.06 Xorg
 1458 nobody    20   0  5368 1380 1188 S  0.3  0.0   0:00.05 dnsmasq
    1 root      20   0  3528 1912 1276 S  0.0  0.0   0:00.58 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S  0.0  0.0   0:00.35 ksoftirqd/0
    5 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 kworker/0:0H
    7 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
    8 root      20   0     0    0    0 S  0.0  0.0   0:00.00 rcu_bh
    9 root      20   0     0    0    0 S  0.0  0.0   0:00.79 rcu_sched
   10 root      RT   0     0    0    0 S  0.0  0.0   0:00.02 watchdog/0
   11 root       0 -20     0    0    0 S  0.0  0.0   0:00.00 khelper
   12 root      20   0     0    0    0 S  0.0  0.0   0:00.00 kdevtmpfs
```

sleep 100 & # start a dummy background process

```
administrator@administrator-VirtualBox:~/os$ sleep 100 &
[1] 2245
```

ps -ef | grep sleep # find process by name

```
administrator@administrator-VirtualBox:~/os$ ps -ef|grep sleep
1000      2245  1959  0 09:40 pts/0    00:00:00 sleep 100
1000      2251  1959  0 09:40 pts/0    00:00:00 grep --color=auto sleep
```

kill # kill process using PID

```
administrator@administrator-VirtualBox:~/os$ kill
kill: usage: kill [-s sigspec | -n signum | -sigspec] pid | jobspec ... or kill -
l [sigspec]
```

nice -n 10 ./task # run program with low priority

```
administrator@administrator-VirtualBox:~$ nice -n 10 ./hello.sh
hello world
```

renice -n -5 -p # change priority of process

```
administrator@administrator-VirtualBox:~/os$ renice -n -5 -p
```

1C. Text Processing and Filtering

Use Case Scenario:

You are analyzing server log files.

cat server.log # view log

```
administrator@administrator-VirtualBox:~$ chmod +x evenodd.shh
administrator@administrator-VirtualBox:~$ ./evenodd.shh
enter a number:
6
6 is even
administrator@administrator-VirtualBox:~$ ./evenodd.shh>server.log

administrator@administrator-VirtualBox:~$ cat server.log
enter a number:
 is even
```

grep "ERROR" server.log # extract error lines

```
administrator@administrator-VirtualBox:~$ grep "error" server.log
```

grep "ERROR" server.log | wc -l # count errors

```
administrator@administrator-VirtualBox:~$ grep "error" server.log|wc -l
0
```

sort server.log | uniq # unique entries

```
administrator@administrator-VirtualBox:~$ sort server.log|uniq
enter a number:
 is even
```

cut -d ":" -f1 server.log # extract first field

```
administrator@administrator-VirtualBox:~$ cut -d "t" -f1 server.log
en
 is even
```

tail -n 5 server.log # last 5 lines

```
administrator@administrator-VirtualBox:~$ tail -n2 server.log
enter a number:
 is even
```

1D. File Permissions and Ownership

Use Case Scenario:

You are a system administrator responsible for sensitive files.

touch secret.txt # create file

```
administrator@administrator-VirtualBox:~/osl$ touch secret.txt
```

ls -l secret.txt # check permissions

```
administrator@administrator-VirtualBox:~/osl$ ls -l secret.txt
-rw-rw-r-- 1 administrator administrator 0 Aug 26 10:52 secret.txt
```

chmod 600 secret.txt # only owner can read/write

```
administrator@administrator-VirtualBox:~/osl$ chmod 600 secret.txt
```

chmod u+x secret.txt # add execute permission

```
administrator@administrator-VirtualBox:~/osl$ chmod u+x secret.txt
```

chmod a-r secret.txt # remove read permission for all

```
administrator@administrator-VirtualBox:~/osl$ chmod a-r secret.txt
```

chown user1: team secret.txt # change owner and group

umask 077 # set default restrictive mask

```
administrator@administrator-VirtualBox:~/osl$ umask 077
```

Result:

Thus, the basic UNIX/Linux commands for file management, process monitoring, text processing, and permissions were executed successfully, and their outputs were verified.

Experiment 2: Shell Programming

Introduction:

A shell in UNIX/Linux is a command interpreter that provides a user interface to the operating system. It can execute commands directly and allows users to automate tasks by writing shell scripts. A shell script is a file containing a sequence of commands, loops, and conditional statements. Shell programming is useful for automating repetitive tasks such as backups, report generation, and system monitoring.

Aim:

To write and execute shell scripts for basic automation tasks such as displaying messages, using variables, conditional statements, and loops.

Algorithm / Procedure:

1. Open a text editor (vi/nano) and create a new shell script file with .sh extension.
2. Write the script starting with the shebang line #!/bin/bash.
3. Use variables, echo statements, conditional statements, and loops as per the requirement
4. Save the file and exit the editor.
5. Make the file executable using chmod +x filename.sh.
6. Run the script using ./filename.sh.
7. Observe the output and record it in the observation table.

Sub-Experiments with Use Case Scenarios

2A. Display "Hello World"

Question (Use Case Scenario):

A system administrator wants to quickly test whether a newly written shell script executes correctly by displaying a simple confirmation message.

Program:

print("Hello, World!")

```
print("Hello, World!")
```

Output:

```
Hello, World!
```

2B. Sum of Two Numbers

Question (Use Case Scenario):

A finance trainee wants to quickly calculate totals of two given values using a shell script instead of a calculator.

Program:

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
sum = a + b
print("The sum is:", sum)
```

```
# Take input from the user
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

# Calculate sum
sum = a + b

# Display result
print("The sum is:", sum)
```

Output:

```
Enter first number: 5
Enter second number: 7
The sum is: 12
```

2C. Check Whether a Number is Even or Odd

Question (Use Case Scenario):

A student wants to verify whether their roll number is even or odd using a simple shell script.

Program:

```
num = int(input("Enter a number: "))

if num % 2 == 0:
    print("The number is Even")
else:
    print("The number is Odd")
```

```
num = int(input("Enter a number: "))

if num % 2 == 0:
    print("The number is Even")
else:
    print("The number is Odd")
```

Output:

```
Enter a number: 7
The number is Odd
```

2D. Print Factorial of a Number

Question (Use Case Scenario):

A researcher needs to calculate factorial values for mathematical or statistical computations using automation.

