10/10

Assignment: Capstone

PART (A)

1) Modern systems rely an OS because they provide abstractions that hide hardware complexity & exp clean interfaces for programs

Key OS abstractions

- Process abstraction
  virtual CPU, scheduling, isolation
- Memory abstraction
  virtual memory, address space
- I/O abstraction
  device independence, buffering, drivers

Services

Process management
PCB creation, context switching, CPU scheduling

Memory management
paging, segmentation, allocation, protection

I/O management
unified driver model, interrupt handeling, DMA

why still needed today?
Even with advanced hardware, OS manages concurrency security, resource allocation & fault tolerance which hardware alone can't do.

2)

Structure	Feature	Pros	Cons
Monolithic	Entire OS in one large kernel	Fast, efficient	Hard to maintain, low reliability
Layered	OS split into layers each using lower layer	Modular, easier debugging	performance overhead
Microkernel	Minimal kernel, services run in user mode	High reliability, fault isolation	Slower due to more IPC

Best for distributed web applications
Microkernel - because it offers fault isolation, modularity, maintainability & high reliability essential for distributed & cloud based deployments

3) Threads vs Processes
Threads are lighter because
- share code/data, processes have seperate spaces
- thread control block (TCB) is smaller than PCB
- context switch between threads avoids MMU updates → faster
- threads consume fewer resources than full process creation

But threads are not always better:
require synchronization → risk of race conditions
shared memory → debugging harder
process isolation offers better security

Thread management is more efficient, but less safe than process mgmt

4) Memory allocation
First fit & Best fit
Processes : 12 MB, 18 MB, 6 MB
Blocks : 20MB, 10MB, 15MB

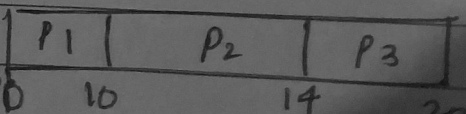
First fit
12MB → 20MB (remaining 8MB)
18 MB → (next block 10MB → skip, 15MB → skip).
can't be allocated
6MB → 10MB (4MB remaining)
Fragmentation 8MB + 4MB + unused 15MB
Best fit

12MB → 15MB (remaining 3MB)
18MB → 20MB (remaining 2MB)
6MB → 10MB (remaining 4MB)
Fragmentation 3MB + 2MB + 4MB (total smallu
First fit → more efficient.

PART (B)

5) FCFS, SJF, RR scheduling
Assume processes
P_1 BT = 10
P_2 BT = 4
P_3 BT = 6

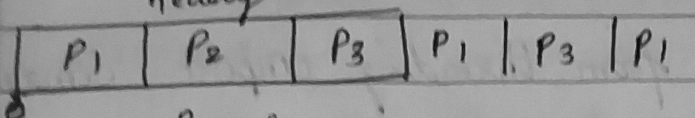
Gantt charts
FCFS
P_1 | P_2 | P_3



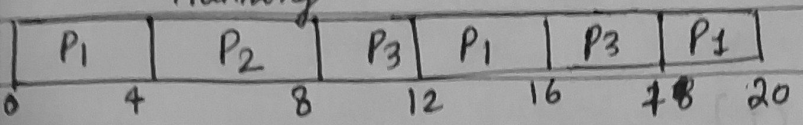
SJF

RR (q=4)

Ready



Running



Average Time

Algo	Avg Waiting	Avg Turnaround
FCFS	High	High
SJF	lowest	lowest
RR	balanced	moderate

Best overall
Round Robin → balances throughput + fairness in multiprogram

6) Banker's Algo
a) checks if resources requests keeps the system in a safe state
Allocation only if :-
Available = Need
Prevents circular wait → avoids deadlock.

b) Detection & Recovery approach
Use wait for graph
Detect cycles → indicates deadlock
Recovery methods

abort one/ few transactions
roll back to checkpoint
release held locks

7) Producer- consumer using Semaphores

python

```
import threading, time from threading import Sem

buffer = []
mutex = Semaphore(1)
empty = Semaphore(3)
full = Semaphore(0)

def producer():
    while True:
        item = "data"
        empty.acquire()
        mutex.acquire()
        buffer.append(item)
        mutex.release()
        full.release()
def consumer():
    while True:
        full.acquire()
        mutex.acquire()
        buffer.pop(0)
        mutex.release()
        empty.release()
```

Ensures

mutual exclusion → mutex

bounded buffer control → empty, full

no race conditions

8) Page replacement FIFO & LRU

Sequence : 2, 1, 4, 2, 3, 4, 3

FIFO

F_3			4	4	4	4	4
F_2		1	1	1	1	1	1
F_1	2	2	2	2	3	3	3
	*	*	*	H	*	H	H

Page fault = 4

LRU

F_3			4	4	4	4	4
F_2		1	1	1	3	3	3
F_1	2	2	2	2	2	2	2
	*	*	*	H	*	H	H

Page fault = 4

Frames differ after step 5 (FIFO replaced 2, LRU replaced 1) but total faults tie for this case

9) Distributed file systems

a) Two critical issues (concise)

1. Consistency vs Performance (Caching & Replication)
   - Cached / replicated copies improve read performance & fault tolerance but risk stale reads & write conflicts. Choosing strong vs eventual consistency affects latency & correctness.
2. Metadata scalability & availability
   - Centralized metadata (file namespace, mapping to chunks) is a single point of failure or bottleneck; distributing metadata increases complexity (consistency, fault-tol.) but is necessary for global scale

b) Architectural approaches (concise)

1. Separation of metadata & data (GFS-style)
- Small, replicated metadata service (masters) & many chunk/data servers. Clients contact metadata service for locations, then interact directly with chunk servers for data - reduces master load & improves throughput

2. Lease-based client caching + primary-backup writes
- Give clients short leases for cached copies; primary (leader) handles writes & invalidates/updates replicas to ensure consistency with bounded staleness.

