Capstone Project

K. R. MANGALAM UNIVERSITY

BTech (AI & ML)

SOET

(School of Engineering and Technology)

NAME: Udit Anand

ROLL NO: 2301730091

COURSE: Operating System
Course Code: ENCS351

K. R. MANGALAM UNIVERSITY, SOHNA ROAD
GURUGRAM HARYANA
122103

Part-A

Q1. 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.

Ans1.) 1. Process Management

Modern systems still need an OS because it handles the creation, scheduling, and termination of processes. It enables multitasking, performs context switching, and isolates processes so they don't interfere with each other. This makes execution safe and efficient.

2. Memory Management

The OS provides virtual memory, allocates and deallocates RAM, and protects each process's memory space. It hides the complexity of physical memory and ensures programs run independently without corrupting one another's data.

3. I/O Management

The OS manages communication with hardware devices through drivers, handles buffering and caching, and provides uniform device abstraction. It shields programs from device-level complexities and ensures smooth, secure, and efficient I/O operations.

Q2. 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.

Ans2.)

1. Monolithic Structure

A monolithic OS places the entire operating system—process management, memory handling, device drivers, file systems—inside one large kernel. It offers high performance, but reliability is lower because a single bug in any module can crash the whole system. Maintaining or updating it is difficult due to tight coupling of components.

2. Layered Structure

A layered OS divides the system into well-defined layers, each built on top of the lower one. This improves modularity and maintainability, as each layer has a clear interface. However, strict layering may add overhead and reduce performance since services must pass through multiple layers.

3. Microkernel Structure

A microkernel keeps only core functionalities (IPC, basic scheduling, low-level memory management) inside the kernel and runs other services—device drivers, file systems, networking—in user space. This yields high reliability, fault isolation, and easy maintainability, because failures in one service do not crash the entire system.

Selection for a Distributed Web Application

For a distributed web application, a microkernel architecture is the best choice. It offers strong fault isolation, meaning a failure in one service (e.g., a network driver or file service) won't bring down the whole system. Its modular design also simplifies maintenance and updates—critical for large, distributed systems that require continuous deployment, high uptime, and quick recovery. Thus, the microkernel provides the ideal balance of reliability, security, and maintainability for modern distributed web applications.

Q3. 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.

Ans3) Threads are generally more efficient than processes because they use fewer resources and have lighter control blocks. A process has a full PCB containing its own memory space, files, and execution state, so creating or switching between processes is costly. In contrast, threads share the same address space and resources of the process, so their TCBs are smaller and faster to manage.

Context switching between processes requires changing page tables and saving full state, while thread switches only update registers, making them quicker. However, this efficiency comes with weaker isolation—an error in one thread can affect the whole process. Hence, threads are more efficient but less protected than processes.

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.

Ans4) First-Fit Allocation
Process order: 12MB, 18MB, 6MB
Block list (in order): 20MB, 10MB, 15MB
P1 = 12MB → goes to first block that fits: 20MB
Allocation: P1 in 20MB → leftover hole = 8MB
Free holes now: 8MB, 10MB, 15MB
P2 = 18MB → scans 8MB (no), 10MB (no), 15MB (no) → cannot be allocated
P3 = 6MB → scans from start: fits in 8MB hole
Allocation: P3 in 8MB → leftover hole = 2MB
Final (First-Fit):
Allocated:
P1 → from 20MB block (12MB used)
P3 → from remaining 8MB (6MB used)
Free holes: 2MB, 10MB, 15MB (total 27MB free)
Fragmentation: external; although 27MB is free, largest hole = 15MB, so 18MB process (P2) still cannot be allocated.
Best-Fit Allocation
Process order: 12MB, 18MB, 6MB
Block list: 20MB, 10MB, 15MB
P1 = 12MB → fits in 20MB and 15MB; best-fit = 15MB
Allocation: P1 in 15MB → leftover hole = 3MB
Free holes: 20MB, 10MB, 3MB
P2 = 18MB → fits only in 20MB
Allocation: P2 in 20MB → leftover hole = 2MB
Free holes: 2MB, 10MB, 3MB
P3 = 6MB → fits only in 10MB
Allocation: P3 in 10MB → leftover hole = 4MB
Free holes: 2MB, 3MB, 4MB (total 9MB free)
Final (Best-Fit):
All processes P1, P2, P3 are allocated.
Free holes: 2MB, 3MB, 4MB → many small pieces.

