K.R. MANGALAM UNIVERSITY

SCHOOL OF ENGINEERING & TECHNOLOGY

2025-26

LAB PRACTICAL FILE

OPERATING SYSTEM LAB

COURSE CODE: ENCS351

NAME: Keshav Sharma

ROLL NO.: 2301730221

SECTION: D

COURSE: BTECH CSE (AI & ML)

SUBMITTED TO: MS. SUMAN

INDEX

10	Process Creation and Management Using Python OS Module
11	File System Operations using Python

EXPERIMENT – 1

Simulate different types of operating systems using Python or Linux commands.

(1) Batch Processing OS

CODE:

```
# Batch Processing Simulation import
time

processes = ["Job1", "Job2", "Job3"]

print("--- Batch Processing OS Simulation ---")
for job in processes: print(f"Processing
{job}...") time.sleep(0.5)

print("All batch jobs completed.")
```

OUTPUT-

Output Clear

```
--- Batch Processing OS Simulation ---
Processing Job1...
Processing Job2...
Processing Job3...
All batch jobs completed.

=== Code Execution Successful ===
```

(2)

Multiprogramming OS

CODE:

```
# Multiprogramming Simulation
processes = ["Program1", "Program2", "Program3"]


print("--- Multiprogramming OS Simulation ---")
print("Multiple programs are loaded in memory...")


for p in processes:
    print(f"{p} is ready/blocked/running simultaneously.")


print("Multiprogramming execution simulated.")
```

OUTPUT-

(3)

Output   Clear

```
--- Multiprogramming OS Simulation ---
Multiple programs are loaded in memory...
Program1 is ready/blocked/running simultaneously.
Program2 is ready/blocked/running simultaneously.
Program3 is ready/blocked/running simultaneously.
Multiprogramming execution simulated.

=== Code Execution Successful ===
```

~~— Multitasking OS —~~

CODE:

Multitasking Simulation import

time

tasks = ["Task1", "Task2", "Task3"]

print("--- Multitasking OS Simulation ---")

print("CPU rapidly switches between tasks...")

Output   Clear

```
--- Multitasking OS Simulation ---
CPU rapidly switches between tasks...
CPU running small time slice of Task1
CPU running small time slice of Task2
CPU running small time slice of Task3
CPU running small time slice of Task1
CPU running small time slice of Task2
CPU running small time slice of Task3
Multitasking simulation completed.

=== Code Execution Successful ===
```

(4)

```
for _ in range(2): # simulate 2 cycles for t in tasks:
    print(f"CPU running small time slice of {t}")
    time.sleep(0.3)

print("Multitasking simulation completed.")
```

OUTPUT-

~~— Real-Time OS~~

CODE:

```
# Real-Time OS Simulation tasks
= ["T1", "T2", "T3"]

print("--- Real-Time OS Simulation ---") for
t in tasks:
    print(f"Task {t} must complete within strict deadline!")

print("Real-time tasks executed.")
```

OUTPUT-

(5)

Output Clear

```
--- Real-Time OS Simulation ---
Task T1 must complete within strict deadline!
Task T2 must complete within strict deadline!
Task T3 must complete within strict deadline!
Real-time tasks executed.

=== Code Execution Successful ===
```

EXPERIMENT – 2

Simulate CPU scheduling algorithms (FCFS, SJF, Round Robin, Priority).
Generate Gantt chart, compute average waiting and turnaround time.

(1) FCFS (First-Come, First-Served)

CODE:

```
# FCFS Scheduling Simulation
n = 3 # number of processes
processes = []

print("--- FCFS Scheduling ---") for
i in range(n):
   name = "P" + str(i+1) at =
   int(input(f"Arrival time of {name}: ")) bt =
   int(input(f"Burst time of {name}: "))
   processes.append([name, at, bt])

# Sort by Arrival Time processes.sort(key=lambda
x: x[1])

time = 0
wt = []
tat = []
gantt =
[]

for p in processes: name,
   at, bt = p
```

```
    if time < at:
        time        =       at
    wt.append(time      -
    at)     time     +=     bt
    tat.append(time     -
    at)
    gantt.append(name)

#  Output  print("\nGANTT  CHART:",  "  |
".join(gantt)) print("\nProcess WT TAT") for
i, p in enumerate(processes):
    print(f"{p[0]}        {wt[i]}  {tat[i]}")
  print(f"\nAverage WT={sum(wt)/n:.2f} Average TAT={sum(tat)/n:.2f}")
```

OUTPUT-

```
Output                                                        Clear
--- FCFS Scheduling ---
Arrival time of P1: 0
Burst time of P1: 4
Arrival time of P2: 1
Burst time of P2: 3
Arrival time of P3: 2
Burst time of P3: 2

GANTT CHART: P1 | P2 | P3

Process  WT  TAT
P1       0   4
P2       3   6
P3       5   7

Average WT=2.67  Average TAT=5.67

=== Code Execution Successful ===
```

