DELTA
Date / /
Pg No.

Assignment - 5 (Capstone Assignment)

Q1.

Ans. Modern computers still need operating system because the OS manages hardware and provides an easy interface for its users & programs.

→ It creates abstraction like processes for running programs, virtual memory for efficient storage use and files for I/O operation.

→ The OS handles processes (CPU scheduling, multi-tasking), memory management (allocation protection) and I/O management (allocation protection) and I/O management (Device communication).

→ It hides hardware complexity, ensures security, and enables resource sharing.

Q2 Compare Monolithic, layered and microkernel operating system structures. From the perspective of readability and maintainability, which would you select for a distributed web application? Justify

Ans. Monolithic OS has all system services in one large kernel – fast but hard to maintain.

↳ Layered OS divides the system into layers each built on the one below → easier to debug.

→ Microkernel OS keeps only core functions (like communication, scheduling) in the kernel, moving others to user space. It is slower but more reliable and secure.

→ For a distributed web applications, a microkernel OS is best because failure in one service don't crash the whole system, and updates are easier.

Q3.

Ans. Threads are lighter than processes because they share the same memory; so the OS doesn't need to create a new address space like

it does in a process.

- A Process Control Block (PCB) stores a lot of Heavy information, while a thread control block (TCB) stores much less.

- Because of this, thread context switching is faster than switching b/w processes. Processes need more resources and protection, making them slower to manage. So threads are usually more efficient, but processes give better isolation and safety.

Qy.

Ans:

Case study: LINUX OS → demonstrating determining a basic call (file creation) & read.

Python program calls OS system call through open(), write(), read().

```
# File creation and reading using system calls.
# Creating files (write system calls)
with open ("sample text", "w") as f:
    f.write ("Hello OS"! This a linux system call demo").
```

```
# Reading the file (Read system call).
with open ("sample.txt", "r") as f:
    Data = f.read().
    print ("File content:", data).
```

Output

- File output content: Hello OS! This is a Linux System called demo

Relevance to OS architecture

- When python uses open(), Read(), or write(), it asks the Linux kernel to perform a task.
- These actions are called System calls, which acts like a bridge b/w

the program and the OS.

- The OS ensures safe access to the file system, manages permissions and stores data correctly.
- This demonstrates how user-level programs depend on kernel services, which is a core concept in OS architecture.

Q4.

Ans ✓

Synchronous checkpointing

All nodes stop together, save their state at the same time then continue.

Like taking a group photo everyone freezes for a moment.

- Diagram

Node A: ---- work ---- [checkpoint] -- work --

Node B: ---- work --- [checkpoint] -- work --

Node C: ---- work --- [checkpoint] -- work --

↑ All checkpoints together

- Pseudocode

Coordinator → send checkpoint request to all nodes.

All nodes pause updates.

Each node saves state to stable storage.

All nodes confirm done.

System Resume.

- Recovery

Load last global checkpoint.

Restart all nodes from the same point in time.

Asynchronous checkpointing

Nodes take checkpoints whenever they want, without stopping others.

Like taking individual selfies instead of a group photo.

- Diagram

Time →

Node A: ---work---[P]---work---[P]---

Node B: ---work---[P]---[P]---

Node C: ---work---[P]---work---

(Check points occur at different times).

- Pseudocode.

Each node independently:

if timer expires:

save checkpoints.

- Recovery

May need message logs to rebuild a consistent state

Q6.

(a) (i) Transparency (Access + Location Transparency)

User should feel like all files are on one computer even though they are spread across many machines.

- The system must wide where the file is stored.
- Ensures easy file sharing and uniform access.

(ii) Fault tolerance & Reliability.

Distributed system must keep working even if one machine crashes.

- File must be replicated or logged.
- Recovery must ensure no data loss.

(b) (i) Client-Server Architecture (NFS-style)

- Client requests file, server stores files.
- cache on the client to reduce load.
- Stateless servers improve crash recovery.

(ii) Replication-based architecture (AFS / Google AFS-style).
- Make copies of files across multiple sites.
- Ensure fault tolerance.
- Read operation becomes faster (nearest copy served).

Benefits: High reliability and availability.

(iii) Distributed Naming & Metadata Servers.
- Stores file names seperately from file data.
- uses lookup tables (like HDF Namenode + Data Node model).

Benefits: Faster search, load balancing, easier management.

Q7.

Ans (a) Banker's algorithm checks before granting a lock whether the system will still be in safe state.
Like a banker giving loans only when he knows every customer can finish and return money.
So the OS only locks accounts if all transactions can still complete, preventing deadlocks.

(b) Detection approach:
- Build a wait-for graph (who is waiting for whose account)
- If the graph forms a cycle → deadlock detected.

Recovery approach:
- Abort or rollback one of the transactions holding the locks.
- Release its lock letting others continue
- Restart the aborted transaction later.

Q8.

Ans: Producer - Consumer using semaphores:

- Mutex (Binary semaphore) :- allow only one thread to access the buffer at a time.
- Empty semaphore → counts empty slots. (producers wait if buffer full)
- Full semaphore → counts filled slots. (consumers wait if Buffer empty).

Simple diagram

Producer → [empty] → [MUTEX] → BUFFER → [FULL] → Consumer.