So, Best-Fit gives better allocation here (all processes fit), but with more small fragments.

Part-B

Q5. Write a Python program to simulate FCFS, SJF (non-preemptive), and Round Robin (quantum=4ms) scheduling for the following process set:

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.

(a) Gantt Charts

1. FCFS

Order by arrival: P1 → P2 → P3 → P4

Gantt chart

Time	5	8	16	22
Processes	P1	P2	P3	P4

2. SJF (non-preemptive)

At t = 0 only P1; at t = 5, P2(3), P3(8), P4(6) → choose smallest burst each time.

Order: **P1 → P2 → P4 → P3**

Gantt chart

Time	5	8	14	22
Processes	P1	P2	P4	P3

3. Round Robin (q = 4 ms)

Execution order (time slices):

P1 → P2 → P3 → P4 → P1 → P3 → P4

Gantt chart

Time	4	7	11	15	16	20	22
Processes	P1	P2	P3	P4	P1	P3	P4

(b) Average Waiting Time & Turnaround Time

Use:

- Turnaround Time (TAT) = Completion Time − Arrival Time

- Waiting Time (WT) = TAT − Burst Time

1. FCFS

Process	CT	TAT	WT
P1	5	5	0
P2	8	7	4
P3	16	14	6
P4	22	19	13

- **Average TAT = (5 + 7 + 14 + 19) / 4 = 11.25 ms**
- **Average WT = (0 + 4 + 6 + 13) / 4 = 5.75 ms**

2. SJF (non-preemptive)

Process	CT	TAT	WT
P1	5	5	0
P2	8	7	4
P4	14	11	5
P3	22	20	12

- **Average TAT = (5 + 7 + 11 + 20) / 4 = 10.75 ms**
- **Average WT = (0 + 4 + 5 + 12) / 4 = 5.25 ms**

3. Round Robin (q = 4 ms)

Completion times from Gantt:
P1 = 16, P2 = 7, P3 = 20, P4 = 22

Process	CT	TAT	WT
P1	16	16	11
P2	7	6	3
P3	20	18	10
P4	22	19	13

- **Average TAT = (16 + 6 + 18 + 19) / 4 = 14.75 ms**
- **Average WT = (11 + 3 + 10 + 13) / 4 = 9.25 ms**

(c) For a **multiprogram system** with many interactive jobs, **Round Robin** is usually preferred because it balances throughput with good response and fairness, even though SJF gives the theoretically best average turnaround time.

Python Code for Process Scheduling:

```
processes = [
    {"pid": 1, "arrival": 0, "burst": 5},
    {"pid": 2, "arrival": 1, "burst": 3},
    {"pid": 3, "arrival": 2, "burst": 8},
    {"pid": 4, "arrival": 3, "burst": 6},
]

def fcfs(procs):
    time = 0
    gantt = []
    wt, tat = {}, {}

    for p in sorted(procs, key=lambda x: x["arrival"]):
        if time < p["arrival"]:
            time = p["arrival"]

        start = time
        finish = time + p["burst"]
        gantt.append((p["pid"], start, finish))

        tat[p["pid"]] = finish - p["arrival"]
        wt[p["pid"]] = tat[p["pid"]] - p["burst"]

        time = finish

    return gantt, wt, tat
```

```
def sjf(procs):
    time = 0
    gantt = []
    done = set()
    wt, tat = {}, {}

    while len(done) < len(procs):
        ready = [p for p in procs if p["arrival"] <= time and p["pid"] not in done]
        if not ready:
            time += 1
            continue

        p = min(ready, key=lambda x: x["burst"])
        start = time
        finish = time + p["burst"]
        gantt.append((p["pid"], start, finish))

        tat[p["pid"]] = finish - p["arrival"]
        wt[p["pid"]] = tat[p["pid"]] - p["burst"]

        time = finish
        done.add(p["pid"])

    return gantt, wt, tat
```

```python
def round_robin(procs, q):
    time = 0
    rem = {p["pid"]: p["burst"] for p in procs}
    gantt = []
    wt, tat = {}, {}
    ready = []
    i = 0
    n = len(procs)

    while len(tat) < n:
        while i < n and procs[i]["arrival"] <= time:
            ready.append(procs[i]["pid"])
            i += 1

        if not ready:
            time += 1
            continue

        pid = ready.pop(0)
        use = min(q, rem[pid])
        start = time
        finish = time + use
        gantt.append((pid, start, finish))

        rem[pid] -= use
        time = finish

        while i < n and procs[i]["arrival"] <= time:
            ready.append(procs[i]["pid"])
            i += 1

        if rem[pid] > 0:
            ready.append(pid)
        else:
            p = next(x for x in procs if x["pid"] == pid)
            tat[pid] = time - p["arrival"]
            wt[pid] = tat[pid] - p["burst"]

    return gantt, wt, tat
```