Program:

```
num = int(input("Enter a number: "))
factorial = 1

for i in range(1, num + 1):
    factorial *= i

print("Factorial is:", factorial)
```

```
num = int(input("Enter a number: "))
factorial = 1

for i in range(1, num + 1):
    factorial *= i

print("Factorial is:", factorial)
```

Output:

```
Enter a number: 5
Factorial is: 120
```

2E. Print Fibonacci series of a Number

Question (Use Case Scenario):

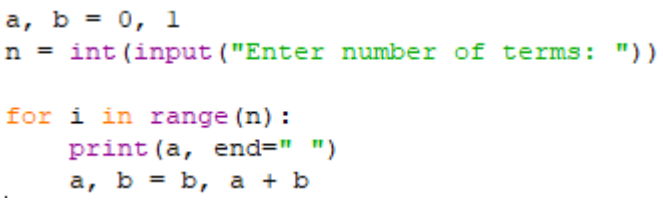
A biologist is studying how a population of rabbits grows, where each new generation equals the sum of the previous two generations.
You decide to use your Fibonacci shell script to model this.

What input (limit) would you give if you want to see the rabbit population growth over 8 generations?

How would you interpret each number in the series in this context?

Program:

```
a, b = 0, 1
n = int(input("Enter number of terms: "))

for i in range(n):
    print(a, end=" ")
    a, b = b, a + b
```



Output:

```
Enter number of terms: 7
0 1 1 2 3 5 8
```

2F. Print the Biggest from the given three Numbers

Question (Use Case Scenario):

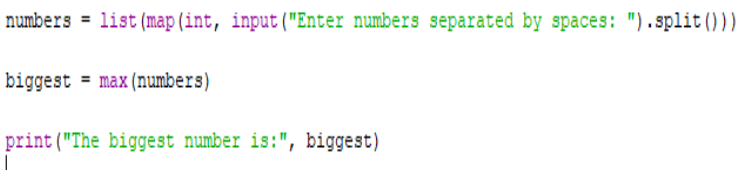
A car dealer wants to know which of three car models gives the best mileage.
They use the script to enter mileage (km/litre).

Algorithm:

1. Start
2. Input: Ask the user to enter a list of numbers separated by spaces.
3. Convert Input: Split the input string into individual values and convert each value to an integer. Store these integers in a list called `numbers`.
4. Find the Biggest Number:
   Use the built-in `max()` function to find the largest number in the list.
   Store this number in a variable called `biggest`.
   Output: Print the value of `biggest` as the largest number.
5. End

Program:

```
numbers = list(map(int, input("Enter numbers separated by spaces: ").split()))

biggest = max(numbers)

print("The biggest number is:", biggest)
```



Output:

```
Enter numbers separated by spaces: 6 9 3
The biggest number is: 9
```

2G. Print the Biggest from the given two Numbers

Question (Use Case Scenario):

A car dealer wants to know which of three car models gives the best mileage.
They use the script to enter mileage (km/litre).

Algorithm:

1. Start
2. Input: Ask the user to enter a list of numbers separated by spaces.
3. Convert Input:
   Split the input string into individual values.
   Convert each value to an integer.
   - Store these integers in a list called `numbers`.
4. Initialize:
   - Assume the first number in the list is the biggest.
   - Store it in a variable called `biggest`.
5. Compare Each Number:
   For each number `num` in the list:
   If `num` is greater than `biggest`:
   Update `biggest` to be `num`.
6. Output: Print the value of `biggest` as the largest number.
7. End

Program:

```
numbers = list(map(int, input("Enter numbers separated by spaces: ").split()))

biggest = max(numbers)

print("The biggest number is:",
biggest)
```

```
numbers = list(map(int, input("Enter numbers separated by spaces: ").split()))

biggest = numbers[0]

for num in numbers:
    if num > biggest:
        biggest = num

print("The biggest number is:", biggest)
```

Output:

```
Enter numbers separated by spaces: 7 9
The biggest number is: 9
```

Result:

Thus, shell scripts for printing messages, arithmetic operations, conditional checks, fibonacci, biggest number and loops were successfully written, executed, and outputs verified

Experiment 3: GREP, SED, AWK Experiments

Use Case Scenario:

You are working as a System Administrator in the Computer Science Engineering Department. You are given different log files (e.g., students.txt, server_logs.txt, employee.txt). You need to use grep, sed, and awk to:

1. Extract specific information (error logs, student records).
2. Modify data (replace, delete, insert).
3. Generate simple reports (salary lists, student marks).

Aim:

To study and implement the use of GREP, SED, and AWK commands in Linux for pattern searching, text substitution, and text processing.

Algorithm:

1. Using `grep` Command

1. Open the terminal.
2. Create a sample file named file.txt and enter the provided data.
   Run the following commands:
   - `grep "hello" file.txt` → Search for a word in a file.
   - `grep -i "hello" file.txt` → Case-insensitive search.
   - `grep -r "hello" /path/to/directory/` → Recursive search in directories.
   - `grep -v "hello" file.txt` → Display lines not containing the word.
   - `grep -c "hello" file.txt` → Count lines matching the pattern.

2. Using `sed` Command

1. Open the same sample file (file.txt).
2. Execute the following commands:
   `sed 's/old/new/' file.txt` → Replace first occurrence in each line.
   `sed 's/old/new/g' file.txt` → Replace all occurrences globally.
   - `sed '/pattern/d' file.txt` → Delete lines containing a pattern.
   - `sed '/^$/d' file.txt` → Delete blank lines.
   - `sed -i 's/old/new/g' file.txt` → Edit and save the file directly.

3. Using `awk` Command

1. Use awk for pattern-based text processing.
2. Execute the following examples:
    - `awk '{print $1}' file.txt` → Print the first column.
    - `awk '{print $1, $3}' file.txt` → Print first and third columns.
    - `awk -F',' '{print $1}' file.txt` → Use comma as field separator.
    - `awk '$1 ~ /pattern/ {print $0}' file.txt` → Print lines where the first field matches a pattern.
    - `awk '{sum += $2} END {print sum}' file.txt` → Sum the values in the second column.

1. grep — Search for Patterns in Text

grep stands for "Global Regular Expression Print." It's used to search for specific patterns within files or input streams.

Basic Syntax:
grep [options] pattern [file...]

