OPERATING SYSTEM

Unit IV — Concurrency, Locks and Synchronization

Topics Covered in this Unit

Concurrency & Shared Data: Race Conditions, Critical Sections, Mutual Exclusion & The Wish for Atomicity
Thread API & Programming: Need for Threads, Thread Creation & Completion (pthread_create, pthread_join)
Locks & Mutual Exclusion: Evaluating Locks (Mutual Exclusion, Fairness, Performance), Controlling Interrupts
Hardware Atomic Primitives: Test-And-Set (TAS), Compare-And-Swap (CAS), Load-Linked / Store-Conditional
Spinlocks vs. Sleeping Locks: Yielding, Queue-Based Solaris park() & Linux futex
Semaphores (Edsger Dijkstra): Definition, Value Invariant, sem_wait() / P() and sem_post() / V()
Binary Semaphores (Mutexes) & Ordering Patterns (Thread Join / Signaling with Semaphores)
Classic Synchronization Problems: Producer-Consumer (Bounded Buffer), Reader-Writer Locks & Dining Philosophers
Common Concurrency Bugs & Deadlocks: 4 Coffman Deadlock Conditions & Prevention / Avoidance Strategies
High-Yield University Exam Question Bank with Detailed Model Answers & Unit Summary

Compiled in a structured, easy-to-understand, textbook style format
Specially curated for B.Tech / B.E. / BCA / MCA / B.Sc Computer Science & Information Technology Students

COURSE SYLLABUS & MAPPING

[UNIT IV SYLLABUS — PRESCRIBED UNIVERSITY CURRICULUM]

This section contains the official syllabus for Unit IV. Verify with your university curriculum mapping.

Unit IV Syllabus Breakdown:

IV Concurrency: Concurrency, persistence, Shared data, uncontrolled scheduling, the wish for atomicity, thread api : need of threads, thread creation, thread completion Locks: the basic idea, pthread locks, building a lock, evaluating locks, controlling interrupts, failed attempt, just using. Semaphores: definition, binary semaphores (locks), semaphores for ordering, the producer/consumer (bounded buffer) problem, reader-writer locks, dining philosophers' problem, how to implement semaphores, common concurrency problems.

• Course Code: _____ • Semester / Year: _____

TABLE OF CONTENTS

4.1 Concurrency, Shared Data & The Thread API

Definition: Thread, Race Condition & Critical Section

• Thread: A lightweight unit of execution within a process. Multiple threads within the same process share the same Address Space (Code, Heap, Open Files), but each thread possesses its own private Program Counter, CPU registers, and Stack (Thread-Local Stack).

• Critical Section: A segment of code that accesses shared variables and must not be concurrently executed by more than one thread.

• Race Condition: A flaw where the program outcome depends on the non-deterministic timing and interleaved scheduling of concurrent threads.

• Mutual Exclusion: The property that guarantees only one thread enters the critical section at any given time.

POSIX Threads (pthreads) API

- **• pthread_create():** Spawns a new concurrent thread: `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);`
- **• pthread_join():** Suspends caller until the specified thread completes: `int pthread_join(pthread_t thread, void **retval);`

4.2 Locks & Hardware Synchronization Primitives

Evaluating Locks: 3 Essential Criteria

1. Mutual Exclusion: Does the lock successfully prevent multiple threads from entering the critical section simultaneously?

2. Fairness: Does every contending thread get a fair chance at acquiring the lock without Starvation?

3. Performance: What is the time/CPU overhead when single-threaded, contending on a single CPU, or contending across multi-cores?

Hardware Atomic Instructions

Atomic Hardware Instruction	C-Style Conceptual Behavior	Properties & Use Case
Test-And-Set (TAS)	int TestAndSet(int *old_ptr, int new) { int old = *old_ptr; *old_ptr = new; return old; }	Atomically sets new value and returns old value in a single memory bus lock. Used to build basic Spinlocks.
Compare-And-Swap (CAS)	int CompareAndSwap(int *ptr, int expected, int new) { int actual = *ptr; if (actual == expected) *ptr = new; return actual; }	Atomically compares *ptr to expected value; if equal, updates to new. Powerful lock-free primitive.
Load-Linked / Store-Conditional (LL/SC)	LL loads value; SC stores new value only if no intervening update occurred to address since LL.	Hardware primitive on ARM and MIPS architectures used to construct atomic read-modify-write sequences.