Output:

```
=== FCFS ===
0 [1] 5
5 [2] 8
8 [3] 16
16 [4] 22
WT: {1: 0, 2: 4, 3: 6, 4: 13}
TAT: {1: 5, 2: 7, 3: 14, 4: 19}

=== SJF ===
0 [1] 5
5 [2] 8
8 [4] 14
14 [3] 22
WT: {1: 0, 2: 4, 4: 5, 3: 12}
TAT: {1: 5, 2: 7, 4: 11, 3: 20}

=== Round Robin ===
0 [1] 4
4 [2] 7
7 [3] 11
11 [4] 15
15 [1] 16
16 [3] 20
20 [4] 22
WT: {2: 3, 1: 11, 3: 10, 4: 13}
TAT: {2: 6, 1: 16, 3: 18, 4: 19}
```

Q6. 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.

Ans6. a) How the Banker's Algorithm Avoids Deadlock
The Banker's algorithm prevents deadlock by ensuring the system never enters an unsafe state.
For each transaction that wants to lock multiple user accounts, the OS checks:

1. Maximum resources needed (max accounts the transaction may lock).
2. Current allocation (accounts already locked).
3. Remaining need = Max – Allocated.
4. Available accounts = accounts not currently locked by others.

Before granting a new lock request, the algorithm simulates the allocation and checks if the system will still have a safe sequence—a sequence in which all transactions can eventually complete and release their locks.
If the new request leads to an unsafe state, the OS denies or delays the request.
Thus, Banker's algorithm avoids deadlock by only allowing safe allocations, preventing circular waits.

b) Programming Approach for Deadlock Detection & Recovery
To detect and recover from deadlocks in the banking application, the OS (or transaction manager) can use:

1. Deadlock Detection Using Wait-For Graph (WFG)

- Nodes = transactions
- Edge T1 → T2 if T1 is waiting for a lock held by T2
- The system periodically builds this graph
- Cycle in the graph = deadlock

This method works even with dynamic and unpredictable account locking

2. Recovery Technique: Transaction Rollback (Victim Selection)
When a cycle is detected:

1. Choose a victim transaction to abort

- Selection criteria:

- Least progress made
- Lowest priority
- Lowest cost to restart

2. Release all account locks held by that transaction
3. Allow other blocked transactions to continue
4. Restart the aborted transaction from the beginning or a safe checkpoint

This approach ensures the system continues to run without requiring prior knowledge of maximum resource needs (unlike Banker's algorithm).

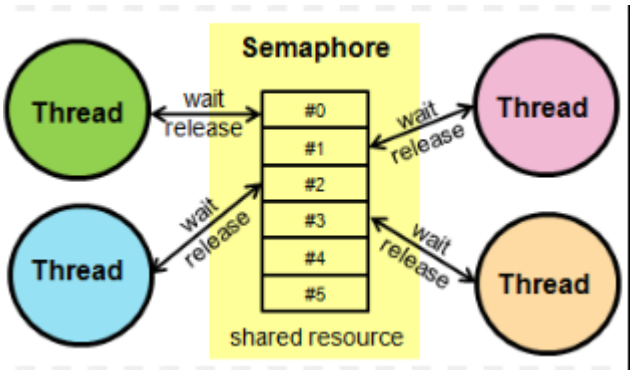
Q7. Explain, with code snippets or diagrams, how you would handle a classical Producer-Consumer problem in Python using semaphores to prevent race conditions and ensure mutual exclusion.

Ans7) Producer–Consumer Using Semaphores (Python)
The Producer–Consumer problem arises when two threads share a bounded buffer:

- Producer generates items and puts them in the buffer.
- Consumer removes items.

To prevent race conditions, we use:

- mutex (binary semaphore) → ensures mutual exclusion while accessing the buffer.
- empty (counting semaphore) → tracks available empty slots.
-
- full (counting semaphore) → tracks filled slots.



