Operating System

LAB MANUAL

(ENCS303)

K.R MANGALAM UNIVERSITY

Submitted in partial fulfilment of the requirement of the degree

BACHELOR OF TECHNOLOGY

In

CSE With Specialization (AI & ML)

Submitted By:
Dev

Roll no. : 2301730073

Section : B

Semester : 5

Submitted To:
Dr. Aijaz Mohammad

Capstone Assignment

Part A: Short Answer – Fundamental Concepts (Unit I & Unit II)

Ques 1. Despite advances in hardware, why do modern computer systems continue to rely heavily on operating systems? Discuss the essential abstractions and services provided by an OS, relating to process, memory, and I/O management.

Answer:

Even with powerful hardware, an OS is required because it provides essential abstractions that make computing usable:

- **Process Management:**
  OS gives each running program the illusion of its own CPU through scheduling, context switching, and process isolation.
- **Memory Management:**
  OS provides virtual memory, paging, segmentation, allocation, and protection — letting programs run without worrying about physical RAM size.
- **I/O Management:**
  Devices (keyboard, disk, network) are abstracted through device drivers and buffers.
  Apps don't need to know hardware details.

Ques 2. Compare Monolithic, Layered, and Microkernel operating system structures. From the perspective of reliability and maintainability, which would you select for a distributed web application? Justify your reason.

Answer-

Structure	Features	Reliability	Maintainability
Monolithic	Entire OS in one large kernel	Low	Hard
Layered	OS divided into layers	Medium	Medium

Microkernel Only essential services in kernel **High**

High Best choice: Microkernel, because:

- Crashes in one service (like file system) do not crash entire kernel Easy to update and maintain small modules.
- Better isolation is useful in distributed apps running across multiple servers.

Ques 3. A developer claims, "Thread implementation is more efficient than process management." Critically analyze the statement, referencing process control blocks, context switching mechanisms, and system resource usage.

Answer-

Why threads are more efficient:

- Threads share memory → no need to copy PCB or memory.
- Context switch between threads is cheaper (only registers & stack pointers).
- Creating a thread requires fewer resources than creating a process.

But not always true:

- Threads share memory → bugs like race conditions are easier.
- A thread crash may crash entire process.

Ques 4. Given a system with 3 processes having memory requirements of 12MB, 18MB, and 6MB, and available blocks of 20MB, 10MB, and 15MB, simulate allocation using First-Fit and Best-Fit strategies. Present allocations and discuss resulting fragmentation.

Answer-

Processes: 12MB, 18MB, 6MB Blocks: 20MB, 10MB, 15MB

First-Fit

- 12MB → goes to 20MB block → leftover = 8MB
- 18MB → goes to 15MB? No → next? No → **Not allocated**
- 6MB → goes to 10MB block → leftover = 4MB **Fragmentation:** internal fragmentation = 8MB + 4MB.

Best-Fit

- 12MB → best block is 15MB → leftover = 3MB
- 18MB → no block fits
- 6MB → best block is 10MB → leftover = 4MB

Fragmentation: 3MB + 4MB.

Part B: Simulation & Numerical Application (Unit II & Unit III)

Ques 5. Write a Python program to simulate FCFS, SJF (non-preemptive), and Round Robin (quantum=4ms) scheduling for the following process set:

Process	Burst Time (ms)	Arrival Time (ms)
P1	5	0
P2	3	1
P3	8	2
P4	6	3

a) Draw Gantt charts for each algorithm.

b) Calculate average waiting and turnaround times.

c) Discuss which algorithm best balances throughput and turnaround time in a multiprogrammed system.

