NAME:_____ STD.:_____ DIV.:_____

DATE :
PAGE :

OPERATING SYSTEM ASSIGNMENT - 3

RUCCHIKA KAPOOR
2301010240

PART - A

Q1 Explain race conditions with a real-world example outside computing, and how mutual exclusion solves it.

Real condition example (Real World):
Two people trying to withdraw money from the same bank account at the same time using 2 diff. ATMs.

→ both check the current balance = ₹5000
→ both attempt to withdraw = ₹4000
→ final balance becomes incorrect (Rs 1000 / negative)

- if the bank locks the account during a transaction, only one withdrawal happens at a time. Thus, the second person must wait, preventing inconsistency

NAME:_____ STD.:_____ DIV.:_____

DATE :
PAGE :

Q2 Compare Peterson's solution and semaphores in terms of implementation complexity and hardware dependency.

Feature	Peterson's Solution	Semaphores
Implementation complexity	simple concept but works only for 2 processes, hard to scale	Easy to implement any number of processes.
Hardware dependency	Requires strict assumptions: atomic reads/writes, memory ordering	Uses hardware atomic instructions (test-and-set, swap); more reliable
Practical Use	Theoretical, rarely used in real OS.	Widely used in real OS synchronization.

Q3 One advantage of using monitors over semaphores in a multi-core ~~safe~~ system.

Monitors combine mutual exclusion + condition variables, ensuring that only one thread can access the monitor at a time, reducing programmer errors.

In multi-core systems, they prevent busy waiting, unlike semaphores that can ~~accidentally~~ accidentally cause deadlocks or Race conditions due to incorrect usage (P & V mismatch)

Q4 How starvation occurs in Reader-write Problem and one method to prevent it.

If readers keep coming continuously, writer may wait forever, never getting access → write starvation.

NAME:_____ STD.:_____ DIV.:_____

DATE :
PAGE :

methods :
use a writer requests access, block new ~~readers~~ readers until the writer finish.

Q5 Drawback of eliminating "Hold and Wait" in deadlock. Prevention.

To eliminate "Hold and Wait", a process must request all required resources at once.

Drawback :
Process may request many resources much before they actually need them, causing low resource utilization and long waiting times. This reduces performance and causes resource wastage

PART-B

Q6 Banker's Algorithm

Given Resources

→ total : A = 10, B = 5, C = 7

→ allocation & max : ... (from PDF)

process	allocation	max
P_0	010	753
P_1	200	322
P_2	302	902
P_3	211	422
P_4	002	533

NAME:_____ STD.:_____ DIV.:_____

DATE :
PAGE :

a) Need matrix = max allocation

process	need (A, B, C)
P_0	(7-0, 5-1, 3-0) = 7, 4, 3
P_1	(3-2, 2-0, 2-0) = 1, 2, 2
P_2	(9-3, 0-0, 2-2) = 6, 0, 0
P_3	(4-2, 2-1, 2-1) = 2, 1, 1
P_4	(5-0, 3-0, 3-2) = 5, 3, 1

Available Resources
total allocation :
A = 0 + 2 + 3 + 2 + 0 = 7
B = 1 + 0 + 0 + 1 + 0 = 2
C = 0 + 0 + 2 + 1 + 2 = 5

Available Total - Allocated
= (10-7, 5-2, 7-5)
= 3, 3, 2

b) Safe State check:
Start Available = (3, 3, 2)

→ P_1 need = (1, 2, 2)
New available = (3+2, 3+0, 2+0) = 5, 3, 2

→ P_3 need = (2, 1, 1)
New available = (5+2, 3+1, 2+1) = 7, 4, 3

→ P_0 need = (7, 4, 3)
New available = (7+0, 4+1, 3+0) = 7, 5, 3

→ P_2 need = (6,0,0)
New available = (7+3, 5+0, 3+2) = 10, 5, 5

→ P_4 need = (5, 3, 1)
New available = (10, 5, 7)

Sequence exists: $P_1 \rightarrow P_3 \rightarrow P_0 \rightarrow P_2 \rightarrow P_4$

∴ System is safe.

c) P_1 requests (1, 0, 2). Can we grant?

Current available = (3, 3, 2)
Request = (1, 0, 2)
Check need:
P_1 need = (1, 2, 2) → request ≤ need.
Request ≤ Available
(1, 0, 2) ≤ (3, 3, 2)

Grant temporarily and check safety:

Allocation P_1 = (3, 0, 2)
Available becomes = (2, 3, 0)
After recomputing safe sequence → still safe.

NAME:_____ STD.:_____ DIV.:_____

DATE :
PAGE :

Q7 Dining Philosophers (Semaphore Simulation)

Scenario showing deadlock
Using binary semaphore chopstick [5]:

```
wait ( chopstick [i]);
wait ( chopstick [(i+1)%5]);
eat ();
signal ( chopstick [i]);
signal ( chopstick [(i+1)%5]);
```

If all 5 philosophers pick their left chopstick, execution stops:

→ P_0 gets C_0
→ P_1 gets C_1
→ P_2 gets C_2
→ P_3 gets C_3
→ P_4 gets C_4

Now all wait forever for the right chopstick → deadlock.

AVOID DEADLOCK:
let the last philosopher pick the right chopstick first

```
if (i == 4) {
    wait ( chopstick [(i+1)%5]);
    wait ( chopstick [i]);
} else {
    wait ( chopstick [i]);
    wait ( chopstick [(i+1)%5]);
}
```

no deadlock

NAME:_____ STD.:_____ DIV.:_____

DATE :
PAGE :

Q8 I/O System Analysis

Given:
→ Interrupt handling time = 5 ~~to~~ ms
→ device transfer rate = 500 KB/s
→ Block per interrupt = ~~1000~~ 100 bytes

a) CPU time per second
Number of interrupts per second =
500 KB/s = 500 X 1024 bytes = 512000 bytes

Interrupts = 512000/100 = 5120 interrupts/s
CPU time = 5120 X 5 ms
= 25600 ms = 25.6 ms / second

b) Improvements:
Use DMA (Direct Memory Access) to transfer data in bulk, generating far fewer interrupts, reducing CPU overhead drastically.

Q9 Case Study: Air traffic control system
a) Critical Sections + IPC mechanism:
→ shared radar data structure
→ flight path calculations tables
→ shared communication buffers with pilots
→ shared airspace map

NAME:_____ STD.:_____ DIV.:_____

DATE :
PAGE :

Required IPC Mechanism:
Use real time message queues / priority - based semaphores

b) Deadlock between radar acquisition and flight path Calculation - detection and recovery.
In real time system, preemption is required
→ Abort all the non-critical process (flight path calculation)
→ Roll back its state
→ Allow radar acquisition to proceed
→ Restart flight calculation using saved checkpoint

This ensures minimal disruption to safety-critical operations.

Satisfied
21/11/25