Spinlocks vs. Sleeping Locks (Yield & Futex)

- • **Simple Spinlock:** A thread spins in a busy-wait `while(TestAndSet(&lock, 1) == 1)` loop until lock is released. Wastes CPU cycles if thread holding lock is preempted.
- • **Spin with Yield (`sched_yield()`):** Spinning thread voluntarily yields CPU to OS scheduler, moving from Running to Ready state. Reduces CPU waste, but context switch costs remain.
- • **Linux Futex (Fast Userspace Mutex):** Hybrid lock: attempts fast atomic CAS in user space without kernel entry. If contended, calls `futex(FUTEX_WAIT)` to put the thread to sleep in kernel queue until awakened via `futex(FUTEX_WAKE)` by lock release.

4.3 Semaphores: Definition & Primitives

Definition: Semaphore (Edsger Dijkstra, 1965)

A Semaphore is a synchronization object initialized to an integer value that supports two atomic operations:

• sem_wait() / P() / down(): Decrements the semaphore value. If the value is negative (< 0), the calling thread is blocked and placed on the semaphore's waiting queue.

• sem_post() / V() / up(): Increments the semaphore value. If there are waiting threads, one blocked thread is awakened.

Two Primary Uses of Semaphores

- **1. Binary Semaphores (Mutex Locks):** Initialized to 1. Used to protect critical sections (`sem_wait(&mutex); // Critical Section; sem_post(&mutex);`).

- **2. Ordering Semaphores (Signaling / Join Pattern):** Initialized to 0. Parent thread calls `sem_wait(&s);` to wait; child thread calls `sem_post(&s);` upon finishing to signal the parent.

4.4 Classic Synchronization Problems

1. The Producer-Consumer (Bounded Buffer) Problem

Producers place data items into a shared buffer of size MAX; Consumers remove items. Solved using 3 semaphores:

- • **`empty`:** Counting semaphore initialized to `MAX` (tracks free buffer slots).
- • **`full`:** Counting semaphore initialized to `0` (tracks filled buffer slots).
- • **`mutex`:** Binary semaphore initialized to `1` (ensures mutual exclusion on buffer insert/extract).

> **NOTE / EXAM POINTER**
> EXAM POINTER: The order of sem_wait calls is critical! Producer must call sem_wait(&empty) BEFORE sem_wait(&mutex). Reversing the order causes a Deadlock if the buffer is full!

2. The Reader-Writer Problem

Multiple concurrent readers can access shared data simultaneously, but a writer requires exclusive mutual exclusion:

- • **`rw_mutex`:** Binary semaphore (initialized to 1) protecting writer access.
- • **`mutex`:** Binary semaphore (initialized to 1) protecting `read_count` integer variable.

First reader acquires `rw_mutex`; last reader releases `rw_mutex`.

3. The Dining Philosophers Problem

5 philosophers sit around a table with 5 chopsticks, needing both left and right chopsticks to eat. If all grab left chopstick simultaneously, Circular Wait Deadlock occurs.

- • **Solution (Breaking Symmetry):** Philosophers 0 to 3 pick up Left then Right; Philosopher 4 picks up Right then Left.

4.5 Deadlocks & Common Concurrency Bugs

The 4 Coffman Deadlock Conditions (ALL 4 must hold for Deadlock to occur)

1. Mutual Exclusion: Resources cannot be shared; held in non-shareable mode.

2. Hold and Wait: A thread holds at least one resource while waiting to acquire additional resources held by other threads.

3. No Preemption: Resources cannot be forcibly confiscated from a thread; must be released voluntarily.

4. Circular Wait: A closed chain of threads exists {T_1, T_2, ..., T_n} such that T_i is waiting for a resource held by T_(i+1) (and T_n waits for T_1).