Examples:
Search for a word in a file:
grep "hello" file.txt

This command prints all lines in file.txt that contain the word "hello".

```
administrator@administrator-VirtualBox:~/72$ grep "john" file.txt
john,25,developer
2025-09-01,john,apple,2
```

Case-insensitive search:
grep -i "hello" file.txt

This command searches for "hello", "Hello", "HELLO", etc., ignoring case differences.

```
administrator@administrator-VirtualBox:~/72$ grep -i "john" file.txt
john,25,developer
2025-09-01,john,apple,2
```

Search recursively in directories:
grep -r "hello" /path/to/directory/

This searches all files in the specified directory and its subdirectories for the word "hello".

Invert match (show lines that do not match):
grep -v "hello" file.txt

This command prints all lines in file.txt that do not contain the word "hello".
Count the number of matching lines:

```
administrator@administrator-VirtualBox:~/72$ grep -v "john" file.txt
#User Data
alice,30,designer
bob,22,developer
eve,28,manager

#product List
apple,1.2
banana,0.5
cherry,2.5
date,3.0

#Sales Records
2025-09-01,alice,banana,3
2025-09-02,bob,apple,1
2025-09-02,eve,cherry,5
```

grep -c "hello" file.txt

This command displays the number of lines in file.txt that contain the word "hello".

```
administrator@administrator-VirtualBox:~/72$ grep -c "john" file.txt
2
```

2. sed — Stream Editor for Modifying Text

sed is a stream editor that allows you to perform basic text transformations on an input stream (a file or input from a pipeline).

Basic Syntax:
sed [options] 'command' [file...]

Examples:

Replace text in a file:
sed 's/old/new/' file.txt

This replaces the first occurrence of "old" with "new" in each line of file.txt.
Replace all occurrences in a line:

```
administrator@administrator-VirtualBox:~/72$ sed 's/john/johns/' file.txt
#User Data
johns,25,developer
alice,30,designer
bob,22,developer
eve,28,manager

#product List
apple,1.2
banana,0.5
cherry,2.5
date,3.0

#Sales Records
2025-09-01,johns,apple,2
2025-09-01,alice,banana,3
2025-09-02,bob,apple,1
2025-09-02,eve,cherry,5
```

sed 's/old/new/g' file.txt

The g at the end makes the replacement global (all occurrences in the line).

```
administrator@administrator-VirtualBox:~/72$ sed 's/john/johns/g' file.txt
#User Data
johns,25,developer
alice,30,designer
bob,22,developer
eve,28,manager

#product List
apple,1.2
banana,0.5
cherry,2.5
date,3.0

#Sales Records
2025-09-01,johns,apple,2
2025-09-01,alice,banana,3
2025-09-02,bob,apple,1
2025-09-02,eve,cherry,5
```

Delete lines containing a specific pattern:

sed '/pattern/d' file.txt

This deletes all lines in file.txt that contain "pattern".

```
administrator@administrator-VirtualBox:~/72$ sed '/apple/d' file.txt
#User Data
john,25,developer
alice,30,designer
bob,22,developer
eve,28,manager

#product List
banana,0.5
cherry,2.5
date,3.0

#Sales Records
2025-09-01,alice,banana,3
2025-09-02,eve,cherry,5
```

Delete blank lines:

sed '/^$/d' file.txt

This deletes all blank lines from file.txt.

```
administrator@administrator-VirtualBox:~/72$ sed '/^$/d' file.txt
#User Data
john,25,developer
alice,30,designer
bob,22,developer
eve,28,manager
#product List
apple,1.2
banana,0.5
cherry,2.5
date,3.0

#Sales Records
2025-09-01,john,apple,2
2025-09-01,alice,banana,3
2025-09-02,bob,apple,1
2025-09-02,eve,cherry,5
```

Replace text in a file and save changes:

sed -i 's/old/new/g' file.txt

The -i option edits the file in place, saving the changes directly to file.txt.

```
administrator@administrator-VirtualBox:~/72$ sed -i 's/john/johns/g' file.txt
```

3. awk — Pattern Scanning and Processing Language

awk is a powerful programming language for pattern scanning and processing. It is used for processing and analyzing text files, especially those with structured data.

Basic Syntax:
awk 'pattern {action}' [file...]

Examples:
Print the first column of a file:
awk '{print $1}' file.txt

This prints the first field (column) of each line in file.txt.

```
administrator@administrator-VirtualBox:~/72$ awk '{print $1}' file.txt
#User
johns,25,developer
alice,30,designer
bob,22,developer
eve,28,manager

#product
apple,1.2
banana,0.5
cherry,2.5
date,3.0

#Sales
2025-09-01,johns,apple,2
2025-09-01,alice,banana,3
2025-09-02,bob,apple,1
2025-09-02,eve,cherry,5
```

Print the first and third columns:
awk '{print $1, $3}' file.txt

This prints the first and third fields of each line in file.txt.

```
administrator@administrator-VirtualBox:~/72$ awk '{print $1,$3}' file.txt
#User
johns,25,developer
alice,30,designer
bob,22,developer
eve,28,manager

#product
apple,1.2
banana,0.5
cherry,2.5
date,3.0

#Sales
2025-09-01,johns,apple,2
2025-09-01,alice,banana,3
2025-09-02,bob,apple,1
2025-09-02,eve,cherry,5
```

Use a comma as a field separator:

awk -F',' '{print $1}' file.csv

This sets the field separator to a comma and prints the first field of each line in file.csv.

```
administrator@administrator-VirtualBox:~/72$ awk -F',' '{print $1}' file.csv
#User Data
john
alice
bob
eve
```

Print lines where the first field matches a pattern:

awk '$1 ~ /pattern/ {print $0}' file.txt

This prints lines where the first field matches "pattern".

```
administrator@administrator-VirtualBox:~/72$ awk '$1 ~ /john/ {print $0}' file.t
xt
johns,25,developer
2025-09-01,johns,apple,2
```

Sum the values in the second column:

awk '{sum += $2} END {print sum}' file.txt

This sums all the values in the second column of file.txt and prints the total.

```
administrator@administrator-VirtualBox:~/72$ awk '{sum +=$2} END {print sum}' fi
le.txt
0
```

****

Sample File for execute all of above commands:

