



**K.R. MANGALAM UNIVERSITY**  
THE COMPLETE WORLD OF EDUCATION

## **OPERATING SYSTEM CAPSTONE PROJECT**

*Submitted in partial fulfilment of the requirement of the degree*

**BACHELORS OF TECHNOLOGY**

In

**CSE With Specialization (AI & ML)**

To

**K.R Mangalam University**

By

**Yash Kumar (2301730114)**

**Under the supervision of**

**Mr. Mohammad Aijaz**



**School of Engineering and Technology  
K.R Mangalam University, Gurugram- 122103, India**

## Part A: Fundamental Concepts

### Question 1: Why Operating Systems Matter Despite Hardware Advances

#### Simplified Answer:

Even though modern hardware is very fast, it's still difficult for software applications to work directly with it. An operating system (OS) acts as a bridge between hardware and software. It makes things simpler and safer.

#### Key Services of an OS:

OS Service	What It Does	Example
<b>Process Management</b>	Creates, schedules, and manages programs while they run	Lets 10 applications run at the same time by sharing CPU time
<b>Memory Management</b>	Allocates and manages RAM; keeps programs from crashing into each other's memory	Each program gets its own memory space so they don't interfere
<b>I/O Management</b>	Controls communication between programs and devices (disk, printers, network)	When you save a file, the OS controls writing data to the hard drive

#### Real-World Example:

When you run 10 applications at once, the OS does three things:

- Divides the CPU time among them using scheduling algorithms
- Gives each program its own virtual memory space
- Manages input/output requests to devices

### Question 2: Comparing OS Structures - Monolithic vs Layered vs Microkernel

#### Simplified Comparison:

Feature	Monolithic	Layered	Microkernel
<b>Structure</b>	All services in kernel (big single program)	Services divided into organized layers	Only essential services in kernel; others in user space
<b>Advantages</b>	Fast performance	Better organization and maintainability	High reliability and security
<b>Disadvantages</b>	If one service crashes, entire OS crashes; hard to fix	More layers = slower performance	More complex; requires more communication

## **Best Choice for Distributed Web Applications: Microkernel**

### **Why Microkernel?**

1. **Reliability:** If a device driver or file service crashes, the core OS doesn't fail—only that service stops
2. **Security:** Less code runs in kernel mode (dangerous), so fewer security risks
3. **Fault Tolerance:** Web applications need to keep running even if parts fail
4. **Scalability:** Services can be updated without restarting the entire system

Example: If a network driver crashes in a Microkernel, your web server keeps running. In a Monolithic OS, the entire system would crash.

## **Question 3: Are Threads More Efficient Than Processes?**

### **Critical Analysis:**

This statement is **generally TRUE**, but with important exceptions.

### **Why Threads Are More Efficient:**

Aspect	Processes	Threads
<b>Memory Usage</b>	Each has separate memory space (heavy)	Share memory with their process (light)
<b>Context Switching</b>	Slow (save/restore entire memory and state)	Fast (save/restore only CPU registers)
<b>Creation Time</b>	Slow	Fast
<b>Communication</b>	Requires Inter-Process Communication (IPC) - complex	Direct shared memory - simple
<b>CPU Utilization</b>	Multi-processing increases CPU use	Multi-threading increases CPU use more efficiently

### **Why Threads Can Be Problematic:**

1. **Synchronization Issues:** Threads share memory, so they need locks to prevent data corruption
2. **Crash Risk:** If one thread crashes badly, it can crash the entire process (affecting all threads)
3. **Debugging Difficulty:** Hard to track bugs in multi-threaded programs

**Conclusion:** Threads are more efficient for most tasks, but processes provide better isolation and safety.

## Question 4: Memory Allocation - First-Fit vs Best-Fit

### Problem Setup:

- Processes need: 12 MB, 18 MB, 6 MB
- Available memory blocks: 20 MB, 10 MB, 15 MB

### First-Fit Allocation:

Allocate each process to the first block large enough.

Process	Required	Block Chosen	Leftover
P1	12 MB	20 MB	8 MB remaining
P2	18 MB	15 MB	Not enough! (block too small)
P3	6 MB	10 MB	4 MB remaining

**Result:** P2 cannot be allocated. Free fragments: 8 MB, 10 MB, 4 MB (Total wasted: 22 MB)

### Best-Fit Allocation:

Allocate each process to the smallest block that fits (to minimize leftover space).