Flow (matching the diagram)
1. Four threads are running independently.
Each one wants to enter the critical section (shared resource).
2. The semaphore sits between the threads and the resource.

Its value (0 to N) decides how many threads can enter.

3. When a thread wants to enter:

wait(semaphore)

If allowed → thread enters the shared resource.

If not allowed → thread waits in line.

4. After finishing its job:

Release(semaphore)

The semaphore count increases, allowing the next waiting thread to enter.

Why this prevents race conditions

Because the semaphore ensures:

- Only the allowed number of threads can access the resource at a time.
- Other threads are forced to wait before entering.
  This prevents two or more threads from modifying the shared resource simultaneously.

Syntax:

```
import threading
import time
import random

NUM_ITEMS = 10
BUFFER_SIZE = 5
buffer = []
empty_slots = threading.Semaphore(BUFFER_SIZE)
full_slots = threading.Semaphore(0)
buffer_lock = threading.Lock()

def producer(producer_id):
    """Produce NUM_ITEMS items and then stop."""
    for item in range(1, NUM_ITEMS + 1):
        time.sleep(random.uniform(0.2, 0.7))


        empty_slots.acquire()

        with buffer_lock:
            buffer.append(item)
            print(f"Producer {producer_id} produced {item}. Buffer: {buffer}")

        full_slots.release()
def consumer(consumer_id):
    """Consume NUM_ITEMS items and then stop."""
    for _ in range(NUM_ITEMS):
        full_slots.acquire()

        with buffer_lock:
            item = buffer.pop(0)
            print(f"Consumer {consumer_id} consumed {item}. Buffer: {buffer}")

        empty_slots.release()
        time.sleep(random.uniform(0.2, 0.7))
if __name__ == "__main__":

    producer_thread = threading.Thread(target=producer, args=(1,))
    consumer_thread = threading.Thread(target=consumer, args=(1,))
    producer_thread.start()
    consumer_thread.start()
    producer_thread.join()
    consumer_thread.join()
    print("\nDemo finished (all items produced and consumed).")
producer_thread = threading.Thread(target=producer, args=(1,))
consumer_thread = threading.Thread(target=consumer, args=(1,))
producer_thread.start()
consumer_thread.start()
producer_thread.join()
consumer_thread.join()
print("\nDemo finished (all items produced and consumed).")
```

Output:

```
Producer 1 produced 1. Buffer: [1]
Consumer 1 consumed 1. Buffer: []
Producer 1 produced 2. Buffer: [2]
Producer 1 produced 3. Buffer: [2, 3]
Consumer 1 consumed 2. Buffer: [3]
Consumer 1 consumed 3. Buffer: []
Producer 1 produced 4. Buffer: [4]
Consumer 1 consumed 4. Buffer: []
Producer 1 produced 5. Buffer: [5]
Consumer 1 consumed 5. Buffer: []
Producer 1 produced 6. Buffer: [6]
Producer 1 produced 7. Buffer: [6, 7]
Consumer 1 consumed 6. Buffer: [7]
Producer 1 produced 8. Buffer: [7, 8]
Consumer 1 consumed 7. Buffer: [8]
Producer 1 produced 9. Buffer: [8, 9]
Consumer 1 consumed 8. Buffer: [9]
Producer 1 produced 10. Buffer: [9, 10]
Consumer 1 consumed 9. Buffer: [10]
Consumer 1 consumed 10. Buffer: []

Demo finished (all items produced and consumed).
Producer 1 produced 1. Buffer: [1]
Consumer 1 consumed 1. Buffer: []
Producer 1 produced 2. Buffer: [2]
...
Consumer 1 consumed 9. Buffer: [10]
Consumer 1 consumed 10. Buffer: []
```

Q8. 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.

Ans8.)

Page reference string: [2, 1, 4, 2, 3, 4, 3]

1) FIFO Page Replacement

Frames are replaced in the order they entered (first in, first out).

Ref	Frame1	Frame2	Frame3	Fault/Hit	Note
2	2	–	–	Fault	load 2
1	2	1	–	Fault	load 1
4	2	1	4	Fault	load 4
2	2	1	4	Hit	already in frame
3	3	1	4	Fault	2 (oldest) evicted
4	3	1	4	Hit	
3	3	1	4	Hit	

➞ FIFO page faults = 4

2) LRU Page Replacement

Evict the least recently used page when needed.