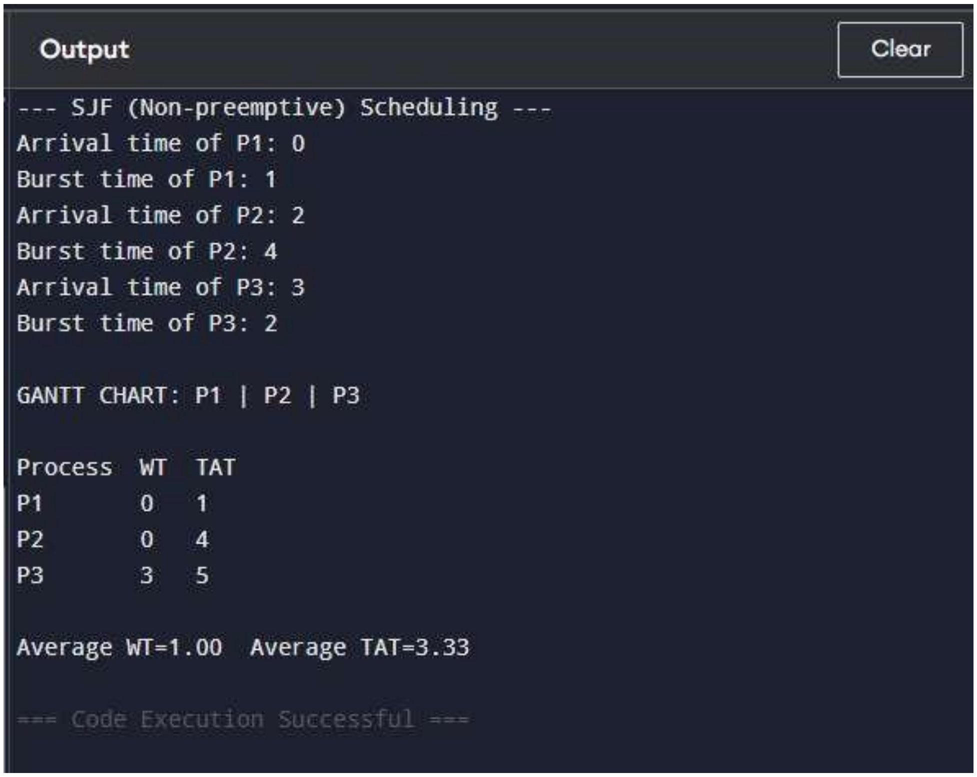
(2) SJF (Non-Preemptive)

CODE:

```
# SJF Scheduling Simulation (Non-preemptive) n
= 3 processes = [] print("--- SJF (Non-
preemptive) Scheduling ---") for i in range(n):
    name = "P" + str(i+1) at =
    int(input(f"Arrival time of {name}: ")) bt =
    int(input(f"Burst time of {name}: "))
    processes.append([name, at, bt])


time = 0 completed =
0 visited = [False]*n
wt = [0]*n tat =
[0]*n gantt = []
while completed < n:
```

```
idx = -1 min_bt =
9999
    for i in range(n):
        if processes[i][1] <= time and not visited[i] and processes[i][2] < min_bt:
            min_bt = processes[i][2]
     idx = i if idx == -1: time += 1
      continue visited[idx] = True
gantt.append(processes[idx][0]) wt[idx] =
time - processes[idx][1] time +=
processes[idx][2] tat[idx] = time -
processes[idx][1] completed += 1 # Output
print("\nGANTT CHART:", " | ".join(gantt))
print("\nProcess WT TAT") for i, p in
enumerate(processes):
    print(f"{p[0]}       {wt[i]} {tat[i]}")
  print(f"\nAverage WT={sum(wt)/n:.2f} Average TAT={sum(tat)/n:.2f}")
```

OUTPUT-



<u>(3) Priority Scheduling (Non-Preemptive)</u>

CODE:

```
# Priority Scheduling (Non-preemptive) n
= 3
processes = []


print("--- Priority Scheduling ---") for
i in range(n):
    name = "P" + str(i+1) at = int(input(f"Arrival time of
    {name}: ")) bt = int(input(f"Burst time of {name}: "))
    pr = int(input(f"Priority of {name} (lower = higher): "))
    processes.append([name, at, bt, pr])


# Sort by Priority then Arrival processes.sort(key=lambda
x: (x[3], x[1]))
```

```
time = 0
wt = []
tat = []
gantt =
[]

for p in processes:
    name, at, bt, pr = p
    if time < at: time =
    at wt.append(time -
    at) time += bt
    tat.append(time - at)
    gantt.append(name)

# Output print("\nGANTT CHART:", " |
".join(gantt)) print("\nProcess WT TAT") for
i, p in enumerate(processes):
    print(f"{p[0]}      {wt[i]} {tat[i]}")
  print(f"\nAverage WT={sum(wt)/n:.2f} Average TAT={sum(tat)/n:.2f}")
```

OUTPUT-

```
Output                                                    Clear
--- Priority Scheduling ---
Arrival time of P1: 4
Burst time of P1: 1
Priority of P1 (lower = higher): 2
Arrival time of P2: 4
Burst time of P2: 3
Priority of P2 (lower = higher): 2
Arrival time of P3: 2
Burst time of P3: 3
Priority of P3 (lower = higher): 1

GANTT CHART: P3 | P1 | P2

Process  WT  TAT
P3       0   3
P1       1   2
P2       2   5

Average WT=1.00  Average TAT=3.33

=== Code Execution Successful ===
```