Process	Required	Best Block	Leftover
P1	12 MB	15 MB	3 MB remaining
P2	18 MB	20 MB	2 MB remaining
P3	6 MB	10 MB	4 MB remaining

**Result:** All processes allocated! Free fragments: 3 MB, 2 MB, 4 MB (Total wasted: 9 MB)

### Fragmentation Analysis:

- **First-Fit:** Creates larger fragments (8 MB, 10 MB, 4 MB) - harder to reuse
- **Best-Fit:** Creates smaller fragments (3 MB, 2 MB, 4 MB) - better fragmentation but slower search

**Conclusion:** Best-Fit performs better here, but First-Fit is usually faster in practice

## Part B: Scheduling & Synchronization

### Question 5: CPU Scheduling Algorithms Comparison

#### Given Process Data:

- P1: BT = 5, AT = 0
- P2: BT = 3, AT = 1
- P3: BT = 8, AT = 2
- P4: BT = 6, AT = 3

## CODE FOR FCFS,SJF(non- preemptive),RR(QT=4ms):

```

from collections import deque
# PROCESSES: (name, burst_time, arrival_time)
PROCESSES = [
    ("P1", 5, 0),
    ("P2", 3, 1),
    ("P3", 8, 2),
    ("P4", 6, 3)
]
def fcfs(processes):
    t = 0
    result = []
    for name, bt, at in sorted(processes, key=lambda x: x[2]):
        if t < at:
            t = at
        start = t
        finish = t + bt
        t = finish
        result.append((name, start, finish, at, bt))
    return result
def sjf_non_preemptive(processes):
    t = 0
    remaining = processes[:]
    result = []
    while remaining:
        available = [p for p in remaining if p[2] <= t]
        if not available:
            t = min(p[2] for p in remaining)
        available = [p for p in remaining if p[2] <= t]
        name, bt, at = min(available, key=lambda x: x[1])
        start = t
        finish = t + bt
        t = finish
        result.append((name, start, finish, at, bt))
        remaining.remove((name, bt, at))
    return result
def round_robin(processes, quantum=4):
    procs = sorted(processes, key=lambda x: x[2])
    n = len(procs)
    remaining_bt = [bt for _, bt, _ in procs]
    t = 0
    i = 0
    q = deque()
    completed = [False] * n
    completion_time = [0] * n
    gantt = []
    while True:
        while i < n and procs[i][2] <= t:
            q.append(i)
            i += 1
        if not q:
            if i < n:
                t = procs[i][2]
                continue
            else:
                break
        idx = q.popleft()
        name, bt, at = procs[idx]
        run = min(quantum, remaining_bt[idx])
        start = t
        t += run
        finish = t
        gantt.append((name, start, finish))
        remaining_bt[idx] -= run
        while i < n and procs[i][2] <= t:
            q.append(i)
            i += 1
        if remaining_bt[idx] > 0:
            q.append(idx)
        else:
            completed[idx] = True
            completion_time[idx] = t
        if all(completed):
            break
    result = [(procs[idx][0], procs[idx][2], completion_time[idx], procs[idx][1]) for idx in range(n)]
    return gantt, result
def compute_metrics(result):
    wt, tat = [], []
    for name, at, ct, bt in result:
        turnaround = ct - at
        waiting = turnaround - bt
        wt.append(waiting)
        tat.append(turnaround)
    return wt, tat, sum(wt)/len(wt), sum(tat)/len(tat)
# RUN ALL ALGORITHMS
print("== FCFS ==")
fcfs_res = fcfs(PROCESSES)
print("Gantt: " + "\n".join([f"[{name}:{start}-{finish}]" for name, start, finish, _, _ in fcfs_res]))
wt, tat, avg_wt, avg_tat = compute_metrics([(r[0], r[3], r[2], r[1]) for r in fcfs_res])
print(f"Avg WT: {avg_wt:.2f}, Avg TAT: {avg_tat:.2f}\n")
print("== SJF ==")
sjf_res = sjf_non_preemptive(PROCESSES)
print("Gantt: " + "\n".join([f"[{name}:{start}-{finish}]" for name, start, finish, _, _ in sjf_res]))
wt, tat, avg_wt, avg_tat = compute_metrics([(r[0], r[3], r[2], r[1]) for r in sjf_res])
print(f"Avg WT: {avg_wt:.2f}, Avg TAT: {avg_tat:.2f}\n")
print("== ROUND ROBIN (q=4) ==")
rr_gantt, rr_res = round_robin(PROCESSES)
print("Gantt: " + "\n".join([f"[{name}:{start}-{finish}]" for name, start, finish in rr_gantt]))
wt, tat, avg_wt, avg_tat = compute_metrics(rr_res)
print(f"Avg WT: {avg_wt:.2f}, Avg TAT: {avg_tat:.2f}")