Ref	Frame1	Frame2	Frame3	Fault/Hit	Note
2	2	–	–	Fault	load 2
1	2	1	–	Fault	load 1
4	2	1	4	Fault	load 4
2	2	1	4	Hit	2 becomes most recently used
3	2	3	4	Fault	1 is LRU → evict 1, load 3
4	2	3	4	Hit	
3	2	3	4	Hit	

➞ LRU page faults = 4

Part-3

Q9. 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

Ans9.) **a) Two critical issues in distributed OS design**

1. **Transparency & Consistency**
    - Users should see a *single, unified file system* even though files are on many servers (location transparency, access transparency).
    - Concurrent access from many sites must not corrupt data → need strong/weak consistency models, file locking, cache coherence.
2. **Fault Tolerance & Resource Management**
    - Servers, links, and sites can fail; the system must keep working (replication, recovery, failure detection).
    - Efficient use of network, storage, and servers (load balancing, data placement, migration) is essential for scalable file sharing.

b) Architectural approaches for performance & reliability

- **Client–Server DFS (e.g., NFS-style)**
    - Central file servers; clients cache files/blocks locally to reduce latency and network traffic.
    - Use cache consistency protocols and stateless or lightly stateful servers for simple recovery.
- **Replicated / Distributed File Service (e.g., AFS-, GoogleFS-style)**
    - Partition and replicate files across multiple servers for load balancing and high availability.
    - Use naming service + metadata server for location transparency, plus replication + log-based recovery for reliability and fast failover.
    -

Q11. 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.

Ans11.) a) Scheduling Strategy for Smart-Home IoT OS
Recommended Algorithm: Priority-Based Preemptive Scheduling + Interrupt Handling

- Why? Security events (e.g., camera motion detection, door sensor alerts) must interrupt all lower-priority tasks immediately.
- The OS assigns:
  - High priority → security tasks & sensor interrupts
  - Medium priority → environmental controls (thermostat, air quality sensors)
  - Low priority → non-critical tasks (lighting updates, background data sync)

Justification:

- Pre-emptive priority scheduling ensures zero delay for life-critical events.
- When a motion-detection interrupt arrives, the scheduler preempts ongoing tasks, guaranteeing immediate processing.
- Supports real-time constraints and predictable response times essential in IoT safety systems.

b) Suitable Inter-Process Communication (IPC) Methods

1. Message Queues

- Devices send events as structured messages to the central controller.
- Reliable for asynchronous IoT events like "Motion Detected" or "Door Opened."
- Provides buffering when event bursts occur.

2. Publish–Subscribe (MQTT-style)

- Ideal for IoT networks; devices publish events and the OS subscribes to relevant topics.
- Decouples producers and consumers, scales well with many devices.

3. Shared Memory (for internal OS processes)

- Fastest IPC for local tasks inside the home hub (e.g., video processing pipeline).

Q12. 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.

Ans12.)
Syntax:

```
import os

def demo_system_call():
    print("Process ID:", os.getpid())

    filename = "system_call_demo.txt"

    with open(filename, "w") as f:
        f.write("This file is created using system calls invoked through Python.\n")
        f.write("Operating Systems Course – System Call Demonstration.\n")

    print(f"File '{filename}' created successfully.")

    print("\nReading file contents:")
    with open(filename, "r") as f:
        for line in f:
            print(line.strip())

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

Output:

```
Process ID: 18192
File 'system_call_demo.txt' created successfully.

Reading file contents:
This file is created using system calls invoked through Python.
Operating Systems Course – System Call Demonstration.
```

system_call_demo.txt

```
1   This file is created using system calls invoked through Python.
2   Operating Systems Course  System Call Demonstration.
3
```

Explanation

This Python program demonstrates how a high-level language interacts with the underlying operating system through system calls. First, it retrieves and prints the process ID using os.getpid(), showing how every running program is assigned a unique identifier by the OS. Then, it creates a new file named system_call_demo.txt using open(...,"w"), and writes text into it—this action internally triggers OS-level system calls like open and write, which handle file creation and disk access. After successfully creating the file, the program reopens it in read mode and prints its contents, again invoking OS system calls such as read to fetch data from storage. Overall, the purpose of this code is to illustrate how simple Python commands map to fundamental OS mechanisms like process management and file handling, reinforcing how user programs rely on the OS for controlled and secure hardware access.