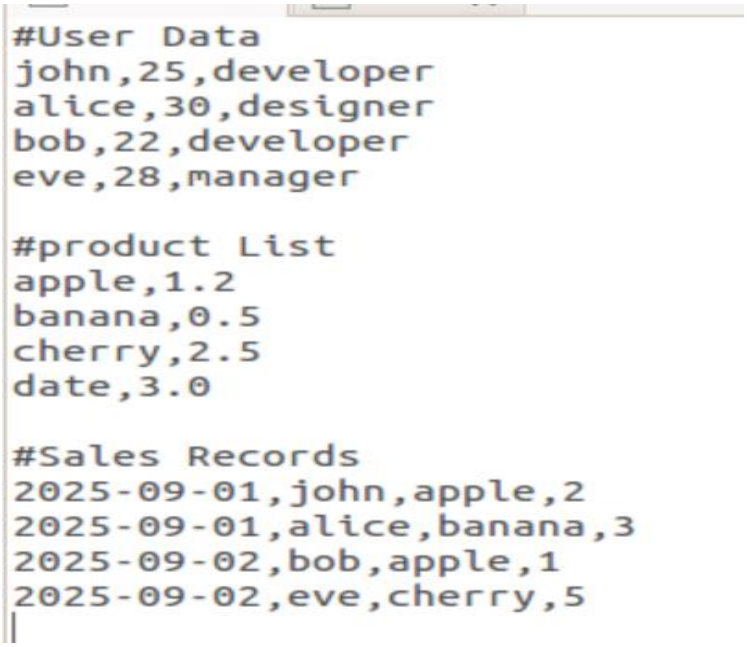
Filename : file.txt

(Type the following code in your Text Editor and Save the filename as file.txt)

```
# User Data
john,25,developer
alice,30,designer
bob,22,developer
eve,28,manager

# Product List
apple,1.2
```

banana,0.5
cherry,2.5
date,3.0

Sales Records
2025-09-01,john,apple,2
2025-09-01,alice,banana,3
2025-09-02,bob,apple,1
2025-09-02,eve,cherry,5



```
#User Data
john,25,developer
alice,30,designer
bob,22,developer
eve,28,manager

#product List
apple,1.2
banana,0.5
cherry,2.5
date,3.0

#Sales Records
2025-09-01,john,apple,2
2025-09-01,alice,banana,3
2025-09-02,bob,apple,1
2025-09-02,eve,cherry,5
```

Result:

The experiment successfully demonstrated the use of:

- `grep` for searching and filtering text using patterns,
- `sed` for stream editing and text substitution, and
- `awk` for advanced text processing and data manipulation in Linux.

Experiment 4: System Calls

(a) stat () System Call

Aim:

To write a C program using stat() system call to obtain file properties such as size, permissions, and last modified time.

Use-Case Scenario:

Before attaching a file to an email, your program must check its size and details to ensure it meets requirements.

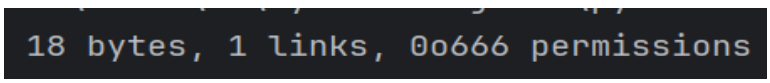
Algorithm:

• Start program.
• Declare struct stat variable.
• Use stat(filename, &statbuf) to fetch file details.
• Display file size, number of links, owner ID, and permissions.
• Stop.

Program:

```
import os
s = os.stat("test.txt")
print(s.st_size, "bytes,", s.st_nlink, "links,", oct(s.st_mode & 0o777), "permissions")
```

Output:



Result:

The program successfully displayed file information using stat() system call.

(b) wait() System Call

Aim:
To implement process synchronization using fork() and wait() system calls.

Use-Case Scenario:
An installer (parent) must wait until a child process finishes installing a module before proceeding.

Algorithm:
• Call fork() to create a child process.
• If the parent, call wait() to pause until child completes.
• If a child, print a message and exit.
• The parent resumes after the child finishes.

Program:
```
import multiprocessing
import time

def child_process():
    print("Child running...")
    time.sleep(2)
    print("Child done.")

if __name__ == "__main__":
    child = multiprocessing.Process(target=child_process)
    child.start()          # Start child process
    child.join()           # Wait for child to finish
    print("Parent continues after child.")
```


```
import multiprocessing
import time

def child_process():
    print("Child running...")
    time.sleep(2)
    print("Child done.")

if __name__ == "__main__":
    child = multiprocessing.Process(target=child_process)
    child.start()          # Start child process
    child.join()           # Wait for child to finish
    print("Parent continues after child.")
```


Output:
```
Child running...
Child done.
Parent continues after child.
```

Result:
The program demonstrated synchronization using wait().

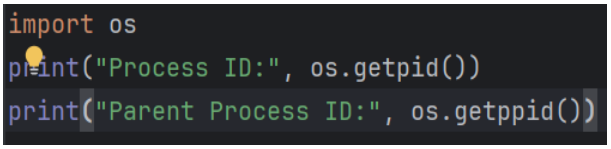
(c) getpid() System Call

Aim:
To write a program that retrieves the process ID (PID) and parent process ID (PPID).

Use-Case Scenario:
In debugging and monitoring, processes must know their IDs for logging.

Algorithm:
• Call getpid() to get process ID.
• Call getppid() to get parent ID.
• Print both values.

Program:
import os
print("Process ID:", os.getpid())
print("Parent Process ID:", os.getppid())



Output:

```
Process ID: 6028
Parent Process ID: 10808
```

Result:
The program successfully printed process IDs using getpid() and getppid().

(d) opendir(), readdir() System Calls

Aim:
To write a program to display contents of a directory using opendir() and readdir().

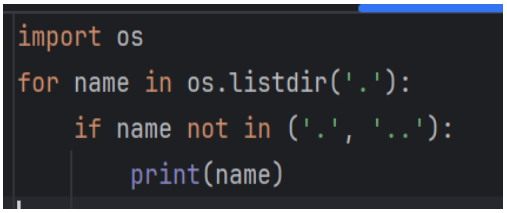
Use-Case Scenario:
A file explorer must list all files inside a folder.

Algorithm:
• Prompt user for a directory name.
• Call opendir() to open directory.

• Use readdir() in loop to read file entries.
• Print names of files.

Program:

```
import os
for name in os.listdir('.'):
    if name not in ('.', '..'):
        print(name)
```



Output:

```
.git
.idea
.venv
CIRCLE.py
circle1.py
exp4.py
file1.txt
main.py
new2file.txt
pan.py
partice.py
poreddy.py
rani.csv
test.txt
```

Result:

The program displayed directory contents using system calls.