```

## a) Gantt Charts

**FCFS (First-Come-First-Served):**

Timeline: [P1: 0-5] [P2: 5-8] [P3: 8-16] [P4: 16-22]

**SJF (Shortest Job First - Non-preemptive):**

Timeline: [P1: 0-5] [P2: 5-8] [P4: 8-14] [P3: 14-22]

**Round Robin (Quantum = 4ms):**

Timeline: [P1: 0-4] [P2: 4-7] [P3: 7-11] [P4: 11-15] [P3: 15-19] [P4: 19-22]

## b) Average waiting and turnaround times

Output:

```
==== FCFS ====
Gantt: [P1:0-5] [P2:5-8] [P3:8-16] [P4:16-22]
Avg WT: 4.00, Avg TAT: 11.25

==== SJF ====
Gantt: [P1:0-5] [P2:5-8] [P4:8-14] [P3:14-22]
Avg WT: 4.00, Avg TAT: 10.75

==== ROUND ROBIN (q=4) ====
Gantt: [P1:0-4] [P2:4-7] [P3:7-11] [P4:11-15] [P1:15-16] [P3:16-20] [P4:20-22]
Avg WT: 9.25, Avg TAT: 14.75
```

## c) Algorithm Comparison

**Best for Multiprogrammed Systems: SJF**

Algorithm	Throughput	Fairness	Waiting Time
FCFS	Lower	Fair	High for long jobs
SJF	Highest	Unfair (long jobs wait)	Lowest
Round Robin	Medium	Fair	Medium

**Recommendation:** Use **SJF** for batch systems (best throughput) or **Round Robin** for interactive systems (better fairness).

## Question 6: Deadlock Handling in Banking Systems

### a) Banker's Algorithm - Deadlock Avoidance

#### How It Works (Simple Explanation):

1. Each transaction declares maximum resources it needs upfront
2. When requesting resources, the OS asks: "If I give this resource, will the system still be safe?"
3. "Safe" means all transactions can eventually complete without deadlock
4. Only grant requests that keep the system in a safe state

**Example:** Bank transfers involving multiple accounts

- Transaction T1 needs accounts (A, B, C) - declares max need upfront
- T1 requests to lock account A - OS checks: "Can T1 and others finish if A is locked?"
- If YES → Lock granted
- If NO → Lock delayed

### b) Deadlock Detection & Recovery in Programming

#### Detection Mechanism:

The OS periodically checks for cycles in the "Wait-for Graph":

- If P1 waits for P2, P2 waits for P3, P3 waits for P1 → **Cycle = Deadlock!**

#### Recovery Approach:

```
from collections import defaultdict, deque

def detect_deadlock(wait_for):
    """Detect cycle in wait-for graph using Kahn's algorithm"""
    graph = defaultdict(list)
    indegree = defaultdict(int)

    # Build graph and indegree
    for p, waits in wait_for.items():
        for q in waits:
            graph[p].append(q)
            indegree[q] += 1
            if p not in indegree:
                indegree[p] = 0

    # BFS with nodes having indegree 0
    q = deque([p for p in indegree if indegree[p] == 0])
    visited = 0

    while q:
        node = q.popleft()
        visited += 1
        for neighbor in graph[node]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                q.append(neighbor)

    deadlock = visited < len(indegree)
    print("DEADLOCK DETECTED!" if deadlock else "No deadlock")
    return deadlock

def recover(wait_for, victim):
    """Abort victim transaction"""
    if victim in wait_for:
        del wait_for[victim]
    for p in list(wait_for.keys()):
        if victim in wait_for[p]:
            wait_for[p].remove(victim)
    print(f"RECOVERED: Aborted {victim}")

# TEST CASE: Deadlock (T1→T2→T3→T1)
wait_for = {"T1": ["T2"], "T2": ["T3"], "T3": ["T1"]}
print("Initial state:")
detect_deadlock(wait_for)
recover(wait_for, "T3")
print("\nAfter recovery:")
detect_deadlock(wait_for)
```

Output:

Initial state:

DEADLOCK DETECTED!