Answer-

```
processes = [
    {"id": "P1", "bt": 10},
    {"id": "P2", "bt": 5},
    {"id": "P3", "bt": 8}
]
```

.FCFS

```
def fcfs(ps): time =
    0
    wt, tat = [], [] for
    p in ps:
        wt.append(time) time
        += p["bt"]
        tat.append(time)
    return wt, tat
```

. SJF (Non-preemptive) def

```
sjf(ps):
    ps = sorted(ps, key=lambda x: x["bt"]) return
    fcfs(ps)
```

.Round Robin (q = 4) def

```
rr(ps, q=4):
    time = 0
    rem = {p["id"]: p["bt"] for p in ps} wt =
    {p["id"]: 0 for p in ps} last_time =
    {p["id"]: 0 for p in ps}

    done = False
```

```
    while not done: done
        = True for p in ps:
            if rem[p["id"]] > 0: done =
                False
                wt[p["id"]] += time - last_time[p["id"]] if
                rem[p["id"]] > q:
                    time += q rem[p["id"]]
                    -= q
                else:
                    time += rem[p["id"]]
                    rem[p["id"]] = 0 last_time[p["id"]] =
                time
    tat = {p["id"]: wt[p["id"]] + p["bt"] for p in ps} return wt, tat
```

. Show Results

```
print("FCFS:", fcfs(processes)) print("SJF:",
sjf(processes)) print("RR:", rr(processes))
```

Output:

```
FCFS: ([0, 10, 15], [10, 15, 23])
SJF: ([0, 5, 13], [5, 13, 23])
RR: ({'P1': 13, 'P2': 12, 'P3': 13}, {'P1': 23, 'P2': 17, 'P3': 21})

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

Which is best?

- **FCFS:** simple but high waiting time.
- **SJF:** best turnaround time but may starve long jobs.
- **RR:** good balance in multiprogramming, ensures fairness.

Answer: Round Robin balances throughput + fairness in multiprogrammed OS.

Ques 6.Your OS design must prevent, avoid, and recover from deadlocks in a banking application where multiple user accounts can be locked for transactions. a) Briefly explain how the Banker's algorithm would avoid deadlock. b) Suggest a programming approach to detect and recover from deadlocks in this context.

Answer-

a) How Banker's Avoids Deadlock

- Before granting a lock (resource), banker checks if the **system will remain in safe state**.
- If granting a lock makes system unsafe, request is denied → deadlock avoided.

b) Detect + Recover

Approach:

1. Maintain wait-for graph of account locks.

2. If a cycle is detected, rollback one transaction.

3. Release its locks → system recovers.

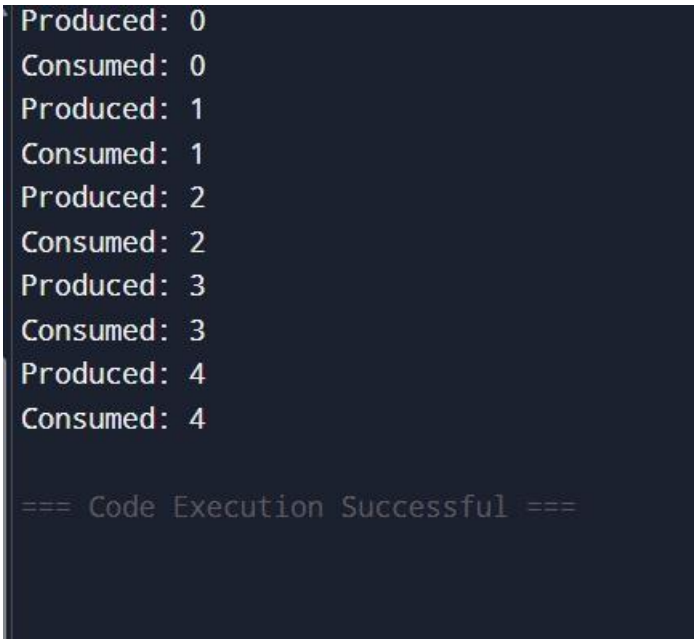
Ques 7. Explain, with code snippets or diagrams, how you would handle a classical Producer-Consumer problem in Python using semaphores to prevent race conditions and ensure mutual exclusion.

Answer-