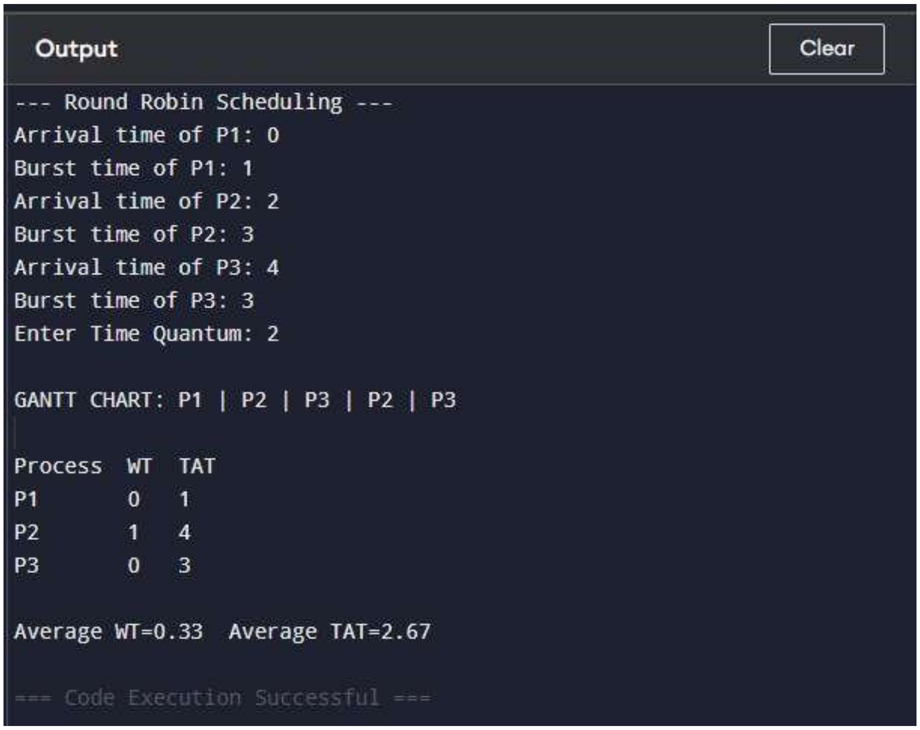
<u>(4) Round Robin</u>

CODE:

```
# Round Robin Scheduling n
= 3
processes = []


print("--- Round Robin Scheduling ---") for
i in range(n):
    name = "P" + str(i+1) at =
    int(input(f"Arrival time of {name}: ")) bt =
    int(input(f"Burst time of {name}: "))
    processes.append([name, at, bt])
```

```
quantum = int(input("Enter Time Quantum: "))
remaining = [p[2] for p in processes] wt = [0]*n
tat = [0]*n time = 0 queue = list(range(n)) gantt
= [] while queue: i = queue.pop(0) name, at, bt
= processes[i] if remaining[i] > quantum:
gantt.append(name) time += quantum
remaining[i] -= quantum
        queue.append(i)
    else:
        gantt.append(name)
        time +=
        remaining[i] wt[i] =
        time - at - bt tat[i] =
        time - at
        remaining[i] = 0
# Output print("\nGANTT CHART:", " |
".join(gantt)) print("\nProcess WT TAT") for
i, p in enumerate(processes):
    print(f"{p[0]}      {wt[i]}  {tat[i]}")
  print(f"\nAverage WT={sum(wt)/n:.2f} Average TAT={sum(tat)/n:.2f}")
```

OUTPUT-



<u>EXPERIMENT – 3</u>

Simulate Worst-fit, Best-fit, and First-fit memory allocation.

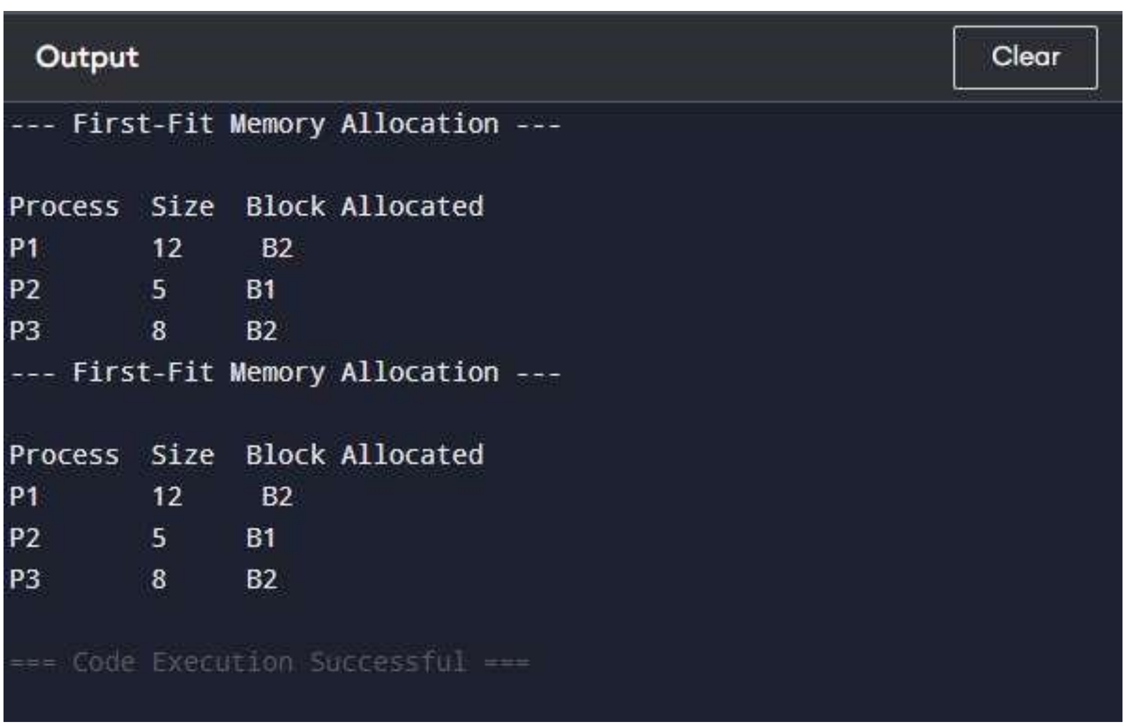
<u>(1) First-Fit Memory Allocation</u>

CODE:

```
# First-Fit Memory Allocation
print("--- First-Fit Memory Allocation ---")

# Input memory blocks
blocks = [10, 20, 15] #
Input process sizes
processes = [12, 5, 8]

allocation = [-1] * len(processes)
```

```
for i, p_size in enumerate(processes):
    for j, b_size in enumerate(blocks):
        if b_size >= p_size:
            allocation[i] = j
            blocks[j] -=
            p_size break

# Output print("\nProcess Size Block
Allocated") for i, p in
enumerate(processes):
    if allocation[i] != -1:
        print(f"P{i+1}    {p}    B{allocation[i]+1}")
    else:
        print(f"P{i+1}    {p}    Not Allocated")
```