RECOVERED: Aborted T3

After recovery:

No deadlock

## Recovery Strategies:

1. **Abort Victim:** Stop one transaction, release its resources
2. **Rollback:** Undo partial work and restart
3. **Preempt Resources:** Force one transaction to release locks
4. **Restart Later:** Put transaction back in queue for retry

**Banking Example:** If T1 and T2 deadlock over accounts:

- Abort T2 (cheaper operation)
- Release all locks held by T2
- Rollback T2's changes
- Restart T2 when system is free

## Question 7: Producer-Consumer with Semaphores

**Problem:** Producer and Consumer share a buffer. Need to prevent race conditions.

### Simple Explanation:

- **Producer** creates data and puts it in buffer
- **Consumer** takes data from buffer and uses it
- **Problem:** Both might access buffer simultaneously, corrupting data
- **Solution:** Use semaphores (special counters) to synchronize

### CODE:

```
import threading
import time
import random

BUFFER_SIZE = 5
buffer = []
empty = threading.Semaphore(BUFFER_SIZE)
full = threading.Semaphore(0)
mutex = threading.Semaphore(1)

def producer():
    for i in range(10):
        time.sleep(random.uniform(0.1, 0.3))
        item = f"Item-{i}"

        empty.acquire()
        mutex.acquire()
        buffer.append(item)
        print(f"Producer added {item} | Buffer: {len(buffer)}/{BUFFER_SIZE}")
        mutex.release()
        full.release()

def consumer():
    for i in range(10):
        time.sleep(random.uniform(0.2, 0.5))

        full.acquire()
        mutex.acquire()
        item = buffer.pop(0)
        print(f"Consumer removed {item} | Buffer: {len(buffer)}/{BUFFER_SIZE}")
        mutex.release()
        empty.release()

# RUN
print("Starting Producer-Consumer...")
p_thread = threading.Thread(target=producer)
c_thread = threading.Thread(target=consumer)

p_thread.start()
c_thread.start()

p_thread.join()
c_thread.join()
print("COMPLETED!")
```

Output:

```
Starting Producer-Consumer...
Producer added Item-0 | Buffer: 1/5
Producer added Item-1 | Buffer: 2/5
Consumer removed Item-0 | Buffer: 1/5
Producer added Item-2 | Buffer: 2/5
Producer added Item-3 | Buffer: 3/5
Consumer removed Item-1 | Buffer: 2/5
Producer added Item-4 | Buffer: 3/5
Producer added Item-5 | Buffer: 4/5
Consumer removed Item-2 | Buffer: 3/5
Producer added Item-6 | Buffer: 4/5
Consumer removed Item-3 | Buffer: 3/5
Producer added Item-7 | Buffer: 4/5
Producer added Item-8 | Buffer: 5/5
Consumer removed Item-4 | Buffer: 4/5
Producer added Item-9 | Buffer: 5/5
Consumer removed Item-5 | Buffer: 4/5
Consumer removed Item-6 | Buffer: 3/5
Consumer removed Item-7 | Buffer: 2/5
Consumer removed Item-8 | Buffer: 1/5
Consumer removed Item-9 | Buffer: 0/5
COMPLETED!
```

### How Semaphores Prevent Race Conditions:

Semaphore	Purpose	Value Change
empty	Counts empty slots in buffer	Decreases when producer adds; increases when consumer takes
full	Counts filled slots in buffer	Increases when producer adds; decreases when consumer takes
mutex	Ensures only one thread accesses buffer at a time	Acquired when entering critical section; released after

**Safety Guarantee:** Only one thread accesses the buffer at a time, so no corruption!

## Question 8: Page Replacement - FIFO vs LRU

**Given Page Access Sequence:** [2, 1, 4, 2, 3, 4, 3]

**Frame Size:** 3 (can hold 3 pages at once)

### FIFO (First-In-First-Out)

Replace the oldest page in memory.

Step	Page	Frames After	Page Fault?
1	2	[2]	Yes
2	1	[2, 1]	Yes
3	4	[2, 1, 4]	Yes
4	2	[2, 1, 4]	No (already there)
5	3	[1, 4, 3]	Yes (remove 2, oldest)
6	4	[1, 4, 3]	No (already there)
7	3	[1, 4, 3]	No (already there)

**Total FIFO Page Faults:** 4

### LRU (Least Recently Used)

Replace the page not used for the longest time.

