DELTA
Date / /
Pg No.

Assignment-4

Q1: Explain race condition with real world example out of computing, and show how mutual exclusion adresses it.

Ans: A race condition occurs when the outcome of the system depends on the specific order or timing of execution of multiple thread/processes accessing a shared resource.

Example: Assuming a situation, where only 1 ticket is left on a booking site. Alice and Bob both book the ticket and the transaction succeeds. The system thinks that there is only -1 tickets left. This results in an error. Final state depends upon who finishes updating the inventory last.

- Mutual Exclusion Sol^n: Mutual exclusion adresses this by ensuring that only one process/thread can access the shared resource the ticket inventory at any given time. For example putting a digital lock.

1. Alice locks the system.
2. Alice books the ticket.
3. The ticket count is updated to 0.
4. Alice unlocks the ticket system.
5. Bob attempts to lock the system when the count is 0 and he is now blocked from purchasing this ticket.

Q2.

Ans:

Feature	Peterson's Sol^n	Semaphores
Implementation complexity	Low. Uses simple load and store instructions Requires two simple shared variables	High. Requires OS support operations like (wait() and signal() and a waiting queue to block processes

Hardware Dependency	High. Relies on a specific CPU architecture's atomic instruction and memory model to function	Low/Moderate: The primitive implementation relies on low-level atomic hardware instructions (like Test-and-set), but high level semaphore itself is an OS abstraction

Q3.

Ans. The producer-consumer problem can use either semaphores or monitors.

- Advantage of Monitor in a multi-core system: Monitor provides more easier and robust synchronization because they encapsulate the shared data and synchronization primitives (like condition variables and mutual exclusion locks into a single language construct. This inherent structure automatically ensures mutual execution on the shared data (the buffer), making it less prone to programming errors (like forgetting to call wait() and signal() correctly) compared to managing separate semaphores manually. This simplicity and safety leads to better maintainability in complex multi-core environments.

Q4.

Ans. Reader-Writer Starvation and Prevention

- How starvation occurs: In the reader-writer priority in the reader-writer solution a constant stream of new readers can continously acquire the lock, while the lock is held by other readers. A writer's process waiting to share the access will be indefinitely blocked because there is always an active reader arriving and gaining access before the writer gets its turn.
- Prevention method: Using a wait/Turnstile Queue: Use a writer-priority or fairness solution. A common fairness method is to implement a turnstile/queue for both readers and writers. When a process reader/writer wants to enter a critical solution, it must first check a second semaphore lock. The lock only releases when the current holder finishes. By forcing all (incoming requests (all readers & writers) through a single gate. It ensures that once a

reader writer reaches the front of the line, new readers are blocked until the writer is finished preventing indefinite reader preference.

Qs. Drawbacks of eliminating "Hold and wait"

Ans: "Hold and wait" condition for ~~deadl~~ deadlock means a process hold at least resources currently already holded by other processes. This can be eliminated by:

1. Requesting all resources at the start (before execution begins).
2. Allowing a process to only hold a resource when it is already holding none.

Practical drawback: Poor resource utilization and potential starvation.

↳ Poor resource utilization and potential: If a process needs (a printer) resource only at the ~~beg~~ end of its execution, requesting it at the start means the printer is held unnecessarily, for the entire execution time, wasting the resource and blocking other processes that could have used it immediately.

↳ starvation: Requesting all resources at once increases the chance that one resource will be unavailable, preventing the process to ever start, leading to starvation.

Part: B Application / Numerical Based

Qs.

~~A system has the following resources~~

DELTA
Date / /
Pg No.

Ans: using banker's algorithm:

Total resource : A = 10, B = 5, C = 7

All(P0) + ---- + Allo(P4) = (0+2+3+2+0, 1+0+0+1, 0 to 1+2) = 7,2,5

Available resources = (10,5,7) - (7,2,5) = 3,3,2

(b)

Process	Max (A, B, C)	Allocation (A, B, C)	Need (A, B, C)
P0	7,5,3	0,1,0	7,4,3
P1	3,2,2	2,0,0	1,2,2
P2	9,0,2	3,0,2	6,0,0
P3	4,2,2	2,1,1	2,1,1
P4	5,3,3	0,0,2	5,3,1

Step	Current Work	Process found need	New work = Allocation	Sequence
1	3,3,2	P1 (1,2,2)	(3,3,2) + (2,0,0)	P1
2	5,3,2	P3 (2,1,1)	~~5,3,2~~ → 7,4,3	P1, P3
3	7,4,3	P4 (5,3,1)	7,4,5	P1, P3, P4
4	7,4,5	P2 (6,0,0)	10,4,7	P1, P3, P4, P2
5	10,4,7	P0 (7,4,3)	10,5,7	P1, P3, P4, P2, P0

(c) We will use the resources-request algorithm

(i) Is request ≤ Need? P1 request (1,0,2) ≤ P1 Need (1,2,2)

(ii) Is request ≤ Available: Request (1,0,2) = Available (3,3,2)

(iii) Hypothetical allocation

Available = (3,3,2) - (1,0,2) = (2,3,0)

Allocation = (2,0,0) + (1,0,2) = (3,0,2)

Need = (1,2,2) - (1,0,2) = 0,2,0

DELTA
Date / /
Pg No.

Ans 7. The dining philosopher problem (DPP) is a classical synchronization problem that model how multiple processes complete for limited, shared.

Deadlock: A circular wait condition will occured ($P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4$) of the necessary condition for deadlock

Steps:

P_0 enters, pickup left fork
P_1 enters, pickup left fork
P_2 enters, pickup left fork
P_3 enters, pickup left fork
P_4 enters, pickup left fork.

Ans 8. $T_{int} = 5\ \mu s = 5 \times 10^{-6}$
$R_{data} = 500000$ bytes.

$N_{int} = \frac{500000}{100} = 5000$ interrupts

$T_{cpu} = N_{it} \times T_{int}$
$= 5000 \times 5\ \mu s = 25{,}000$