OUTPUT-



```
Output                                   Clear
--- First-Fit Memory Allocation ---

Process  Size  Block Allocated
P1       12     B2
P2       5     B1
P3       8     B2
--- First-Fit Memory Allocation ---

Process  Size  Block Allocated
P1       12     B2
P2       5     B1
P3       8     B2

=== Code Execution Successful ===
```

(2) Best-Fit Memory Allocation

CODE:

```
# Best-Fit Memory Allocation
print("--- Best-Fit Memory Allocation ---")


blocks = [10, 20, 15]
processes = [12, 5, 8]


allocation = [-1] * len(processes)
for i, p_size in enumerate(processes):
    best_idx = -1 for j, b_size in
    enumerate(blocks):
        if b_size >= p_size:
            if best_idx == -1 or b_size < blocks[best_idx]:
                best_idx = j if
    best_idx != -1:
        allocation[i] = best_idx blocks[best_idx]
        -= p_size


# Output print("\nProcess Size Block
Allocated") for i, p in
enumerate(processes):
    if allocation[i] != -1:
        print(f"P{i+1}        {p}     B{allocation[i]+1}")
    else:
        print(f"P{i+1}        {p}     Not Allocated")
```

OUTPUT-

```
Output                                                    Clear
--- Best-Fit Memory Allocation ---

Process  Size  Block Allocated
P1       12     B3
P2       5     B1
P3       8     B2

=== Code Execution Successful ===
```

<u>(3) Worst-Fit Memory Allocation</u>

CODE:

```
print("--- Worst-Fit Memory Allocation ---")


blocks = [10, 20, 15] processes
= [12, 5, 8]
allocation = [-1] * len(processes)


for i, p_size in enumerate(processes):
    worst_idx = -1 for j, b_size in
    enumerate(blocks):
        if b_size >= p_size:
            if worst_idx == -1 or b_size > blocks[worst_idx]:
                worst_idx = j if
    worst_idx != -1:
        allocation[i] = worst_idx blocks[worst_idx]
        -= p_size
```

```
# Output print("\nProcess Size Block
Allocated") for i, p in
enumerate(processes):
    if allocation[i] != -1:
        print(f"P{i+1}        {p}     B{allocation[i]+1}")
    else:
        print(f"P{i+1}        {p}     Not Allocated")
```

OUTPUT-

```
Output                                        Clear
--- Worst-Fit Memory Allocation ---

Process  Size  Block Allocated
P1       12     B2
P2       5     B3
P3       8     B1

=== Code Execution Successful ===
```

EXPERIMENT – 4

Implement Banker's Algorithm for deadlock avoidance/prevention.

(1) Deadlock Avoidance (Banker's Algorithm)

CODE:

```
# Banker's Algorithm — Deadlock Avoidance
print("--- Banker's Algorithm for Deadlock Avoidance ---")


# Number of processes and resources n
= 3 # processes
m = 3 # resources


# Maximum resources needed by each process max_resources
= [
    [7, 5, 3],
    [3, 2, 2],
    [9, 0, 2]
]


# Currently allocated resources allocated
= [
    [0, 1, 0],
    [2, 0, 0],
    [3, 0, 2]
]


# Available resources
```

```
available = [3, 3, 2]


# Calculate Need matrix
need = [[max_resources[i][j] - allocated[i][j] for j in range(m)] for i in range(n)]


print("\nNeed Matrix:") for
i in range(n):
    print(f"P{i+1}: {need[i]}")


# Safety Algorithm
finish  =  [False]*n
safe_seq = []


while len(safe_seq) < n:
    allocated_in_round = False for
    i in range(n):
        if not finish[i] and all(need[i][j] <= available[j] for j in range(m)):
            for j in range(m):
                available[j]                    +=
    allocated[i][j]    finish[i]    =    True
    safe_seq.append(f"P{i+1}")
    allocated_in_round   =   True   if   not
    allocated_in_round: break


#        Output        if
len(safe_seq)   ==   n:
print("\nSystem  is  in
```

SAFE state

(Avoidance).")

print("Safe

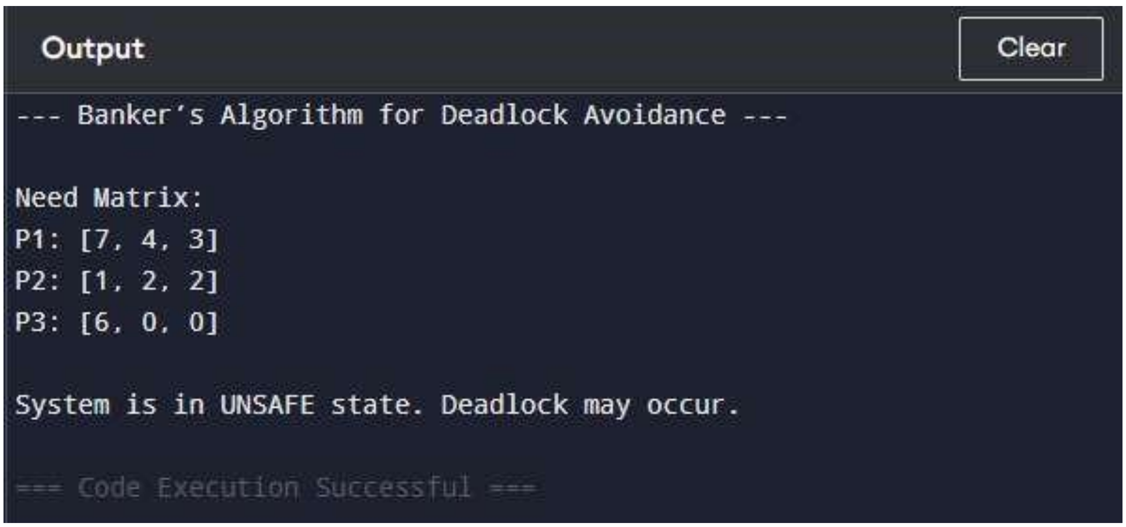
sequence:", " ->

".join(safe_seq))

else:

    print("\nSystem is in UNSAFE state. Deadlock may occur.")