Step	Page	Frames After	Page Fault?
1	2	[2]	Yes
2	1	[2, 1]	Yes
3	4	[2, 1, 4]	Yes
4	2	[1, 4, 2]	No (mark 2 as recent)

5	3	[4, 2, 3]	Yes (remove 1, least recent)
6	4	[4, 2, 3]	No (already there)
7	3	[4, 2, 3]	No (already there)

**Total LRU Page Faults: 4**

## Comparison

Algorithm	Page Faults	Best When
FIFO	4	Rarely optimal; simple but not smart
LRU	4	Access patterns have locality (pages used recently will be used again)

**Conclusion:** LRU generally performs better than FIFO in real systems because programs tend to reuse recently-accessed pages (temporal locality principle).

## Part C: Distributed Systems

### Question 9: Designing a Distributed File System

#### a) Two Critical Issues in Distributed OS Design

##### Issue 1: Consistency in File Sharing

**Problem:** Files are stored across multiple servers. When two users access the same file from different locations:

- User A modifies the file on Server 1
- User B reads the file from Server 2
- Should User B see User A's changes immediately?

##### Challenges:

- Network delays: Changes take time to propagate
- Caching: Each server caches files locally for speed
- Conflicts: What if both users modify the same file simultaneously?

##### Issue 2: Resource Management & Load Balancing

**Problem:** Multiple users and devices share limited server resources.

##### Challenges:

- One server might become overloaded (too many requests)
- Other servers might be underutilized

- Need automatic load balancing to distribute work
- Failure handling: If one server crashes, work must move elsewhere

## b) Architectural Approaches for Distributed File Management

Approach	How It Works	Benefits
<b>Client-Server with Replication</b>	Files replicated across multiple servers; clients read from nearest/fastest server	Improves performance and availability; if one server fails, copies exist elsewhere
<b>Distributed Lock Manager (DLM)</b>	Central lock service coordinates file access across all servers; prevents simultaneous conflicting modifications	Prevents data corruption when multiple clients access same file
<b>Caching with Coherence Protocol</b>	Servers cache files locally but maintain consistency using coherence protocol (similar to CPU caches)	Fast access through caching; consistency through protocol

**Recommended Architecture:** Combine all three:

1. **Replicate** files across multiple data centers
2. **Use DLM** to manage concurrent access
3. **Cache** aggressively with coherence protocol

This ensures **performance, consistency, and fault tolerance.**

## Question 10: Synchronous vs Asynchronous Checkpointing

**What is Checkpointing?** Periodically save system state so recovery is possible after failure.

### Synchronous Checkpointing

#### How It Works:

1. Coordinator tells all nodes: "Stop! Checkpoint now!"
2. All nodes freeze current execution
3. All nodes save their state to stable storage
4. All nodes synchronize timestamps
5. All nodes resume execution

#### Visual Example:

Time: 0s 1s 2s 3s 4s 5s

Node 1: [Running] [STOP] [Save] [Sync] [Resume] [Running]

Node 2: [Running] [STOP] [Save] [Sync] [Resume] [Running]

Node 3: [Running] [STOP] [Save] [Sync] [Resume] [Running]

←→ All synchronized →←

### **Strengths:**

- **Global Consistency:** All checkpoints aligned - safe recovery to any checkpoint
- **Easy Recovery:** All nodes at same state; simple rollback
- **Transparent:** Application doesn't need to manage checkpoints

### **Weaknesses:**

- **High Blocking Time:** All nodes stop (entire system pauses)
- **Scalability Issue:** With many nodes, coordination overhead grows
- **Performance Penalty:** Frequent synchronization slows everything

## **Asynchronous Checkpointing**

### **How It Works:**

1. Each node saves state independently, whenever it wants
2. No global synchronization
3. Nodes continue running without waiting for others

### **Visual Example:**

Time: 0s 1s 2s 3s 4s 5s

Node 1: [Running] [Save] [Running] [Save] [Running]

Node 2: [Running] [Running] [Save] [Running] [Save]

Node 3: [Save] [Running] [Running] [Save] [Running]

←— Each independent —→

### **Strengths:**

- **No Blocking:** Nodes never stop; continuous progress
- **High Performance:** Minimal overhead
- **Scalability:** Works well with many nodes
- **Better for IoT/Edge:** Devices can checkpoint independently

### **Weaknesses:**

- **Recovery Complexity:** Checkpoints not aligned; need to reconstruct consistent state
- **Cascading Rollbacks:** Recovering to one node's checkpoint might require rolling back others
- **Data Loss Risk:** If recovery isn't done carefully, some committed work might be lost

