REG.NO.:

SLOT: G1+TG1

Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING
CONTINUOUS ASSESSMENT TEST - II
WINTER SEMESTER 2025-2026

Programme Name & Branch : B. Tech / CSE
Course Code and Course Name : BACSE106 - Operating Systems
Date of Examination : 23.3.2026
Exam Duration : 90 minutes **Maximum Marks: 50**

CO2: Apply process scheduling algorithms and analyze thread models and deadlock-handling methods.
CO3: Implement synchronization mechanisms and solve concurrency problems in multi-process systems.
CO4: Evaluate memory management and file system strategies used in operating systems.

Q. No	Questions
1	**Step 1: Given** Total GPU instances **Resource** / **Total** P / 4 Q / 7 R / 10 S / 9 Departments: Analytics, ML_Training, Web_Hosting, Database, Security **Step 2: Allocation Matrix** **Department** / **P** / **Q** / **R** / **S** Analytics / 1 / 2 / 1 / 0 ML_Training / 1 / 1 / 2 / 3 Web_Hosting / 0 / 0 / 1 / 0 Database / 2 / 1 / 1 / 0 Security / 0 / 1 / 1 / 0
	Step 3: Maximum Matrix Department / P / Q / R / S Analytics / 1 / 4 / 3 / 5 ML_Training / 2 / 4 / 5 / 4

Web_Hosting	1	2	2	0
Database	6	7	6	3
Security	0	3	3	3

Step 4: Calculate Available Vector

Available = Total − Allocated

Resource	Available
P	4 − 4 = **0**
Q	7 − 5 = **2**
R	10 − 6 = **4**
S	9 − 3 = **6**

Initial **Work = (0,2,4,6)**

a) Calculate the Need matrix for each department. **(2 Marks)**

$$Need = Max - Allocation$$

Department	P	Q	R	S
Analytics	0	2	2	5
ML_Training	1	3	3	1
Web_Hosting	1	2	1	0
Database	4	6	5	3
Security	0	2	2	3

b) Use Banker's Algorithm to determine if the current state is Safe or Unsafe. Show all steps with Available, Allocation, Need and Work vectors. **(4 Marks)**

Analytics
Need = (0,2,2,5) ≤ Work (0,2,4,6)
Execute Analytics
Work = Work + Allocation

$$(0,2,4,6) + (1,2,1,0) = (1,4,5,6)$$

ML_Training
Need = (1,3,3,1) ≤ Work (1,4,5,6)

Execute

Work = $(1,4,5,6) + (1,1,2,3) = (2,5,7,9)$

Web_Hosting
Need = (1,2,1,0) ≤ Work (2,5,7,9)
Execute
Work = $(2,5,7,9) + (0,0,1,0) = (2,5,8,9)$

Safe sequence:
Analytics → ML_Training → Web_Hosting

Security

Need = (0,2,2,3) ≤ Work (2,5,8,9)

Execute

Work =$(2,5,8,9) + (0,1,1,0) = (2,6,9,9)$
Safe sequence:
Analytics → ML_Training → Web_Hosting → Security

Database

Need = (4,6,5,3)

Work = (2,6,9,9)

Condition fails because $4 > 2$
Database cannot execute. So System State = UNSAFE

c) ML_Training submits a resource request: (1, 2, 1, 0) **(4 Marks)**

Request = **(1,2,1,0)**

Check Request ≤ Need

ML_Training Need = (1,3,3,1)

Request = (1,2,1,0)

Condition satisfied.

Check Request ≤ Available

Available = (0,2,4,6)

Request = (1,2,1,0)

Fails because

$$P_{request} = 1 > P_{available} = 0$$

Conclusion

The request **cannot be granted immediately** because sufficient **P resources are not available**.

Deadlock Conditions

Deadlock occurs when the following four necessary conditions hold simultaneously:

- Mutual Exclusion

GPU instances cannot be shared between departments.

- Hold and Wait

Departments hold allocated GPUs while requesting more.

- No Preemption

GPUs cannot be forcibly taken away.

- Circular Wait

Departments form a circular chain waiting for each other's resources.