OUTPUT-



<u>(2) Deadlock Prevention (Banker’s Algorithm / Resource Allocation Limits)</u>

CODE:

Banker’s Algorithm — Deadlock Prevention print("---

Banker’s Algorithm for Deadlock Prevention ---")

Total resources total

= [10, 5, 7]

Maximum resources needed by each process max_resources

= [

```
    [7, 5, 3],
    [3, 2, 2],
    [9, 0, 2]
]
# Currently allocated resources allocated
= [
    [0, 1, 0],
    [2, 0, 0],
    [3, 0, 2]
]
# Available resources
available = [total[i] - sum(allocated[j][i] for j in range(len(allocated))) for i in
range(3)] print("Available resources:", available)
# Simple prevention: do not allow allocation if request > (total - 1) for
i, req in enumerate(max_resources):
    if any(req[j] > total[j] - 1 for j in range(len(req))):
        print(f"Request by P{i+1} denied (prevention).") else:
        print(f"Request by P{i+1} can be allocated safely.")
```

OUTPUT-

```
Output                                                Clear
--- Banker's Algorithm for Deadlock Prevention ---
Available resources: [5, 4, 5]
Request by P1 denied (prevention).
Request by P2 can be allocated safely.
Request by P3 can be allocated safely.

=== Code Execution Successful ===
```

EXPERIMENT – 5

Implement Producer-Consumer problem using semaphores.

CODE:

```
# Producer-Consumer Problem using Semaphores
import threading import time import random

# Buffer and semaphores buffer
= []
buffer_size = 2

empty = threading.Semaphore(buffer_size) # empty slots full
= threading.Semaphore(0) # full slots
mutex = threading.Semaphore(1)          # mutual exclusion

  # Producer function def
 producer(items):
    for    item    in    items:    empty.acquire()
      mutex.acquire()           buffer.append(item)
      print(f"Produced: {item} | Buffer: {buffer}")
      mutex.release()                      full.release()
      time.sleep(random.uniform(0.1, 0.5))
# Consumer function def consumer(n): for _ in
range(n): full.acquire() mutex.acquire() item =
buffer.pop(0) print(f"Consumed: {item} | Buffer:
{buffer}")    mutex.release()    empty.release()
time.sleep(random.uniform(0.1, 0.5))
```

```
# Sample items to produce
items_to_produce = [1, 2, 3]
num_items = len(items_to_produce)

# Threads t1 = threading.Thread(target=producer,
args=(items_to_produce,)) t2 =
threading.Thread(target=consumer, args=(num_items,))

# Start threads
t1.start()
t2.start()

# Wait for completion
t1.join() t2.join()
print("Producer-
Consumer
Simulation
Completed.")
```

OUTPUT-

Output Clear

```
Produced: 1 | Buffer: [1]
Consumed: 1 | Buffer: []
Produced: 2 | Buffer: [2]
Consumed: 2 | Buffer: []
Produced: 3 | Buffer: [3]
Consumed: 3 | Buffer: []
Producer-Consumer Simulation Completed.

=== Code Execution Successful ===
```

EXPERIMENT – 6

Process synchronization using semaphores.

CODE:

```
# Process Synchronization using Semaphores import
threading
import time

# Shared resource shared_counter
= 0
# Semaphore for mutual exclusion mutex =
threading.Semaphore(1) # Process 1 function def process1():
global shared_counter for i in range(3): mutex.acquire()
shared_counter += 1 print(f"Process 1 incremented counter to
{shared_counter}") mutex.release()
    time.sleep(0.2)

# Process 2 function def
process2(): global
shared_counter for i in
range(3):
mutex.acquire()
shared_counter += 1
print(f"Process 2
incremented counter to
{shared_counter}")
mutex.release()
```

time.sleep(0.3)

Create threads t1 =
threading.Thread(target=process1) t2 =
threading.Thread(target=process2)

# Start threads
t1.start()
t2.start()

# Wait for completion
t1.join() t2.join()

print("Final value of shared counter:", shared_counter) print("Process Synchronization Simulation Completed.")

OUTPUT-

```
Output                                              Clear
Process 1 incremented counter to 1
Process 2 incremented counter to 2
Process 1 incremented counter to 3
Process 2 incremented counter to 4
Process 1 incremented counter to 5
Process 2 incremented counter to 6
Final value of shared counter: 6
Process Synchronization Simulation Completed.

=== Code Execution Successful ===
```

EXPERIMENT – 7

Use UNIX/Linux I/O system calls (open, read, write, close, seek, stat).