(e) open() System Call

Aim:

To create and open a file using open() system call.

Use-Case Scenario:

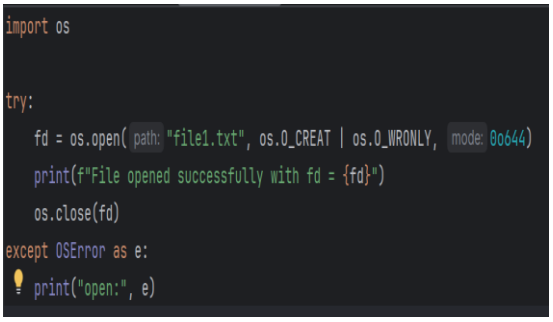
Before writing to a log, a program must open or create the file.

Algorithm:

• Call open(filename, flags, mode) to open or create file.
• Check if file descriptor is valid.
• Close file.

Program:

```
import os

try:
    fd = os.open("file1.txt", os.O_CREAT |
os.O_WRONLY, 0o644)
    print(f"File opened successfully with fd = {fd}")
    os.close(fd)
except OSError as e:
    print("open:", e)
```



Output:

```
File opened successfully with fd = 3
```

Result:

File was successfully opened/created using open() system call.

(f) File I/O System Calls: open(), read(), write(), lseek()

Aim:

To implement file I/O using low-level UNIX system calls.

Use-Case Scenario:

A program must log data to a file, then read it back for verification.

Algorithm:

- Open file using open().
- Write text using write().
- Use lseek() to move file pointer.
- Read content using read().
- Print the read content.
- Close file.

Program:

```
import os

fd = os.open("new2file.txt", os.O_CREAT | os.O_RDWR, 0o644)
os.write(fd, b"Hello World")
```

```
os.lseek(fd, 0, os.SEEK_SET)
data = os.read(fd, 50)
print("Read from file:", data.decode())
os.close(fd)
```

```
import os

fd = os.open( path: "new2file.txt", os.O_CREAT | os.O_RDWR, mode: 0o644)
os.write(fd, data: b"Hello World")
os.lseek(fd, position: 0, os.SEEK_SET)
data = os.read(fd, length: 50)
print("Read from file:", data.decode())
os.close(fd)
```

Output:

```
Read from file: Hello World
```

Result:

The program demonstrated file creation, writing, reading, and seeking using system calls.

Experiment 5: Process Management

a.fork()

Aim:

To write a Python program to create a new process using the `multiprocessing` module (an alternative to `os.fork()` which works on Windows).

Algorithm:

1. Import the required modules: `multiprocessing` and `os`.
2. Define a function (`child_process`) that prints its process ID (PID) and parent process ID (PPID).
3. In the main function:
    - Create a Process object, passing the child function as the target.
    - Start the child process using `start()`.
      Print the parent process ID and the child's process ID.
    - Wait for the child process to complete using `join()`.
4. Execute the main function inside the `if __name__ == "__main__":` block.

Program:

```
from multiprocessing import Process

import os


def child_process():

    print(f"Child process: PID = {os.getpid()}, Parent PID = {os.getppid()}")
```

```
def main():
    p = Process(target=child_process)
    p.start()  # start child process
    print(f"Parent process: PID = {os.getpid()}, Child PID = {p.pid}")
    p.join()   # wait for the child to finish

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

```
from multiprocessing import Process
import os

def child_process():
    print(f"Child process: PID = {os.getpid()}, Parent PID = {os.getppid()}")

def main():
    p = Process(target=child_process)
    p.start()  # start child process
    print(f"Parent process: PID = {os.getpid()}, Child PID = {p.pid}")
    p.join()   # wait for the child to finish

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

Output:

```
Parent process: PID = 10824, Child PID = 2372
```

Result:

The program successfully demonstrates process creation in Python using the `multiprocessing` module.

B. EXEC()

Aim:

To demonstrate process creation in Python using the `multiprocessing` module on Windows.

Algorithm:

1. Import `multiprocessing`, `os`, and `subprocess`.
2. Define `child_process()` that executes a system command (`dir`).
3. In `main()`:
    - Print a message for the parent process.
    - Create and start a child process.
    - Wait for the child to finish using `.join()`.
4. Print a message after the child completes.

Program:

```
import multiprocessing
import subprocess
import os

def child_process():
    # Equivalent of 'ls -l' on Windows
    subprocess.run(["cmd", "/c", "dir"])
    print("\nChild process: Finished running 'dir' command.")

def main():
    print("\nParent Process")
    p = multiprocessing.Process(target=child_process)
    p.start()   # Start the child process
    p.join()    # Wait for the child process to finish
    print("\nParent process: Child completed.")

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

import multiprocessing

import subprocess

import os

def child_process():

Equivalent of 'ls -l' on Windows

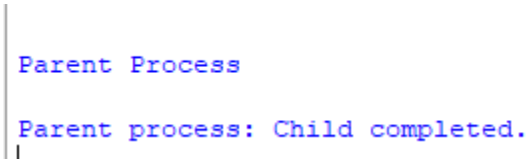
subprocess.run(["cmd", "/c", "dir"])

print("\nChild process: Finished running 'dir' command.")

def main():

```
    print("\nParent Process")
    p = multiprocessing.Process(target=child_process)
    p.start()  # Start the child process
    p.join()   # Wait for the child process to finish
    print("\nParent process: Child completed.")

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

Output:



Result:

The program successfully demonstrates process creation on Windows using `multiprocessing`.
The child process executes the `dir` command, and the parent waits until it finishes.

Experiment 6A: CPU Scheduling Algorithm FIFO

Use Case Scenario:

In a real-world system such as a print server or batch processing system, tasks are often executed in the order they arrive. The First In First Out (FIFO) scheduling algorithm models this behaviour by serving each process sequentially based on arrival time. This ensures fairness and predictability; though longer jobs can delay smaller ones. For example, in a print queue, the first document submitted will be printed first, even if it is very large.