PYTHON CODE (Semaphore Solution)

```
import threading, time, random
from threading import Semaphore

buffer = []
Buffer_size = 5

mutex = Semaphore(1)
empty = Semaphore(BUFFER_SIZE)

def producer():
    while true:
        item = random.randint(1,100)
        empty.acquire()
        mutex.acquire()
        buffer.append(item)
        print("produced:", item)
        mutex.release()
        full.release()

def consumer():
    while True:
        full.acquire()
```

```
mutex.acquire()
item = Buffer.pop(0)
print("consumed:", item)
mutex.release()
empty.release()

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

Q9.

Ans. (a) Scheduling strategy

- security devices (cameras, alarms) = highest priority
→ their interrupts preempt other tasks.

- normal tasks (lights, AC) = lower priority → 2 units only when no critical events.

Justification: Ensures instant response to security events while delaying non-critical work

(b) IPC methods: use message queues + shared memory + signals:

- Message queues: send alerts from devices → OS reliably (ordered + asynchronous).
- shared memory: Fast data sharing between processes (eg sensor-reading).
- signals/interrupts: notify OS immediately when security triggers event

Justification: Provides fast, reliable, event-driven communication in a smart home - IoT environment.

DELTA
Date / /
Pg No.

Q10.

First-fit (Scan from $B_1 \to B_2 \to B_3$ for each process):

- $P_1(12) \to B_2$ (20) → leftover 8.
- $P_2(18) \to B_2$ single block ≥ 18 (B1 has 8, B2 = 10, B3 = 15) → not allo

(ii) SJF (Non-preemptive)

Best fit (choose smallest block that fits):

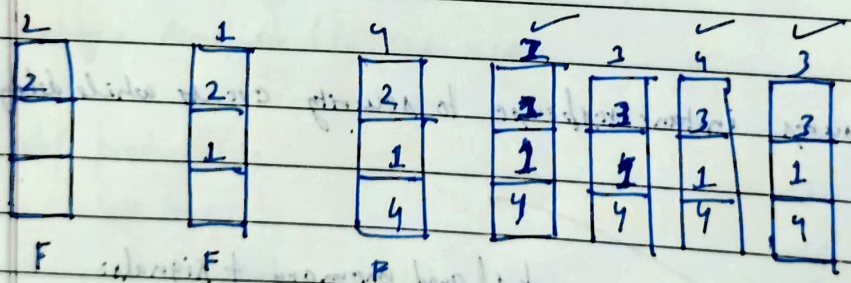
- $P_1(12) \to B_3(15)$ → leftover 3.
- $P_2(18) \to B_2(20)$ → leftover 2.
- $P_3(6) \to B_2(10)$ → leftover 4.

all processes allocated; Internal frag = 3+2+4 = 9 MB;
no external fragmentation

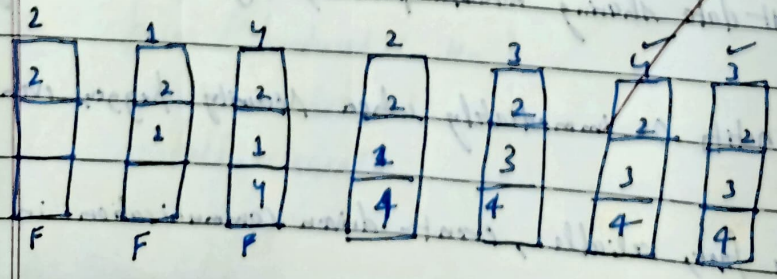
Q11. FIFO

Ans:

(a)



FIFO page fault = 4.



LRU page fault = 4

DELTA
Date / /
Pg No.

(b) (ii) SJF (non-preemptive)

Processes	WT	TAT
P_1	0	5
P_2	4	7
P_4	5	11
P_3	12	20

Avg WT = 5.25
Avg TAT = 10.75

(ii) Round Robin (q = 4)

Processes	WT	TAT
P_1	10 - 0 - 5 = 11	16
P_2	7 - 1 - 3 = 3	6
P_3	10	18
P_4	12	19

Avg WT = 9.25
Avg TAT = 14.75

(c) ~~Discuss which algorithm is best balances~~ FCFS
- Simple but suffers from "convoy effect"
- Long jobs block ones.
- Worst responsiveness.

SJF (non-preemptive)
- Minimize average waiting time
- Best turnaround time amongst the three.
- But requires knowing burst-time.

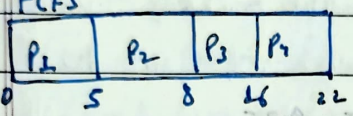
Round Robin
- Best fairness and responsiveness.
- Higher context-switch overhead.
- Higher waiting-time but better throughput in multiprogrammed system.

Q12:

(a) Ans:

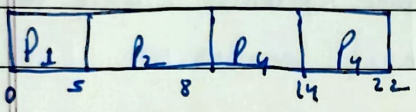
Processes	Burst Time (ms)	Arrival time (ms)
P_1	5	0
P_2	3	1
P_3	8	2
P_4	6	3

(i) FCFS



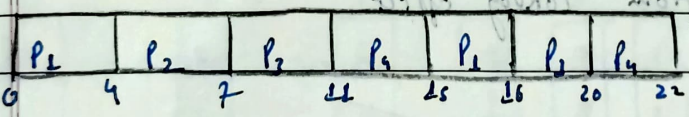
order: $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$

(ii) SJF (non-preemptive)



- $t=0 \rightarrow P_1$
- $t=5 \rightarrow P_2, P_3, P_4$ (Choose P_2)
- After $P_2 \rightarrow$ choose P_4.
- Finally P_3.

(iii) Round Robin: (q = 4 ms)



P_1: Runs 4 → Remaining 4
P_2: Runs 3 → Done.
P_3: Runs 4 → Remaining 4
P_4: Runs 4 → Remaining 2
P_1: Runs 1 → Done
P_3: Runs 4 → Done
P_4: Runs 2 → Done.

(b)

(i) FCFS

Process	WT	TAT
P_1	0	5
P_2	4	7
P_3	6	14
P_4	13	19

Avg WT = (0+4+6+13)/4 = 5.75

Avg TAT = 11.25