CODE:

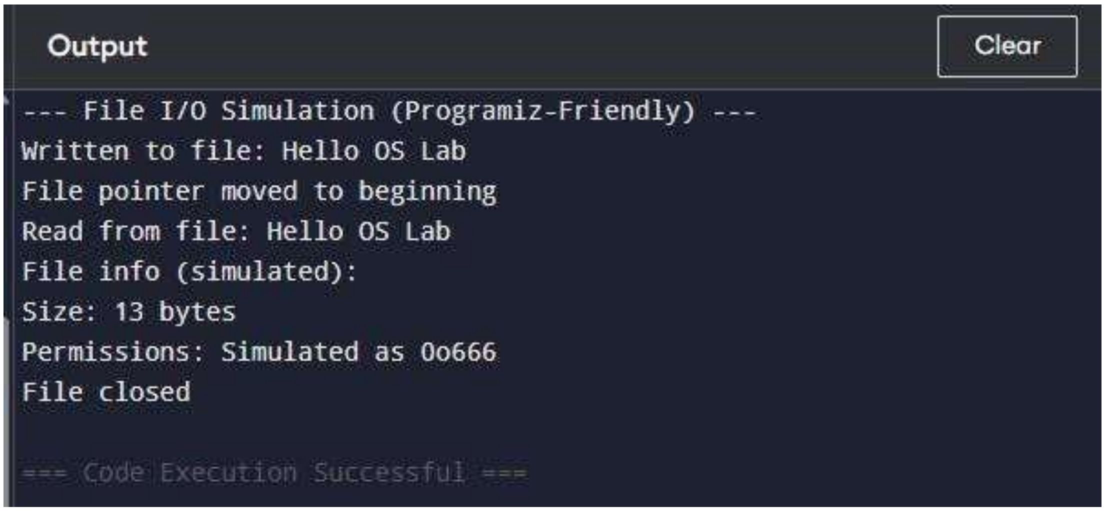
```
import io

print("--- File I/O Simulation (Programiz-Friendly) ---")

# Create an in-memory file f
= io.StringIO()

# Write to "file" text =
"Hello OS Lab\n"
f.write(text)
print(f"Written to file: {text.strip()}")

# Move file pointer to beginning
f.seek(0)
print("File pointer moved to beginning")

# Read from "file" content
= f.read()
print("Read from file:", content.strip())

# Simulate file info size =
len(content.encode())
print("File info (simulated):")
print(f"Size: {size} bytes")
print(f"Permissions:
Simulated as 0o666")
```

Close "file"

f.close() print("File

closed")

OUTPUT-



EXPERIMENT – 8

System Startup, Process Creation, and Termination Simulation in Python.

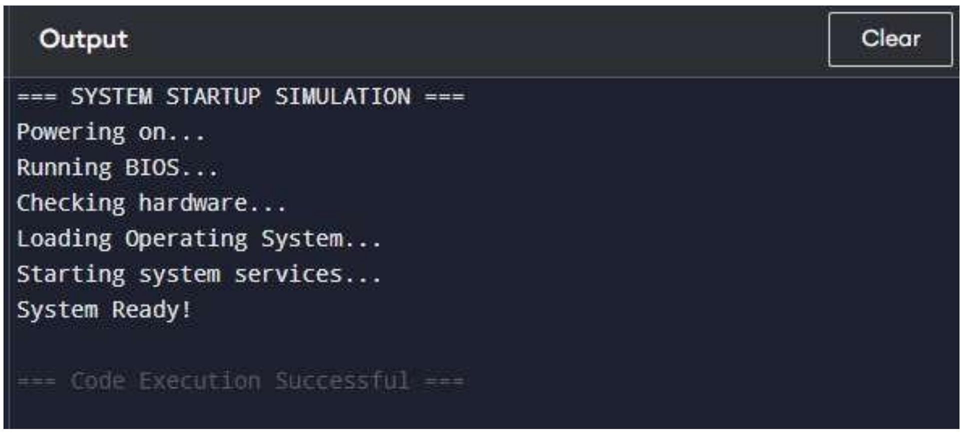
(1) System Startup Stimulation

CODE:

```
print("=== SYSTEM STARTUP SIMULATION ===")

boot_steps = [
    "Powering on...",
    "Running BIOS...",
    "Checking hardware...",
    "Loading Operating System...",
```

```
    "Starting system services...",
    "System Ready!"
]
for step in boot_steps: print(step)
```

OUTPUT-



<u>(2) Process Creation Stimulation</u>

CODE:

```
print("=== PROCESS CREATION SIMULATION ===")
num = int(input("Enter number of processes to create: "))

process_table = []

for i in range(num): name = input(f"Enter name for Process
{i+1}: ") pid = 2000 + i process_table.append((pid, name))
print(f"Process Created → PID: {pid}, Name: {name}")
print("\nProcess Table:") for pid, name in process_table:
print(f"PID: {pid} | Name: {name}")
```

OUTPUT-

```
Output                                                    Clear
=== PROCESS CREATION SIMULATION ===
Enter number of processes to create: 3
Enter name for Process 1: Chrome
Process Created → PID: 2000, Name: Chrome
Enter name for Process 2: VsCode
Process Created → PID: 2001, Name: VsCode
Enter name for Process 3: MusicPlayer
Process Created → PID: 2002, Name: MusicPlayer

Process Table:
PID: 2000 | Name: Chrome
PID: 2001 | Name: VsCode
PID: 2002 | Name: MusicPlayer

=== Code Execution Successful ===
```

<u>(3) Process Termination Simulation</u>

CODE:

print("=== PROCESS TERMINATION SIMULATION ===")

Preloaded process table for demonstration process_table

= [

(3000, "Chrome"),

(3001, "Notepad"),

(3002, "Camera")

]

print("Current Processes:") for pid,

name in process_table: print(f"PID:

{pid} | Name: {name}")

pid_to_terminate = int(input("\nEnter PID to terminate: "))

```
found = False for p in
process_table:
    if     p[0]     ==     pid_to_terminate:     found     =     True
        process_table.remove(p)  print(f"Process  Terminated  →
        PID: {pid_to_terminate}") break

if not found:
    print("Invalid PID! No such process.")

print("\nUpdated Process Table:")
if    len(process_table)    ==    0:
print("No active processes.") else:
    for pid, name in process_table:
        print(f"PID: {pid} | Name: {name}")
```

OUTPUT-

```
Output                                                    Clear
=== PROCESS TERMINATION SIMULATION ===
Current Processes:
PID: 3000 | Name: Chrome
PID: 3001 | Name: Notepad
PID: 3002 | Name: Camera

Enter PID to terminate: 3001
Process Terminated → PID: 3001

Updated Process Table:
PID: 3000 | Name: Chrome
PID: 3002 | Name: Camera

=== Code Execution Successful ===
```

EXPERIMENT – 9

Create and manipulate threads in Python.

(1) Create "Threads"

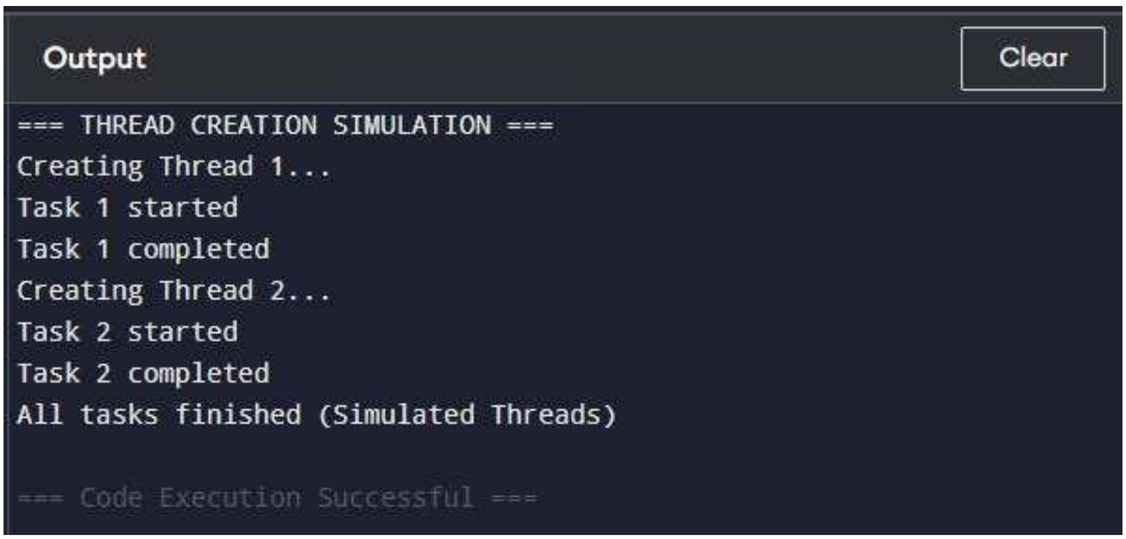
CODE:

```
import time


print("=== THREAD CREATION SIMULATION ===")


def task1(): print("Task 1
  started") time.sleep(1)
  print("Task 1 completed")


def task2(): print("Task 2
  started") time.sleep(1)
  print("Task 2 completed")
```

print("Creating Thread 1...") task1()

print("Creating Thread 2...") task2()

print("All tasks finished (Simulated Threads)")
OUTPUT-



(2) Parallel Execution

CODE:

import time

print("=== THREAD INTERLEAVING SIMULATION ===")

def threadA():

    for i in range(3):

        print(f"Thread A → Step {i+1}") time.sleep(0.5)

def threadB():

```
    for i in range(3):
        print(f"Thread B → Step {i+1}") time.sleep(0.5)

print("Starting simulated parallel execution...\n")
for i in range(3):
    threadA()
    threadB()

print("\nSimulation Complete")
```

OUTPUT-

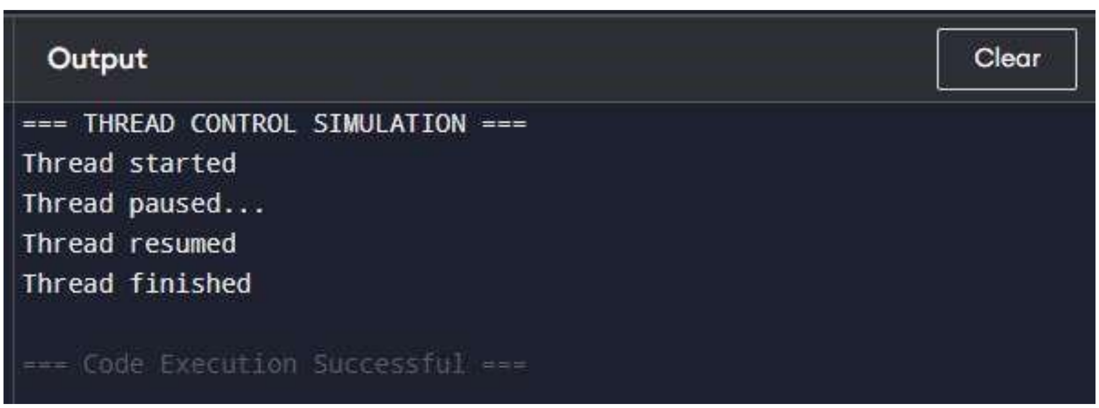
```
Output                                                Clear
=== THREAD INTERLEAVING SIMULATION ===
Starting simulated parallel execution...

Thread A → Step 1
Thread A → Step 2
Thread A → Step 3
Thread B → Step 1
Thread B → Step 2
Thread B → Step 3
Thread A → Step 1
Thread A → Step 2
Thread A → Step 3
Thread B → Step 1
Thread B → Step 2
Thread B → Step 3
Thread A → Step 1
Thread A → Step 2
Thread A → Step 3
Thread B → Step 1
Thread B → Step 2
Thread B → Step 3

Simulation Complete
```

(3) Start, Pause, Resume "Threads"