2 **Given**

Initial **counter value = 50**

Process	**Increment**
P1	+5
P2	+3
P3	+4
P4	+2

Execution order: **P2 → P1 → P4 → P3**

Synchronization mechanism: **Semaphore / hardware atomic instruction**

(a) Hardware atomic instruction preventing race conditions (with pseudocode) – (3 Marks)

A hardware atomic instruction (such as Test-and-Set) performs the check and update operation in one indivisible step.

This ensures that only one process can enter the critical section at a time, preventing multiple processes from modifying the shared counter simultaneously.

Test-and-Set Pseudocode

```
boolean TestAndSet(boolean *lock){
   boolean old = *lock;
```

```
  *lock = true;
  return old;
}
```

Process Entry Using Atomic Instruction

```
while(TestAndSet(lock))
  ;      // busy waiting
// Critical Section
counter = counter + increment_value
lock = false
```

- Initially **lock = false**
- When a process executes **TestAndSet**, it sets the lock to **true**
- Other processes must wait until the lock becomes **false**
- This guarantees **mutual exclusion**, preventing **race conditions**

(b) Semaphore wait() and signal() operations – (3 Marks)

wait() operation (P operation)

```
wait(S){
  while(S <= 0)
    ;    // busy waiting
  S = S - 1;
}
```

signal() operation (V operation)

```
signal(S){
  S = S + 1;
}
```

Using Semaphore in Critical Section

Initialization

```
semaphore S = 1
counter = 50
Process Pi
```

wait(S) // Entry Section

counter = counter + increment_value // Critical Section

signal(S) // Exit Section

- **wait(S)** decreases semaphore and allows entry if resource is free
- **signal(S)** releases the semaphore so other processes can enter

(c) Final Counter Value – (4 Marks)

Initial counter = 50

Execution order: P2 → P1 → P4 → P3

Step	**Process**	**Operation**	**Counter**
Initial	—	—	50
1	P2	50 + 3	**53**
2	P1	53 + 5	**58**
3	P4	58 + 2	**60**
4	P3	60 + 4	**64**

Final Counter Value

Final Counter = 64

Limitation of this synchronization mechanism

One limitation is busy waiting (spin waiting).

- Processes repeatedly check the lock/semaphore while waiting.
- This wastes CPU cycles.
- In multiprocessor systems with many processes, it can reduce system performance and CPU efficiency.

3 This problem is a **modified Producer–Consumer synchronization problem**.

- **Chefs → Producers**
- **Delivery agents → Consumers**
- **Pickup counter → Shared buffer (size = 8)**

Conditions:

- Delivery agents can collect **only when at least 2 meals exist**.
- After **4 deliveries**, chefs must **pause preparation**.
- Chefs resume only when **3 delivery requests** arrive.

a) Define the required synchronization variables and state how the solution maintains **mutual exclusion** and buffer constraints. **(4 Marks)**

Shared Variables

mutex = 1 // Binary semaphore for mutual exclusion

empty = 8 // Number of empty slots in buffer

full = 0 // Number of meals available

twoMeals = 0 // Ensures at least 2 meals before delivery starts

pauseChef = 1 // Controls chef pause after 4 deliveries

requestChef = 0 // Delivery requests to restart chefs

counter = 0 // Current meals in buffer

deliverCount = 0 // Number of deliveries completed

requestCount = 0 // Delivery requests received

Mutual Exclusion

- mutex ensures only one process accesses the shared buffer at a time.
- Prevents race conditions when updating counter, deliverCount, etc.

Buffer Constraint

- empty ensures chefs do not place meals when buffer is full.
- full ensures delivery agents collect meals only when meals exist.

Special Conditions

- twoMeals allows delivery only when **counter ≥ 2**.
- pauseChef stops chefs after **4 deliveries**.
- requestChef resumes chefs after **3 delivery requests**.

b) Write the pseudocode for the **chef** process. **(3 Marks)**

Chef()

```
while(TRUE)
{
wait(pauseChef) // Pause if 4 deliveries completed
wait(empty) // Check buffer space
wait(mutex)
place_meal()
counter++
if(counter >= 2)
```

signal(twoMeals)
signal(mutex)
signal(full)
}

- Chef waits if buffer is full.
- Chef resumes only if pause condition is lifted.
- When 2 meals exist, delivery agents are allowed to collect.

c) Write the pseudocode for the **delivery agent** process. **(3 Marks)**

DeliveryAgent()

```
while(TRUE)
{
wait(twoMeals) // Ensure at least 2 meals available
wait(full)
wait(mutex)
collect_meal()
counter--
deliverCount++
if(deliverCount == 4)
{
pauseChef = 0
requestCount = 0
}
requestCount++
if(requestCount == 3)
{
deliverCount = 0
signal(pauseChef)
}
signal(mutex)
signal(empty)
}
```

- Delivery waits until two meals are available.

