Assignment : 3
Operating System

Rohini
23010102266

Part A

1. Race condition (real world example):
A race condition happens when two people try to use or change the same resource at the same time, causing an unpredictable outcome.
Example. Two people updating the same paper sign-up sheet at once. Person A writes their name, and at the same time Person B also writes theirs. Because they both read the old version first, one person's entry may overwrite or conflict with the other.

How mutual exclusion fixes it (in OS terms):
Mutual exclusion ensures that only one process access a critical section at a time. In the example, using a rule that only one person can hold the sign-up sheet at a time prevents conflicts. In an OS, this is done with locks or mutexes, which stop multiple processes from modifying shared data simultaneously, eliminating race conditions.

2. Peterson's Solution:
- Implementation complexity: Simple but limited (only for two processes)
- Hardware dependency: No special hardware—pure software using shared variables.

- Semaphores:
Implementation complexity: More complex to implement inside the OS.
Hardware dependency: Requires atomic hardware instructions (eg. test and set) for safe operation.

3. Monitors automatically handle mutual exclusion so in multicore system they prevent multiple threads from entering critical sections at the same time without relying on the programmer to manage locks manually, reducing errors & race conditions.

4. In the reader - writer, starvation can occur when one group is repeatedly favored - eg., if readers keep arriving, they may block writers indefinitely, preventing a writer from ever getting access.

One method to prevent it:
Use fair scheduling (FIFO queue): Serve readers and writers in the order they arrive, ensuring that once a writer is waiting, no new readers are allowed to enter before the writer gets its turn.

5. Eliminating "hold and wait" requires processes to request all resources at once, which is impractical because it leads to low resource utilization - processes may hold resources they don't need yet, blocking others & reducing

system efficiency.

6. Banker's Algorithm

→ Given : Total (10, 5, 7)

Process	Allocation	Max (A, B, C)
P_0	0, 1, 0	7, 5, 3
P_1	2, 0, 0	3, 2, 2
P_2	3, 0, 2	9, 0, 2
P_3	2, 1, 1	4, 2, 2
P_4	0, 0, 2	5, 3, 3

a) Need : Max - Allocation

Process	Need (A, B, C)
P_0	7, 4, 3
P_1	1, 2, 2
P_2	6, 0, 0
P_3	2, 1, 1
P_4	5, 3, 1

Available : Total - Allocation Sum

Allocation Sum = (7, 2, 1) → Available = (3, 3

b) Safe State Check :

Work : (3, 3, 1)

- P_1 can run (Need ≤ Work) → New work = (5, 3,
- P_3 can run → Work = (7, 4, 2)
- P_4 can run → work = (7, 5, 4)
- P_0 can run → work = (7, 5, 4)
- P_2 can run → work = (10, 5, 7)

Safe sequence : $P_1 \rightarrow P_3 \rightarrow P_4 \rightarrow P_0 \rightarrow P$

System is safe

c) If P requests (1, 2)
New Need = (0, 2, 0) Request ≤ Available (1, 3, 2)
≤ (2, 3, 1) → yes · 3 > 1
Cannot be granted immediately

7. Dining Philosophers:
→ Deadlock scenario: Each philosopher picks up their left chopstick first, all hold one chopstick & none can pick the right one → deadlock

Solution: Use semaphore array chopstick [5] = 1 and mutex for limit
wait (mutex);
wait (chopstick [i]);
wait (chopstick [(i+1) % 5]);
signal (chopstick [i]);
signal (chopstick [(i+1) % 5]);
signal (mutex);

8) I/O system Analysis:
Given:
Interrupt time = 5
Transfer Rate = 500 KB/s = 500 × 1024 = 512000 bytes
Block = 100 bytes.

a) Interrupts per second = 512000 / 100 = 5120
CPU Time = 5120 × 5ms = 25.6 ms per second

b) Improvement: use Direct Memory Access (DMA) to transfer data directly without frequent CPU interrupts

Student 21/11/24