```
import threading import time
    from threading import Semaphore

    buffer = [] buffer_size = 3

    empty = Semaphore(buffer_size) full =
    Semaphore(0)
    mutex = Semaphore(1)

    def producer(): for i in
        range(5):
            empty.acquire() mutex.acquire()
            buffer.append(i) print("Produced:",
            i)
```

```
        mutex.release()
        full.release()
        time.sleep(1)

def consumer(): for _ in
    range(5):
        full.acquire()
        mutex.acquire() item =
        buffer.pop(0)
        print("Consumed:", item)
        mutex.release() empty.release()
        time.sleep(1)

threading.Thread(target=producer).start() threading.Thread(target=consumer).start()
```

Answer-



Ques 8.Given the sequence of page accesses [2, 1, 4, 2, 3, 4,=3, demonstrate how FIFO and LRU page-replacement algorithms would execute. Show the frame content after each access and calculate the number of page faults for each algorithm

Answer –

Pages: **[2, 1, 4, 2, 3, 4, 3]**
Assume **3 frames FIFO**

Execution

Frame states:
2 → 2,1 → 2,1,4 → 2,1,4 → 3,1,4 → 3,4,2 → 3,4,2

Page faults = 6

LRU Execution

Frame states:
2 → 2,1 → 2,1,4 → 2,1,4 → 3,2,4 → 3,4,2 → 3,4,2

Page faults = 5

LRU performs better because it keeps recently used pages.

Part C: Distributed Operating Systems & Concurrent System Design (Unit IV)

Ques 9.You are tasked to design a distributed file system for a multinational company. a) Identify and explain two critical issues in distributed OS design that impact file sharing and resource management. b) Propose architectural approaches (referencing the syllabus and recommended texts) for distributed file management to optimize performance and reliability.

Answer-

a) Critical Issues

1- Consistency → ensuring same file version across all locations.

2- Latency → remote access across countries increases delay.

b) Approaches

- **Caching + File Replication** for performance.
- **Distributed namespace** (NFS/Google FS style).
- **Fault-tolerant metadata servers**.

Ques 10. Illustrate, with diagrams or pseudocode, how synchronous checkpointing and recovery mechanisms work in distributed databases. Evaluate the strengths and weaknesses of synchronous vs asynchronous checkpointing.

Answer –

How it works:

- All nodes stop updates.
- Save state to stable storage at same logical time.
- System resumes.

Strength: no inconsistent checkpoints.
Weakness: delays entire system → lower performance. Asynchronous: faster but inconsistent states possible, requires message logging.

Ques 11. Multiple IoT devices interact with a centralized OS in a smart home. a) Propose a process scheduling strategy (with algorithm selection and justification) to prioritize security device interrupts (e.g., camera motion detection) over less critical tasks (e.g., lighting adjustments). b) Outline the inter-process communication methods suitable for this environment and briefly justify their use.

Answer –

a) Scheduling Strategy

Use **Priority Scheduling**:

- High priority → security devices (camera motion, alarms).
- Medium → sensors
- Low → lighting/AC tasks.

b) IPC Methods

- **Message Queues** → safe async communication
- **Shared Memory with locks** → fast for sensor data
- **Sockets** → device-to-central hub communication

Ques 12. Select either LINUX or Windows OS (case study from self learning) and implement a Python program that demonstrates a basic system call (e.g., file creation, reading, or threading). Include output screenshots/code and comment on its relevance to OS architecture as learned in this course.

Answer -

```
import os
path = "/tmp/demo.txt"

fd = os.open(path, os.O_CREAT | os.O_WRONLY) os.write(fd,
b"Hello from OS system call!") os.close(fd)

fd = os.open(path, os.O_RDONLY) data =
os.read(fd, 100) os.close(fd)

print("File created successfully!") print("Content:",
data.decode())
```

OUTPUT: