Process Synchronization and Semaphores

Week 9

SDB

Autumn 2025

Agenda: Process Synchronization with Semaphores

Ensuring Order and Consistency in Concurrent Execution

1. **Race Conditions and Critical Section Problem** *(The core challenge)*
2. **Requirements for Mutual Exclusion** *(What makes a solution correct?)*
3. **Semaphores: Types and Operations** *(A powerful synchronization tool)*
4. **Classic Use Case: Producer–Consumer Problem** *(Applying semaphores to a common pattern)*
5. **Pitfalls in Semaphore Usage** *(What can go wrong?)*

Agenda: Process Synchronization with Semaphores

Ensuring Order and Consistency in Concurrent Execution

1. **Race Conditions and Critical Section Problem** *(The core challenge)*
2. **Requirements for Mutual Exclusion** *(What makes a solution correct?)*
3. **Semaphores: Types and Operations** *(A powerful synchronization tool)*
4. **Classic Use Case: Producer–Consumer Problem** *(Applying semaphores to a common pattern)*
5. **Pitfalls in Semaphore Usage** *(What can go wrong?)*

Think Ahead: Data Integrity in Parallel

When multiple threads or processes operate on shared data concurrently, unexpected and incorrect results can emerge. How can we design our code to ensure that critical operations on shared resources are performed atomically and correctly, preventing data corruption and maintaining system integrity?

Race Conditions: The Danger of Concurrent Access

Definition:

A **race condition** occurs when two or more processes or threads concurrently access and manipulate shared data, and the outcome of the execution depends on the particular order in which the accesses take place (i.e., the relative timing of events).

Race Conditions: The Danger of Concurrent Access

Definition:

A **race condition** occurs when two or more processes or threads concurrently access and manipulate shared data, and the outcome of the execution depends on the particular order in which the accesses take place (i.e., the relative timing of events).

Common Example: Incrementing a Shared Counter

- Consider a shared integer variable ‘count‘ initialized to 0.
- Two threads, T1 and T2, both want to increment ‘count‘ by 1.
- The operation ‘count++‘ is not atomic; it typically involves three machine instructions:
  1. Load ‘count‘ from memory into a register.
  2. Increment the register.
  3. Store the register's value back to ‘count‘ in memory.

Race Conditions: The Danger of Concurrent Access

Definition:

A **race condition** occurs when two or more processes or threads concurrently access and manipulate shared data, and the outcome of the execution depends on the particular order in which the accesses take place (i.e., the relative timing of events).

Common Example: Incrementing a Shared Counter

- Consider a shared integer variable ‘count‘ initialized to 0.
- Two threads, T1 and T2, both want to increment ‘count‘ by 1.
- The operation ‘count++‘ is not atomic; it typically involves three machine instructions:
  1. Load ‘count‘ from memory into a register.
  2. Increment the register.
  3. Store the register's value back to ‘count‘ in memory.

Result: Without proper synchronization, the final value of ‘count‘ might be 1 instead of the expected 2.

Race Condition Example: Interleaved Execution I

Scenario: ‘count++‘ with Two Threads

- Initial ‘count = 0‘
- Thread 1 wants ‘count++‘
- Thread 2 wants ‘count++‘

Expected Final ‘count‘: 2

Race Condition Example: Interleaved Execution II

Analogy: Concurrent Bank Deposits

Imagine two people trying to deposit money into the same bank account at the same time. If both read the current balance, add their deposit, and then write the new balance back without coordinating, one deposit might be lost.

The Critical Section Problem I

Critical Section:

A **critical section** is a segment of code where a process or thread accesses shared resources (data, files, devices). Only one process/thread should be allowed to execute its critical section at any given time to prevent race conditions.

Goal: Design a protocol that processes can use to cooperate such that no two processes are in their critical sections simultaneously. **A Solution to the Critical**

Section Problem Must Satisfy Three Requirements:

1. **Mutual Exclusion:**
   - **Requirement:** If process P_i is executing in its critical section, then no other process can be executing in its critical section.
   - **Why it's needed:** This is the core requirement to prevent race conditions and ensure data integrity.
2. **Progress:**

The Critical Section Problem II

- **Requirement:** If no process is executing in its critical section and some processes want to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next. This selection cannot be postponed indefinitely.
- **Why it's needed:** Prevents deadlocks where processes are perpetually waiting for each other or a decision that never comes.

3. **Bounded Waiting:**
   - **Requirement:** A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - **Why it's needed:** Prevents starvation, where a process might repeatedly lose the "race" to enter its critical section. Ensures fairness.

What is a Semaphore? A Synchronization Primitive I

Definition

A **semaphore** is an integer variable used for signaling and synchronization among processes or threads. It acts as a counter that can be incremented or decremented.

Key Characteristics:

- Semaphores are accessed only through two **atomic operations**: ‘wait()‘ and ‘signal()‘.
- **Atomic means indivisible:** These operations are guaranteed to execute completely without interruption from other processes, preventing race conditions on the semaphore itself.

The Two Basic Operations (originally P and V):

- `wait(S)` (also called ‘P(S)‘ or ‘down()‘):
  - Decrements the semaphore value S.

What is a Semaphore? A Synchronization Primitive II

- If S becomes negative (or ≤ 0 depending on implementation), the process executing the ‘wait()‘ operation is **blocked** (put into a waiting queue) until S becomes positive.
- Conceptually: "acquire a resource" or "wait for a signal."

- `signal(S)` (also called ‘V(S)‘ or ‘up()‘):
  - Increments the semaphore value S.
  - If there are processes waiting on this semaphore, one of them is **unblocked** (moved from the waiting queue to the ready queue).
  - Conceptually: "release a resource" or "send a signal."

Types of Semaphores:

- **Binary Semaphore (Mutex):** Integer value can only be 0 or 1. Used for mutual exclusion (like a lock). Initialized to 1.
- **Counting Semaphore:** Integer value can range over an unrestricted domain. Used to control access to a resource that has multiple instances. Initialized to the number of available resources.

Semaphore Usage: Mutual Exclusion Example I

Protecting a Critical Section with a Binary Semaphore (Mutex)

- A binary semaphore (often called ‘mutex‘) is initialized to 1.
- A process/thread must perform ‘wait(mutex)‘ before entering its critical section.
- It must perform ‘signal(mutex)‘ after exiting its critical section.

```
// Global binary semaphore, initialized to 1
semaphore mutex = 1;

void process_or_thread_function() {
    while (true) {
        // Entry Section: Request access to critical section
        wait(mutex); // Decrements mutex. If mutex is 0, thread blocks.

        // Critical Section: Access shared data (e.g., ‘count++‘)
        // Only one thread can be here at a time.
        // ... (your shared data manipulation code) ...

        // Exit Section: Release access
        signal(mutex); // Increments mutex. If threads are waiting, one is unblocked.

        // Remainder Section: Non-critical operations
        // ...
    }
}
```

Semaphore Usage: Mutual Exclusion Example II

How it ensures Mutual Exclusion:

- If ‘mutex‘ is 1, the first ‘wait()‘ succeeds, ‘mutex‘ becomes 0, and the thread enters the critical section.
- If another thread then calls ‘wait()‘, ‘mutex‘ is 0, so that thread blocks.
- When the first thread calls ‘signal()‘, ‘mutex‘ becomes 1, and the blocked thread (if any) is unblocked and can now enter.

This simple pattern ensures that only one process is in the critical section at any time.

Classic Synchronization Problem: The Producer–Consumer Problem I

The Scenario:

- Two types of processes:
  - A **Producer** process that generates data items.
  - A **Consumer** process that consumes data items.
- They share a common, fixed-size **buffer** (e.g., an array or queue).

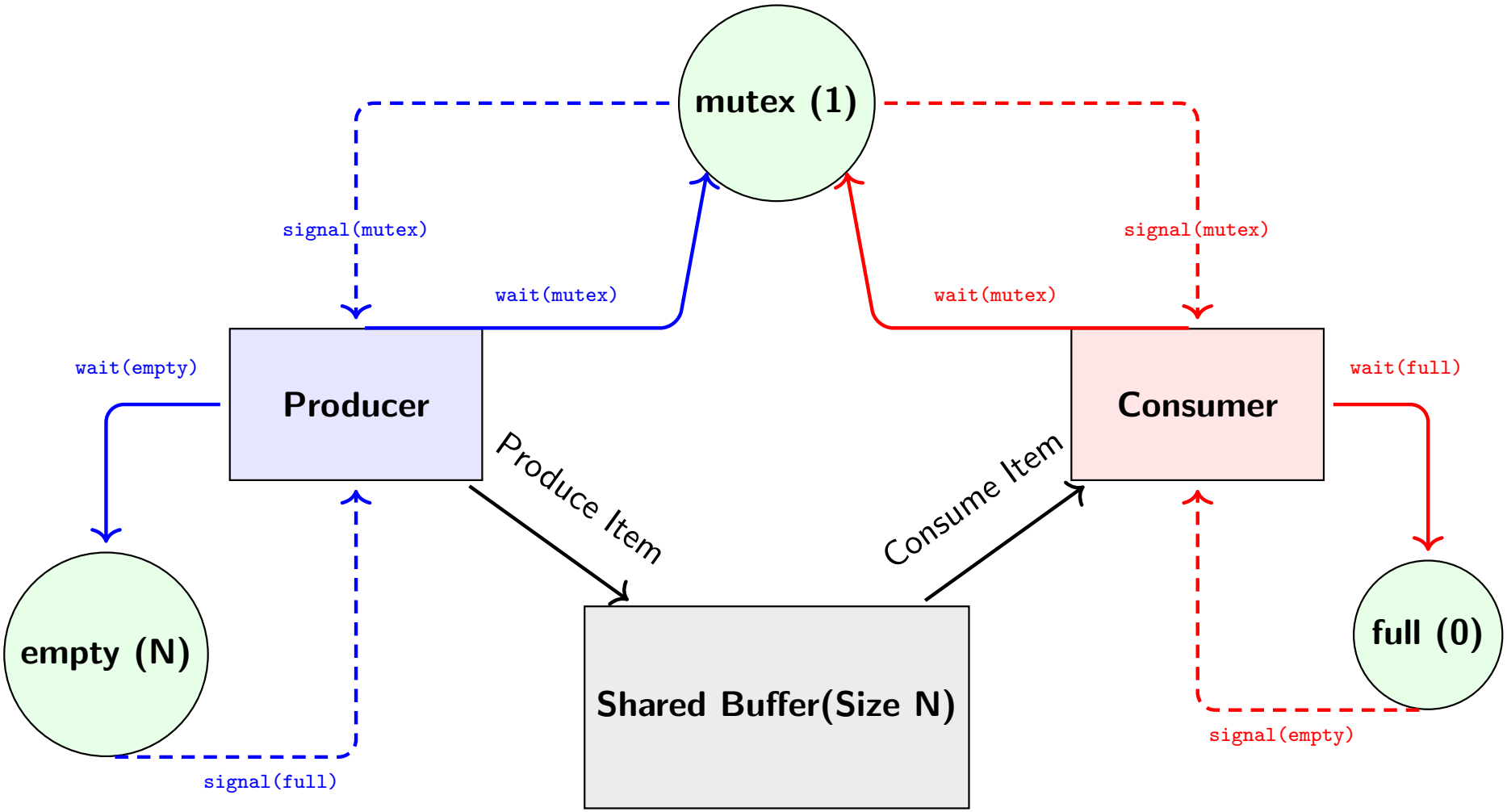
The Challenge (What needs synchronization):

1. **Mutual Exclusion (Shared Buffer Access):**
   - Only one process (producer or consumer) should be allowed to access the buffer at a time to prevent corrupting the data (e.g., two producers adding at the same index, or a producer and consumer accessing the same slot concurrently).

Classic Synchronization Problem: The Producer–Consumer Problem II

2. **Coordination (Buffer Full/Empty):**
   - The producer must not try to add an item to a full buffer. It must wait if the buffer is full.
   - The consumer must not try to remove an item from an empty buffer. It must wait if the buffer is empty.

Producer–Consumer: Visual Flow with Semaphores



Producer–Consumer with Semaphores: The Code I

Semaphores Initialization:

- ‘mutex = 1‘: Binary semaphore for mutual exclusion on the buffer.
- ‘empty = N‘: Counting semaphore, initialized to buffer size ‘N‘, counts empty slots.
- ‘full = 0‘: Counting semaphore, initialized to 0, counts full slots.

```
semaphore mutex = 1;       // Controls access to the buffer (mutual exclusion)
semaphore empty = N;       // Counts empty slots in the buffer (initially N)
semaphore full = 0;        // Counts full slots in the buffer (initially 0)

// Producer Process/Thread
void Producer() {
  while (true) {
    // 1. Produce an item (not involving shared resources)
    // ... item = produce_new_item() ...

    // 2. Wait for an empty slot and acquire buffer lock
    wait(empty);     // Decrement empty count. If 0, blocks (buffer full).
    wait(mutex);     // Acquire mutex. If 0, blocks (buffer busy).

    // 3. Add item to buffer (Critical Section)
    // ... add_item_to_buffer(item) ...

    // 4. Release buffer lock and signal a full slot
```

Producer–Consumer with Semaphores: The Code II

```
    signal(mutex);  // Release mutex. Unblocks waiting consumer/producer.
    signal(full);   // Increment full count. Unblocks waiting consumer.
  }
}

// Consumer Process/Thread
void Consumer() {
  while (true) {
    // 1. Wait for a full slot and acquire buffer lock
    wait(full);     // Decrement full count. If 0, blocks (buffer empty).
    wait(mutex);    // Acquire mutex. If 0, blocks (buffer busy).

    // 2. Remove item from buffer (Critical Section)
    // ... item = remove_item_from_buffer() ...

    // 3. Release buffer lock and signal an empty slot
    signal(mutex);  // Release mutex. Unblocks waiting producer/consumer.
    signal(empty);  // Increment empty count. Unblocks waiting producer.

    // 4. Consume the item (not involving shared resources)
    // ... consume_item(item) ...
  }
}
```

Producer–Consumer with Semaphores: The Code III

Explanation of Order:

- ‘wait(empty)‘ / ‘wait(full)‘: Act as ”gatekeepers” to ensure there are resources (empty/full slots) before even attempting to enter the critical section.
- ‘wait(mutex)‘: Ensures only one process can modify the buffer at a time.
- ‘signal(mutex)‘: Releases the critical section.
- ‘signal(full)‘ / ‘signal(empty)‘: Inform the other process type that a slot has become available/full.

The order of ‘wait()‘ and ‘signal()‘ is crucial to prevent deadlocks and ensure correct behavior!

Pitfalls in Semaphore Usage: When Things Go Wrong I

Semaphores are powerful but low-level primitives. Incorrect usage can lead to severe concurrency issues.

Deadlock:

- **Cause:** Two or more processes are indefinitely waiting for an event that can only be caused by one of the waiting processes (e.g., releasing a resource that another needs).
- **Example:** If the Producer in the P-C problem did ‘wait(mutex); wait(empty);‘ and the Consumer did ‘wait(mutex); wait(full);‘ (reversed order), a deadlock could occur.
- **Consequence:** System becomes unresponsive, processes halt.

Starvation:

- **Cause:** A process may wait indefinitely within the semaphore's waiting queue, even though other processes are continually entering and leaving their critical sections. This can happen with unfair scheduling of the waiting queue.

Pitfalls in Semaphore Usage: When Things Go Wrong II

- **Example:** If a ‘signal()‘ operation always wakes up the same waiting process, others might never get a chance.
- **Consequence:** Some processes never make progress.

Busy Waiting (Spinlock):

- **Cause:** In some naive semaphore implementations, if a ‘wait()‘ operation cannot proceed, the process might repeatedly check the semaphore value in a tight loop instead of blocking (sleeping).
- **Consequence:** Wastes CPU cycles by keeping the CPU busy even when no useful work is being done, especially detrimental on single-core systems.
- **Modern OS Semaphores:** Typically implement ‘wait()‘ by putting the process to sleep (blocking) and ‘signal()‘ by waking it up, avoiding busy waiting.

Pitfalls in Semaphore Usage: When Things Go Wrong III

Mitigation & Better Practices:

- Use **higher-level synchronization constructs** (like Mutex Locks, Condition Variables, Monitors) that abstract away the raw semaphore operations and handle many pitfalls automatically.
- Rigorous **design and analysis** of all entry/exit conditions and potential interleavings.
- Follow established **best practices** and patterns for concurrent programming.

Quick Quiz: Understanding Semaphores

Test Your Conceptual Understanding:

1. **Define:** What is the "critical section" in concurrent programming, and why is it important to protect it?

2. **Requirements:** List and briefly explain the three fundamental requirements that any correct solution to the Critical Section Problem must satisfy.

3. **Semaphore Operations:** Describe what happens when a process calls ‘wait(S)‘ if ‘S‘ is currently 0. What happens when a process calls ‘signal(S)‘?

4. **Producer-Consumer Logic:** In the Producer-Consumer problem, why is it crucial to use *three* semaphores (‘mutex‘, ‘empty‘, ‘full‘) instead of just one or two? What specific purpose does each serve?

Quick Quiz: Understanding Semaphores

Test Your Conceptual Understanding:

1. **Define:** What is the "critical section" in concurrent programming, and why is it important to protect it?

2. **Requirements:** List and briefly explain the three fundamental requirements that any correct solution to the Critical Section Problem must satisfy.

3. **Semaphore Operations:** Describe what happens when a process calls ‘wait(S)‘ if ‘S‘ is currently 0. What happens when a process calls ‘signal(S)‘?

4. **Producer-Consumer Logic:** In the Producer-Consumer problem, why is it crucial to use *three* semaphores (‘mutex‘, ‘empty‘, ‘full‘) instead of just one or two? What specific purpose does each serve?

Think & Discuss

Formulate your answers before reviewing the solutions or discussing with peers.

Key Takeaways I

- **Race Conditions** occur when concurrent access to shared data leads to unpredictable outcomes.
- The **Critical Section Problem** aims to prevent race conditions by ensuring only one process/thread enters a critical section at a time, adhering to **Mutual Exclusion, Progress, and Bounded Waiting**.
- **Semaphores** are low-level integer synchronization primitives, providing atomic ‘wait()‘ (decrement/block) and ‘signal()‘ (increment/unblock) operations.
- **Binary Semaphores (Mutexes)** are used for mutual exclusion. **Counting Semaphores** manage multiple resources.
- The **Producer-Consumer Problem** is a classic example demonstrating the use of semaphores for both mutual exclusion and coordination (empty/full buffer slots).
- Improper use of semaphores can lead to severe issues like **deadlock**, **starvation**, and **busy waiting**.

Key Takeaways II

Reflection Prompt: The Balance of Power

Semaphores give programmers precise control over synchronization but are prone to errors. Why do you think higher-level constructs are often preferred in modern programming, even if they internally use semaphores or similar primitives? What does "abstraction" buy us in this context?

Next Week Preview: Advanced Synchronization & Deadlocks

Beyond Semaphores: Complex Scenarios and Problem Solving

- **Mutex Locks:** A simpler, more common mutual exclusion primitive.
- **Condition Variables:** For threads to wait for specific conditions (complementary to mutexes).
- **Monitors:** High-level language constructs encapsulating shared data and synchronization.
- **Classical Synchronization Problems (Revisited):**
  - Dining Philosophers Problem
  - Readers-Writers Problem
- **Deadlock Concepts:** Characterization, Prevention, Avoidance, Detection, Recovery.

Next Week Preview: Advanced Synchronization & Deadlocks

Beyond Semaphores: Complex Scenarios and Problem Solving

- **Mutex Locks:** A simpler, more common mutual exclusion primitive.
- **Condition Variables:** For threads to wait for specific conditions (complementary to mutexes).
- **Monitors:** High-level language constructs encapsulating shared data and synchronization.
- **Classical Synchronization Problems (Revisited):**
  - Dining Philosophers Problem
  - Readers-Writers Problem
- **Deadlock Concepts:** Characterization, Prevention, Avoidance, Detection, Recovery.

Prep Tip for Next Session

Think about how the "Producer-Consumer" problem would change if you had multiple producers and multiple consumers. How might 'wait()' and 'signal()' calls need to be more carefully managed? This will prepare you for more complex scenarios.

Outline

Exercise: Applying Semaphores to a New Problem I

Part 1: The ‘Incrementer‘ and ‘Decrementor‘ Problem

Imagine two types of threads sharing a global integer variable ‘value‘, initially 0.

- ‘Incrementer‘ threads: Each calls ‘value++‘ 100 times.
- ‘Decrementor‘ threads: Each calls ‘value–‘ 100 times.

You have one ‘Incrementer‘ thread and one ‘Decrementor‘ thread.

Task:

1. Why would a race condition occur if ‘value++‘ and ‘value–‘ are not synchronized?
2. Write pseudocode for the ‘Incrementer‘ and ‘Decrementor‘ functions using a **binary semaphore (‘mutex‘)** to ensure the final ‘value‘ is 0. Initialize the semaphore appropriately.

Part 2: The "Limited Resources" Problem

You have 5 identical printers shared among multiple processes. Processes request a printer, use it, and then release it.

Task:

Exercise: Applying Semaphores to a New Problem II

1. Which type of semaphore (binary or counting) would you use to manage access to these printers? Why?
2. Write pseudocode for a ‘request_printer()‘ function and a ‘release_printer()‘ function using your chosen semaphore. Initialize the semaphore appropriately.
3. What would happen if more than 5 processes tried to acquire a printer concurrently?

Reminder

Focus on the correct order of ‘wait()‘ and ‘signal()‘ operations and proper semaphore initialization.

Appendix: Advanced Topics to Explore I

Deepening Your Understanding of Process Synchronization

I. Synchronization Primitive Internals

- **Hardware Support for Synchronization:** Explore atomic instructions like ‘TestAndSet‘, ‘CompareAndSwap‘ (CAS), and ‘FetchAndAdd‘ that are fundamental building blocks for semaphores and other locks.
- **Spinlocks vs. Mutexes (Detailed):** When busy-waiting is acceptable (e.g., in kernel code for very short critical sections) versus when blocking is preferred.
- **Memory Barriers/Fences:** How they ensure memory operations are visible to other processors in a specific order, crucial for multi-core synchronization.

Appendix: Advanced Topics to Explore II

II. Advanced Synchronization Constructs

- **Readers-Writers Locks:** A type of mutex that allows multiple readers but only one writer at a time, optimizing for read-heavy workloads.
- **Barriers:** A synchronization primitive that makes processes wait at a certain point until all participating processes have reached that point.
- **Transactional Memory (TM):** A high-level concurrency control mechanism that allows multiple threads to execute transactions on shared memory concurrently, with hardware or software handling conflicts.

Appendix: Advanced Topics to Explore III

III. Concurrency Challenges & Solutions

- **Priority Inversion (Revisited):** How it manifests in synchronization and solutions like Priority Inheritance Protocol.
- **Lock-Free and Wait-Free Data Structures:** Designing concurrent data structures without traditional locks, using atomic operations to guarantee progress (e.g., non-blocking queues).
- **Distributed Synchronization:** How processes synchronize across multiple machines in a network (e.g., distributed mutexes, Paxos, Raft).