CODE:

```
import time


print("=== THREAD CONTROL SIMULATION ===")


def worker(): print("Thread
    started")     time.sleep(1)
    print("Thread paused...")
    time.sleep(1)
    print("Thread  resumed")
    time.sleep(1)
    print("Thread finished")


worker()
```

OUTPUT-



EXPERIMENT – 10

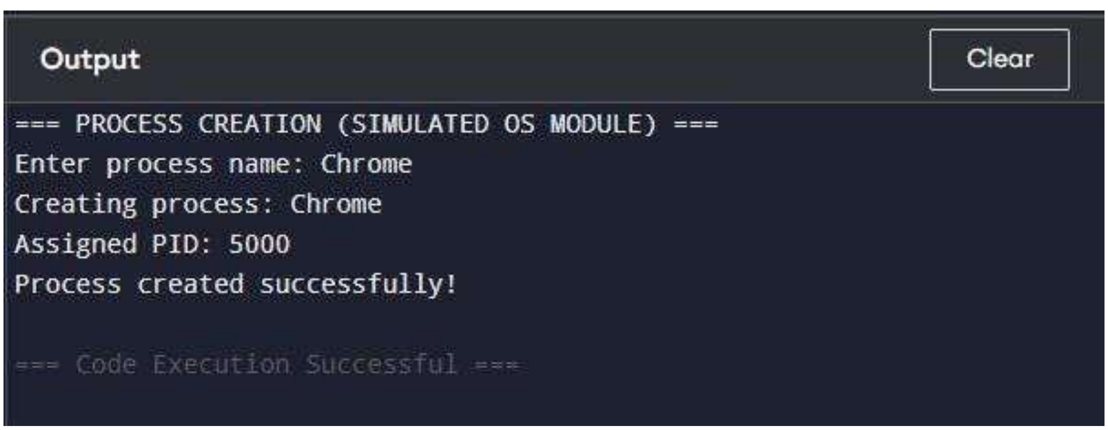
Process Creation and Management Using Python OS Module.

(1) Simulate Process Creation

CODE:

```
print("=== PROCESS CREATION (SIMULATED OS MODULE) ===")


def    create_process(name,    pid):
    print(f"Creating process: {name}")
    print(f"Assigned PID: {pid}")


process_name = input("Enter process name: ") pid
= 5000


create_process(process_name, pid) print("Process
created successfully!")
```

OUTPUT-



(2) Simulate Process Table

CODE:

```
print("=== PROCESS TABLE SIMULATION ===")


processes = [
```

```
    (1001, "System"),
    (1002, "Chrome"),
    (1003, "MusicApp")
]

print("PID\tProcess Name")
print(" -----------------------")
for pid, name in processes:
    print(f"{pid}\t{name}")
```

OUTPUT-

(3) Process Termination

CODE:

```
print("=== PROCESS TERMINATION SIMULATION ===")

process_table = [
    (6000, "Chrome"),
    (6001, "Notes"),
    (6002, "WhatsApp")
]
```

```
print("Current Processes:") for pid,
name in process_table: print(f"PID:
{pid} | Name: {name}")

pid_to_kill = int(input("\nEnter PID to terminate: "))


found = False for p in
process_table:
   if p[0] == pid_to_kill: process_table.remove(p)
      print(f"Process {pid_to_kill} terminated (simulated)")
      found = True
      break


if not found:
   print("No such process!")
print("\nUpdated Process Table:") for
pid, name in process_table:
   print(f"PID: {pid} | Name: {name}")
```

OUTPUT-

Output Clear

```
=== PROCESS TERMINATION SIMULATION ===
Current Processes:
PID: 6000 | Name: Chrome
PID: 6001 | Name: Notes
PID: 6002 | Name: WhatsApp

Enter PID to terminate: 6002
Process 6002 terminated (simulated)

Updated Process Table:
PID: 6000 | Name: Chrome
PID: 6001 | Name: Notes

=== Code Execution Successful ===
```

EXPERIMENT – 11

File System Operations using Python

(1) Create & Write to File

CODE:

```
from io import StringIO


print("=== FILE CREATION & WRITE OPERATION (SIMULATED) ===")


file = StringIO() # virtual in-memory file


file.write("Hello OS Lab\n")
file.write("This is a write operation in a simulated file.")


print("File created and written successfully (SIMULATED FILE).")
```

OUTPUT-

```
Output                                                    Clear
=== FILE CREATION & WRITE OPERATION (SIMULATED) ===
File created and written successfully (SIMULATED FILE).

=== Code Execution Successful ===
```

<u>(2) Read File Content</u>

CODE:

```
from io import StringIO


print("=== FILE READ OPERATION (SIMULATED) ===")


# Create virtual file again (since Programiz resets each run)
file = StringIO("Hello OS Lab\nThis is a write operation in a simulated file.")


file.seek(0)
content = file.read()


print("File Content:") print(content)
```

OUTPUT-

```
Output                                                    Clear
=== FILE READ OPERATION (SIMULATED) ===
File Content:
Hello OS Lab
This is a write operation in a simulated file.

=== Code Execution Successful ===
```

(3) File “Stat”

CODE:

from io import StringIO import

time

print("=== FILE INFORMATION (STAT SIMULATION) ===")

data = "Hello OS Lab\nThis is a write operation in a simulated file." virtual_file

= StringIO(data)

size = len(data.encode())

created = time.ctime()

modified = time.ctime()

print(f"Filename: simulated_file.txt")

print(f"Size: {size} bytes")

print(f"Created: {created}")

print(f"Modified: {modified}")

OUTPUT-

```
Output                                                Clear
=== FILE INFORMATION (STAT SIMULATION) ===
Filename: simulated_file.txt
Size: 59 bytes
Created: Thu Nov 20 06:19:45 2025
Modified: Thu Nov 20 06:19:45 2025

=== Code Execution Successful ===
```