- After 4 deliveries, chefs pause.
- After 3 requests, chefs resume meal preparation.

4 Free memory blocks: 150, 400, 300, 400, 500, 350, 250 KB

Processes: P1=198, P2=313, P3=221, P4=516, P5=75, P6=255 KB

1) First Fit Allocation

(Allocate first block that is large enough)

PID	Process Size	Allocated Block	Remaining
P1	198	400	202
P2	313	400	87
P3	221	300	79
P4	516	Not allocated	—
P5	75	150	75
P6	255	500	245

Remaining Free Blocks

75, 202, 79, 87, 245, 350, 250

External Fragmentation: Yes (many small blocks)

Unallocated Process: P4

(Allocate smallest block that can satisfy the request)

Allocation

2) Best Fit Allocation

PID	Process Size	Allocated Block	Remaining
P1	198	250	52
P2	313	350	37
P3	221	300	79
P4	516	Not allocated	—
P5	75	150	75
P6	255	400	145

Remaining Free Blocks

52, 37, 79, 75, 145, 400, 500

External Fragmentation: Yes (many small unusable blocks)

Unallocated Process: P4

3) Worst Fit Allocation

(Allocate largest available block)

PID	Process Size	Allocated Block	Remaining
P1	198	500	302
P2	313	400	87
P3	221	400	179
P4	516	Not allocated	—
P5	75	350	275
P6	255	302	47

Remaining Free Blocks

150, 87, 179, 275, 47, 300, 250

External Fragmentation: Yes

Unallocated Process: P4

4) Comparison of Algorithms

First Fit: Fast allocation, Moderate fragmentation
Best Fit: Uses smallest suitable block, Creates many small fragments
Worst Fit: Leaves larger free blocks, Wastes large memory blocks

5 **a)**

i) Number of bits for page offset

Page size = **8 KB**

$$8 \text{ KB} = 8192 \text{ bytes} = 2^{13}$$

Therefore,

$$\text{Page offset bits} = 13$$

Answer: 13 bits

ii) Number of entries in one page table

Virtual address space = **36 bits**

Page offset = **13 bits**

$$Page\ number\ bits = 36 - 13 = 23$$

Number of pages (and page table entries):

$$2^{23} = 8{,}388{,}608$$

b) Physical Address for (Data, 350)

- Base (Data) = 8000
- Offset = 350
- Limit = 600

Check validity:

$350 < 600$

So the address is **valid**.

Physical address:

$8000 + 350 = 8350$

Physical Address = 8350

Validity of Logical Address (Stack, 450)

- **Base (Stack) = 12000**
- **Offset = 450**
- **Limit = 400**

Check condition:

$450 > 400$

Since the offset **exceeds the limit**, the address **exceeds the segment size**.

Result: Invalid logical address (Segmentation fault).

c) Given

- TLB search time = 15 ns
- Memory access time = 120 ns
- Effective Access Time (EAT) = 165 ns
- Two-level page table

Step 1: Time for TLB Hit

$$\boldsymbol{TLB + 1\ memory\ access}$$

$$\boldsymbol{15 + 120 = 135\ ns}$$

Step 2: Time for TLB Miss

Two-level page table requires 3 memory accesses.

$$15 + (3 \times 120) = 15 + 360 = 375\ ns$$

Step 3: Effective Access Time Formula

$$EAT = h(hit\ time) + (1 - h)(miss\ time)$$

$$165 = h(135) + (1 - h)(375)$$

Step 4: Solve

$$165 = 135h + 375 - 375h$$

$$165 = 375 - 240h$$

$$240h = 210$$

$$h = 0.875$$

Step 5: Convert to Percentage

TLB Hit Ratio

$$0.875 \times 100 = 87.5\%$$

TLB Miss Ratio

$$100 - 87.5 = 12.5\%$$

✅ **TLB Hit Ratio = 87.5%**

❌ **TLB Miss Ratio = 12.5%**

****

,