Aim:

To write a program to implement FIFO CPU scheduling algorithm

Algorithm:

1.Start
2.Input the number of processes `n`.
3.For each process `i`, read its Burst Time (BT).
4.Set the Waiting Time (WT) for the first process as `0`.
5.For each process `i` from 2 to `n`:

- `WT[i]=WT[i-1]+BT[i-1]`

6.Calculate Turnaround Time (TAT) for each process:

- `TAT[i] = WT[i] + BT[i]`

7.Compute

- Average Waiting Time = (Sum of all WT) / n
- Average Turnaround Time = (Sum of all TAT) / n

8.Display all Waiting Times, Turnaround Times, and Averages.
9.Stop

Program:

```
n = int(input("Enter number of processes: "))
bt = [int(input(f"Enter Burst Time for P{i+1}: ")) for i in range(n)]
```

```
wt, tat = [0]*n, [0]*n

for i in range(1, n):
    wt[i] = wt[i-1] + bt[i-1]
    tat[i] = wt[i] + bt[i]
tat[0] = bt[0]

print("\nProcess\tBT\tWT\tTAT")
for i in range(n):
    print(f"P{i+1}\t{bt[i]}\t{wt[i]}\t{tat[i]}")

print("\nAverage Waiting Time:", sum(wt)/n)
print("Average Turnaround Time:", sum(tat)/n)
```

```
n = int(input("Enter number of processes: "))
bt = [int(input(f"Enter Burst Time for P{i+1}: ")) for i in range(n)]

wt, tat = [0]*n, [0]*n

for i in range(1, n):
    wt[i] = wt[i-1] + bt[i-1]
    tat[i] = wt[i] + bt[i]
tat[0] = bt[0]

print("\nProcess\tBT\tWT\tTAT")
for i in range(n):
    print(f"P{i+1}\t{bt[i]}\t{wt[i]}\t{tat[i]}")

print("\nAverage Waiting Time:", sum(wt)/n)
print("Average Turnaround Time:", sum(tat)/n)
```

Output:

```
Enter number of processes: 3
Enter Burst Time for P1: 4
Enter Burst Time for P2: 5
Enter Burst Time for P3: 8

Process BT   WT   TAT
P1   4    0    4
P2   5    4    9
P3   8    9    17

Average Waiting Time: 4.333333333333333
Average Turnaround Time: 10.0
```

Result:

Thus, the CPU scheduling algorithm (FIFO) was executed successfully.

Experiment 6B: CPU Scheduling Algorithm Round Robin

Use Case Scenario:

In multitasking environments like time-sharing systems or operating systems handling multiple users, Round Robin (RR) scheduling ensures that each process receives an equal share of CPU time. It is widely used in real-time systems where responsiveness is critical. For example, in a web server handling multiple client requests, each request (process) is given a time slice (quantum) to ensure that all clients experience smooth performance without one monopolizing system resources.

Aim:

To write a program to implement Round Robin CPU scheduling algorithm

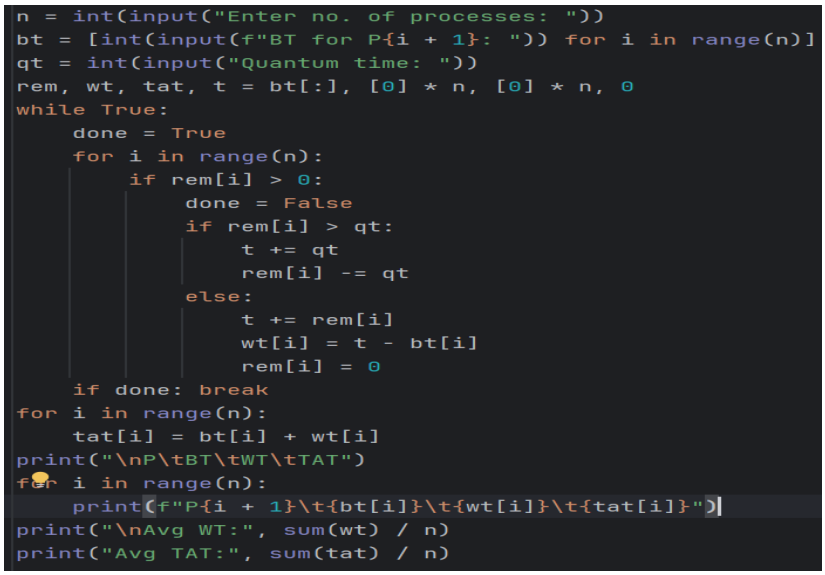
Algorithm:

1. Read the number of processes and their burst times.
2. Read the quantum time.
3. Initialize waiting time and turnaround time for each process to zero.
4. Repeat until all processes are completed:
    - For each process, if its burst time is greater than 0:
        - If burst time ≥ quantum time, reduce burst time by quantum time.
        - Else, reduce burst time to 0.
        - Update waiting and turnaround times accordingly.
5. Compute average waiting time and average turnaround time.
6. Display results.

Program:

```
n = int(input("Enter no. of processes: "))

bt = [int(input(f"BT for P{i+1}: ")) for i in range(n)]

qt = int(input("Quantum time: "))



rem, wt, tat, t = bt[:], [0]*n, [0]*n, 0
```

```
while True:
    done = True
    for i in range(n):
        if rem[i] > 0:
            done = False
            if rem[i] > qt:
                t += qt
                rem[i] -= qt
            else:
                t += rem[i]
                wt[i] = t - bt[i]
                rem[i] = 0
    if done: break

for i in range(n):
    tat[i] = bt[i] + wt[i]

print("\nP\tBT\tWT\tTAT")
for i in range(n):
    print(f"P{i+1}\t{bt[i]}\t{wt[i]}\t{tat[i]}")

print("\nAvg WT:", sum(wt)/n)
print("Avg TAT:", sum(tat)/n)
```



Output:

```
Enter no. of processes: 3
BT for P1: 2
BT for P2: 3
BT for P3: 5
Quantum time: 2

P   BT  WT  TAT
P1  2   0   2
P2  3   4   7
P3  5   5   10

Avg WT: 3.0
Avg TAT: 6.333333333333333
```

Result:

Thus, the CPU scheduling algorithm (Round Robin) was executed successfully.

Experiment 7: Banker's Algorithm for Deadlock Avoidance

Use Case Scenario:

Consider a banking system where:
Each customer (process) has a maximum loan limit (MAX).
Each has already borrowed a certain amount (Allocation).
The bank has a limited amount of money (Available).
Before granting a new loan (resource), the bank ensures that it can still satisfy the maximum demand of all customers.
If yes, the system remains in a safe state — no customer will be indefinitely waiting (no deadlock).