## Comparison Table

Aspect	Synchronous	Asynchronous
<b>Blocking Time</b>	High	None
<b>Consistency</b>	Easy to ensure	Complex
<b>Recovery</b>	Simple	Complex (may need cascade)
<b>Performance</b>	Slower	Faster
<b>Scalability</b>	Poor	Good
<b>Best For</b>	Small systems, critical applications	Large distributed systems, edge computing

### Recommendation:

- Use **Synchronous** for small distributed databases where consistency is critical
- Use **Asynchronous** for large-scale systems like cloud data centers

## Question 11: Smart Home IoT Scheduling

**Scenario:** Smart home with cameras, lights, locks, temperature sensors. Need to prioritize security over comfort.

### a) Process Scheduling Strategy

#### Priority Levels:

1. **Critical (Highest):** Security devices - Camera motion detection, door lock alerts
2. **High:** Safety - Fire/smoke detectors, temperature warnings
3. **Medium:** Regular - Light automation, routine tasks
4. **Low:** Non-critical - Statistics collection, logging

#### Recommended Algorithm: Priority Preemptive Scheduling

#### How It Works:

1. Assign priority number to each device's task (1=Critical, 4=Low)
2. Always run the highest priority ready task
3. If high-priority task arrives, pause lower-priority task
4. Return to lower-priority when high-priority completes

#### Justification:

- **Security first:** If camera detects break-in, it must execute immediately
- **Preemption:** Can interrupt light dimming to handle security event
- **Fairness:** Low-priority tasks still run when no urgent events occur

## b) Inter-Process Communication (IPC) Methods for IoT

### Best IPC Methods:

IPC Method	Use Case	Why Suitable
<b>MQTT (Publish-Subscribe)</b>	All devices send events to central hub	Lightweight, efficient; perfect for IoT; handles device failures gracefully
<b>Message Queues</b>	Pass security alerts between services	Reliable delivery with persistence; if consumer crashes, message waits
<b>Shared Memory</b>	Fast local communication between nearby modules	Ultra-fast for local sensors on same gateway; minimal latency
<b>REST API</b>	Remote device queries (e.g., check camera feed)	Stateless, scales well; easy for cloud integration

## Question 12: Linux System Calls Demo

**Objective:** Demonstrate OS system calls for file operations and threading.

Linux is selected as the operating system for this case study. The goal is to show how a simple Python program uses Linux system calls indirectly to create, read, inspect, and delete a file.

```
import os

FILENAME = "linux_demo.txt"

DATA = "Hello from a Linux file system call demo in Python!\n"

def main():

    print("== Python File Demo on Linux ==\n")

    # STEP 1: Create and write a file

    # This uses high-level Python I/O, which internally calls Linux sys_open and sys_write.

    print("STEP 1: Creating and writing file...")

    with open(FILENAME, "w", encoding="utf-8") as f:

        f.write(DATA)

    print(f" Wrote text to {FILENAME}\n")

    # STEP 2: Read the file back

    # Internally mapped to Linux sys_open and sys_read.

    print("STEP 2: Reading file...")
```

```

with open(FILENAME, "r", encoding="utf-8") as f:
    content = f.read()
    print(" File content:")
    print(" ", content)

    # STEP 3: Get file metadata using os.stat
    # This wraps the Linux stat() system call.

    print("STEP 3: File metadata (stat):")

    st = os.stat(FILENAME)
    print(f" Size: {st.st_size} bytes")
    print(f" Permissions (octal): {oct(st.st_mode & 0o777)}")
    print(f" Inode number: {st.st_ino}")

    print()

    # STEP 4: Delete the file using os.remove (unlink system call)

    print("STEP 4: Deleting file...")
    os.remove(FILENAME)

    print(" File deleted. Demo complete.")

if __name__ == "__main__":
    main()

```

- Linux provides system calls such as open, read, write, stat, and unlink to allow user programs to work with files. Python's open(), os.stat(), and os.remove() are high-level wrappers that eventually invoke these underlying system calls through the C library, which triggers a switch from user mode to kernel mode.
- This program demonstrates the file abstraction offered by the operating system. The Python code never manipulates disk blocks directly; instead, it treats the file as a simple stream of bytes, while the Linux kernel handles details like locating the inode, managing the file descriptor, caching data, and scheduling disk I/O.
- Security and protection are also illustrated. When open() or os.stat() are called, the kernel checks file permissions (read/write bits in the mode field) before allowing access, enforcing the access control policy defined by the Linux permission model