Deadlock Prevention Strategies

Coffman Condition	Prevention Strategy & Implementation
Prevent Circular Wait (Most Common)	Enforce a strict Global Lock Acquisition Order (e.g., if acquiring Lock A and Lock B, all threads must acquire Lock A before Lock B).
Prevent Hold and Wait	Require threads to acquire all required locks atomically in a single global lock acquisition routine.
Prevent No Preemption	Use `pthread_mutex_trylock()`. If a lock cannot be acquired, release all currently held locks and retry later.
Prevent Mutual Exclusion	Use lock-free data structures built with atomic hardware instructions (CAS / Fetch-And-Add).

4.6 High-Yield University Exam Question Bank & Model Answers

Part A: Short Answer Questions (2–3 Marks)

- **Q1. What is a Race Condition? How does Mutual Exclusion resolve it?**

Answer: A Race Condition occurs when multiple concurrent threads read and write shared data without synchronization, producing erratic results depending on thread interleaving. Mutual Exclusion ensures only one thread enters the critical section at a time, enforcing atomic updates.

- **Q2. Define the sem_wait() and sem_post() operations.**

Answer: `sem_wait()` atomically decrements the semaphore value; if the value is < 0, the calling thread blocks. `sem_post()` atomically increments the semaphore value; if any threads are waiting, it awakens one blocked thread.

- **Q3. State the 4 Coffman conditions for Deadlock.**

Answer: (1) Mutual Exclusion, (2) Hold and Wait, (3) No Preemption, and (4) Circular Wait.

- **Q4. Differentiate between a Spinlock and a Sleeping Lock.**

Answer: A Spinlock busy-waits in a CPU loop until the lock becomes free (good for very short critical sections on multi-cores). A Sleeping Lock puts the contending thread to sleep in the kernel queue, yielding CPU to other runnable threads.

Part B: Long Answer Questions (8–10 Marks)

- **Q5. Solve the Producer-Consumer (Bounded Buffer) problem using Semaphores. Provide complete C-style pseudocode and explain why mutex order is critical.**

Model Answer Outline:

- 1. Problem description: Shared bounded buffer of capacity N; Producers add items, Consumers remove items.
- 2. Semaphore definitions: `empty` (initialized to N), `full` (initialized to 0), `mutex` (initialized to 1).
- 3. Producer algorithm pseudocode: sem_wait(&empty) → sem_wait(&mutex) → InsertItem() → sem_post(&mutex) → sem_post(&full).
- 4. Consumer algorithm pseudocode: sem_wait(&full) → sem_wait(&mutex) → RemoveItem() → sem_post(&mutex) → sem_post(&empty).
- 5. Detailed deadlock analysis proving why acquiring `mutex` before `empty` causes deadlock when the buffer is full.

- **Q6. Discuss Deadlocks in Operating Systems. Explain the 4 Coffman conditions, Resource Allocation Graphs (RAG), and Deadlock Prevention vs. Banker's Algorithm Avoidance.**

Model Answer Outline:

- 1. Definition of Deadlock: Permanent blocking of a set of threads.
- 2. Comprehensive breakdown of the 4 Coffman conditions: Mutual Exclusion, Hold & Wait, No Preemption, Circular Wait.
- 3. Resource Allocation Graphs (RAG): Request edges, Assignment edges, cycle detection for deadlock identification.
- 4. Deadlock Prevention: Eliminating Circular Wait via Global Lock Ordering.
- 5. Deadlock Avoidance: Dijkstra's Banker's Algorithm (Safe state evaluation, Available, Max, Allocation, Need matrices).

4.7 Unit IV Summary & Quick Revision Sheet

Concurrency & Sync Domain	Key Theoretical & Code Takeaways for Exams
Threads & Race Conditions	Shared address space, private stack/registers. Race conditions resolved via Mutual Exclusion.
Lock Primitives	Test-And-Set (TAS), Compare-And-Swap (CAS), Spinlock vs Yield vs Futex (Userspace fast path).
Semaphores	sem_wait() decrements & blocks if <0; sem_post() increments & wakes. Binary (1) vs Counting.
Classic Problems	Producer-Consumer (empty=N, full=0, mutex=1), Reader-Writer (rw_mutex), Dining Philosophers (Asymmetric pickup).
Deadlock Conditions	4 Coffman: Mutex, Hold&Wait, No Preemption, Circular Wait. Prevent via Global Lock Hierarchy.