Aim:

To implement the Banker's Algorithm for Deadlock Avoidance and determine whether the system is in a safe state or not.

Objective:

To understand how the Banker's Algorithm works to avoid deadlocks in a multiprocess system.
To compute the safe sequence of process execution if the system is in a safe state.

Theory:

In a multi-programming environment, several processes compete for limited resources such as CPU, memory, and I/O devices.
A deadlock occurs when a group of processes are permanently blocked because each process is holding a resource and waiting for another one.

The Banker's Algorithm, introduced by Edsger Dijkstra, helps the operating system avoid deadlock by ensuring that resource allocation always leaves the system in a safe state.

A safe state means that there exists a sequence of all processes such that each process can obtain its maximum required resources, execute, and release resources without causing a deadlock.

Data Structures Used

Variable	Description
ins[]	Total number of instances for each resource type
avail[]	Available resources at a given time
allocated[][]	Allocation matrix showing resources currently allocated to each process
MAX[][]	Maximum demand matrix
need[][]	Resources still needed (MAX - ALLOCATED)
output[]	Stores the safe sequence of processes

Algorithm (Banker's Algorithm):

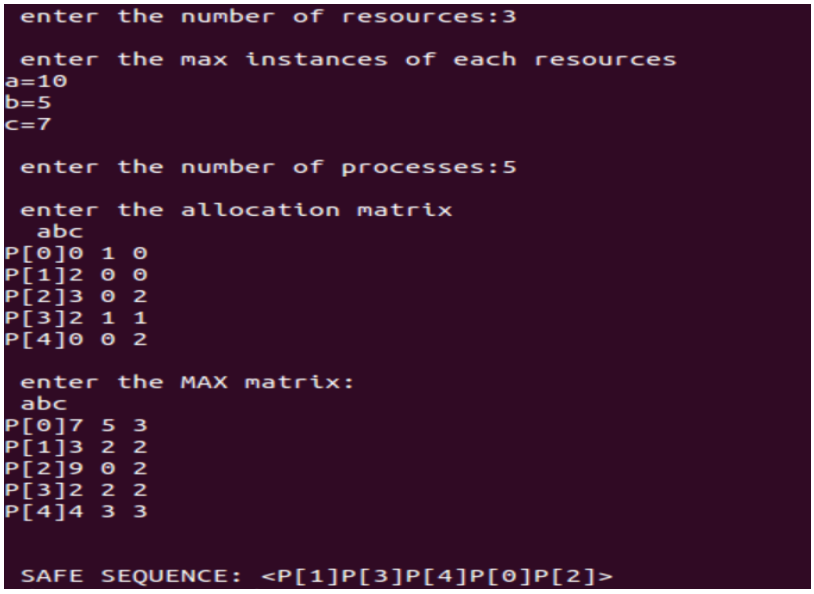
1. Input Data o Number of processes n o Number of resources m o Total resources (ins[m]), Allocation[n][m], and MAX[n][m]
2. Calculate Available[j] = Total[j] - Σ Allocation[i][j] Need[i][j] = MAX[i][j] - Allocation[i][j]
3. Find a process Pi such that for all j, Need[i][j] <= Available[j].
4. If such a process is found: o Add Pi to the safe sequence. o Update Available[j] = Available[j] + Allocation[i][j]. o Mark process Pi as finished.
5. Repeat until all processes are finished or no further process can proceed.
6. If all processes finish → Safe State Else → System is in Deadlock (Unsafe State)

Program:

```
r = int(input("Resources: "))
ins = [int(input(f"{chr(97+i)}: ")) for i in range(r)]
p = int(input("Processes: "))

print("Allocation:")
alloc = [list(map(int, input(f"P{i}: ").split())) for i in range(p)]
print("Max:")
maxm = [list(map(int, input(f"P{i}: ").split())) for i in range(p)]

avail = [ins[j] - sum(alloc[i][j] for i in range(p)) for j in range(r)]
need = [[maxm[i][j] - alloc[i][j] for j in range(r)] for i in range(p)]

safe, done = [], [0]*p
while len(safe) < p:
    for i in range(p):
        if not done[i] and all(need[i][j] <= avail[j] for j in range(r)):
            avail = [avail[j] + alloc[i][j] for j in range(r)]
            safe.append(i)
            done[i] = 1
            break
    else:
        print("Not safe state")
        exit()
print("Safe sequence:", "<", " ".join(f"P{i}" for i in safe), ">")
```

```
        print("Not safe state")
        exit()
print("Safe sequence:", "<", " ".join(f"P{i}" for i in safe), ">")
```

Output:



Result:

The program successfully determines the safe sequence of process execution using the Banker's Algorithm. Hence, the system is in a safe state, and deadlock is avoided.

Experiment 8: PROCESS SYNCHRONIZATION

Write a program to solve the Producer Consumer Problem using Semaphores

Use Case Scenario:

In a factory system, a **producer** creates products and places them into a **buffer**, while a **consumer** takes them out for processing. The **buffer has limited capacity**, so the producer must wait if it's full, and the consumer must wait if it's empty.

Semaphores are used to synchronize both processes so they don't access the buffer at the same time.

Aim:

To write a program to solve the producer–consumer problem using semaphores.

Algorithm:

1. Start
2. First segment is for Producer:
    - (a) Get count of items to be produced.
    - (b) Add count with current buffer size.
    - (c) If new buffer is full → don't produce anything (Producer has to wait until consumer consumes some items).
    - (d) Enter the item and increment buffer count.
3. Second segment is for Consumer:
    - (a) Get count of required items to be consumed.
    - (b) Subtract count from current buffer size.
    - (c) If new buffer size < 0 → cannot consume.
4. Stop