3. Consistent hashing + sharding for scale
- Partition namespace across metadata shards to distribute load; use replication within shards for availability.

4. Tiered consistency
- choose file-level policies (strong for transactional files, eventual for logs/analytics) to balance latency & reliability.

10) Synchronous checkpointing & recovery

Mechanism (pseudocode + ASCII diagram)

```
Coordinator: initiate_checkpoint(k)
    broadcast("checkpoint_request", k)
    wait for ACK's from all processes
    if all ACKs received:
        commit global checkpoint k to stable storage
        broadcast("checkpoint_committed", k)
```

Process i on "checkpoint_request k":
flush outgoing messages to peers (or ensure deterministic logging)
write local state_i_k to stable storage
send ACK to coordinator

ASCII timeline
Coordinator → P_1: checkpoint_request
Coordinator → P_2: checkpoint_request
P_1: flush & write state1_k → ACK
P_2: flush & write state2_k → ACK
Coordinator: receive all ACKs → checkpoint committed

Recovery (concise)
On failure, restore each process from last committed global checkpoint k & replay logged messages if message logging is used

Strengths & Weakness
→ Synchronous
- Strength: simple, yields a consistent global state, easy recovery
- Weakness: High runtime overhead, stops progress (blocking) during checkpoint; poor scalability

→ Asynchronous
- Strength: Low runtime interference; scalable
- Weaknesses: Harder recovery (may require complex message logging / analysis); risk of domino effect unless message logging or coordinated techniques used.

11) Smart Home IoT (centralized OS)

a) Scheduling strategy (algorithm + justification)

Hybrid: Immediate Interrupt Handling + Preemptive Priority Scheduling (EDF for soft-real time / Priority-based with strict interrupt handling).

Concretely:

- Hardware/firmware interrupts for security sensors (camera motion) preempt CPU immediately & trigger short, high-priority ISR that records event metadata & signals a high-priority real-time task.
- Scheduler: Use preemptive priority scheduler where security-processing threads have highest priority with bounded execution time (or use EDF when deadlines are explicit). Lower-priority tasks (lightning) run when no higher-priority tasks pending.

Justification: Security events require minimal latency, bounded response, preemptive priority + ISR ensures immediate attention & bounded interferences. EDF can be used if tasks have hard deadlines & utilization is analyzable.

Design notes:

Keeps ISR's minimal (signal/enqueue) to avoid long interrupt handling in kernel.

Offload heavy processing to dedicated edge AI (GPU) if available to reduce scheduling pressure on main OS.

b) IPC methods suitable (brief + reason)

1. Message queues (local): decoupled, priority-aware useful for event notification (motion detected enqueue alert) Easy to throttle and drop

low priority messages if overloaded.

2. Shared memory + mutex / semaphore : for high - bandwidth local data (camera frames) where copying is expensive. Use ring buffers + sequence numbers to avoid copying & reduce latency.

3. MQTT / lightweight broker (network) : for high - bandwidth local data (camera frames) where copying for device - to - cloud or cross room messaging - lightweight support QoS levels & is standard in IoT.

4. Unix domain sockets / RPC (local) : for structured requests between system components with lower overhead than TCP.

Why these : Messages queues + shared memory combine low latency, controlled synchronization & decoupling - suitable for mixed - criticality IOT environments.

12) Case study : LINUX - Python demo of basic system calls
I ran a short Python demo that uses POSIX - like system call wrappers (os.open, os.write, os.read) & also created threads (which use OS threading primitive
Output from running the script is below & file was saved at /mnt/data/os_syscall_demo.txt

code run

```
import os, threading, time
fd = os.open("/mnt/data/os_syscall_demo.txt", os.O_
CREAT | os.O_WRONLY, 0o644)
os.write(fd, b"OS syscall demo : file created using os.
open / os.write \n line 2 \n")
```

```
os.close(fd)

fd = os.open("/mnt/data/os_syscall_demo.txt",
     RDONLY)
content = os.read(fd, 1024)
os.close(fd)
print(content.decode())

def worker(n):
    print(f"Worker {n} running (Thread id: {threa
      get_ident()})")
    time.sleep(0.1)
    print(f"Worker {n} done")

t1 = threading.Thread(target=worker, args=(1,))
t2 = threading.Thread(target=worker, args=(2,))
t1.start(); t2.start()
t1.join(); t2.join()
```

Observed output

```
OS syscall demo: file created using os.open/os.write
Line 2\n

Worker 1 running (threading id: 139265115637
Worker 2 running (threading id: 139265107277
Worker 1 done
Worker 2 done
Threads completed
```

Note: Python threading calls into OS threading pr
(pthreads on POSIX)