Program:

```
buffer_size = int(input("Enter buffer size: "))

current = 0



while True:

    print("\n1. Produce  2. Consume  3. Exit")

    ch = int(input("Enter choice: "))
```

```
    if ch == 1:
        n = int(input("Produce how many? "))
        if current + n <= buffer_size:
            current += n
            print(f"Produced {n}. Buffer = {current}/{buffer_size}")
        else:
            print("Buffer full or not enough space!")
    elif ch == 2:
        n = int(input("Consume how many? "))
        if current >= n:
            current -= n
            print(f"Consumed {n}. Buffer = {current}/{buffer_size}")
        else:
            print("Not enough items to consume!")
    elif ch == 3:
        print("Exiting...")
        break
    else:
        print("Invalid choice!")
```

```
buffer_size = int(input("Enter buffer size: "))
current = 0
while True:
    print("\n1. Produce  2. Consume  3. Exit")
    ch = int(input("Enter choice: "))
    if ch == 1:
        n = int(input("Produce how many? "))
        if current + n <= buffer_size:
            current += n
            print(f"Produced {n}. Buffer = {current}/{buffer_size}")
        else:
            print("Buffer full or not enough space!")
    elif ch == 2:
        n = int(input("Consume how many? "))
        if current >= n:
            current -= n
            print(f"Consumed {n}. Buffer = {current}/{buffer_size}")
        else:
            print("Not enough items to consume!")
    elif ch == 3:
        print("Exiting...")
        break
    else:
        print("Invalid choice!")
```

Output:

```
Enter buffer size: 4

1. Produce  2. Consume  3. Exit
Enter choice: 2
Consume how many? 5
Not enough items to consume!

1. Produce  2. Consume  3. Exit
Enter choice: 3
Exiting...
```

Result:

Thus, we implemented the Producer–Consumer problem using C and executed it successfully.

Experiment 9: PROCESS SYNCHRONIZATION

Write a C program To implement Dining Philosophers problem using Threads and Semaphores.

Use Case Scenario:

Five **philosophers** sit around a table with one **chopstick between each pair**.
Each philosopher needs **two chopsticks to eat** and must wait if one is not available.
Semaphores are used to ensure that philosophers **eat without conflicts or deadlock**, maintaining synchronization among them.

Aim:

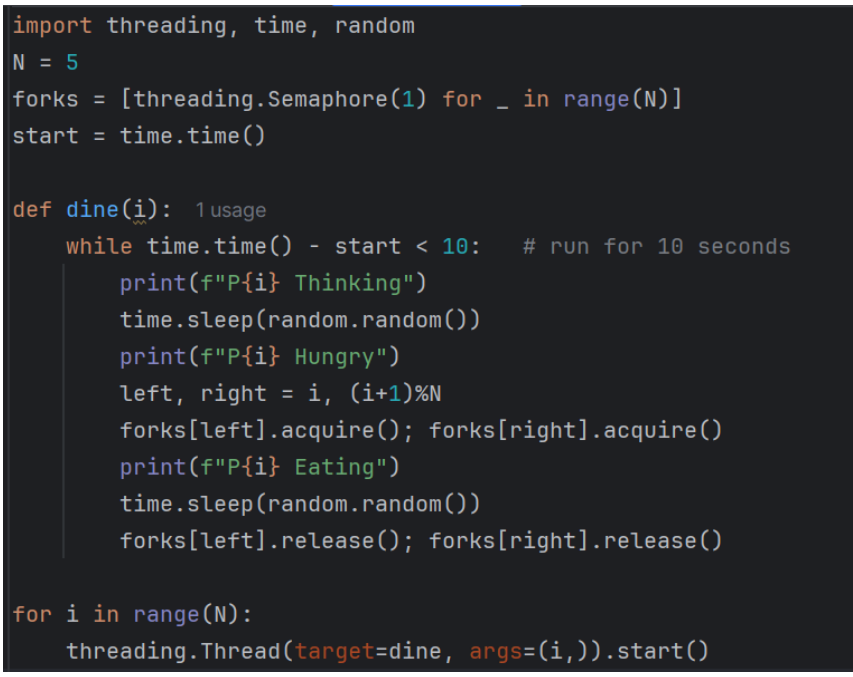
To implement Dining Philosophers problem using Threads and Semaphores.

Algorithm:

1. osophers.
2. Declare one thread for each philosopher.
3. Declare oneDefine number of phil semaphore per philosopher.
4. When a philosopher is hungry:
    - Check if chopsticks on both sides are free.
    - Acquire both chopsticks and eat.
    - Release chopsticks after eating.
    - If chopsticks aren't free, wait until they are available.

Program:

```
import threading, time, random
N = 5
forks = [threading.Semaphore(1) for _ in
range(N)]
start = time.time()

def dine(i):
    while time.time() - start < 10:   # run
for 10 seconds
        print(f"P{i} Thinking")
        time.sleep(random.random())
        print(f"P{i} Hungry")
        left, right = i, (i+1)%N
        forks[left].acquire();
forks[right].acquire()
        print(f"P{i} Eating")
        time.sleep(random.random())
        forks[left].release();
forks[right].release()

for i in range(N):
    threading.Thread(target=dine, args=(i,)).start()
```



Output:

```
P0 Hungry
P0 Eating
P3 HungryP4 Hungry

P3 Eating
P0 Thinking
P3 ThinkingP4 Eating

P3 Hungry
P2 HungryP1 Hungry

P0 Hungry
P4 ThinkingP3 Eating

P3 ThinkingP2 Eating

P4 Hungry
P3 Hungry
P1 EatingP2 Thinking

P1 ThinkingP0 Eating

P2 Hungry
P1 Hungry
P0 ThinkingP4 Eating

P0 Hungry
P4 ThinkingP3 Eating

P4 Hungry
P3 ThinkingP2 Eating

P2 ThinkingP1 Eating

P3 Hungry
P1 ThinkingP0 Eating

P2 Hungry
P1 Hungry
P4 EatingP0 Thinking

P0 Hungry
P4 ThinkingP3 Eating

P4 Hungry
P3 ThinkingP2 Eating

P2 ThinkingP1 Eating

P3 Hungry
P2 Hungry
P1 ThinkingP0 Eating

P0 ThinkingP4 Eating

P0 Hungry
P1 Hungry
P4 ThinkingP3 Eating
```

```
P4 Hungry
P3 ThinkingP2 Eating

P2 ThinkingP1 Eating

P3 Hungry
P2 Hungry
P1 ThinkingP0 Eating

P0 ThinkingP4 Eating

P0 Hungry
P1 Hungry
P4 ThinkingP3 Eating

P2 Eating
P4 Hungry
P1 Eating
P0 Eating
P4 Eating
```

Result:

Thus, implementation of Dining Philosophers Problem using semaphores was executed successfully.