Classical Synchronization Problems

L2

SDB

Autumn 2025

Agenda: Classical Synchronization Problems & Solutions

Applying Synchronization Primitives to Common Concurrency Challenges

1. **The Dining Philosophers Problem** *(Resource contention leading to deadlock)*
2. **The Readers–Writers Problem** *(Managing concurrent reads and exclusive writes)*
3. **The Bounded-Buffer Problem (Review)** *(Producer-Consumer revisited)*
4. **Mutexes vs. Semaphores** *(Choosing the right tool)*
5. **Synchronization Primitives in Real-World APIs** *(Mapping theory to practice)*

Agenda: Classical Synchronization Problems & Solutions

Applying Synchronization Primitives to Common Concurrency Challenges

1. **The Dining Philosophers Problem** *(Resource contention leading to deadlock)*
2. **The Readers–Writers Problem** *(Managing concurrent reads and exclusive writes)*
3. **The Bounded-Buffer Problem (Review)** *(Producer-Consumer revisited)*
4. **Mutexes vs. Semaphores** *(Choosing the right tool)*
5. **Synchronization Primitives in Real-World APIs** *(Mapping theory to practice)*

Think Ahead: Beyond Simple Mutual Exclusion

We've learned how to protect a single critical section. But what happens when multiple processes need to acquire **multiple** shared resources, or when different types of access (read vs. write) require nuanced synchronization? How do we prevent complex issues like deadlock and starvation in these scenarios?

Classical Problem 1: The Dining Philosophers Problem I

Origin: Proposed by Edsger W. Dijkstra (1965) to illustrate issues of resource contention and deadlock in parallel computing.

Setup:

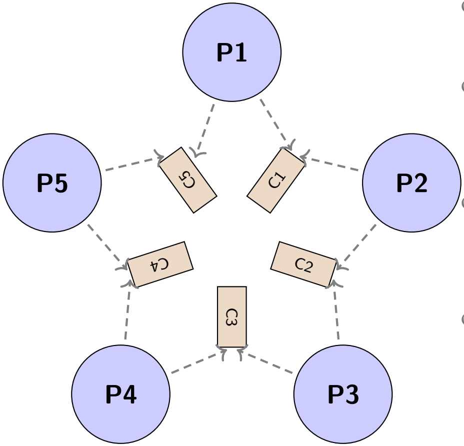
- **N Philosophers** are seated around a circular table. (Commonly 5 for simplicity).
- Each philosopher alternates between two states: **thinking** and **eating**.
- In the center of the table is a bowl of food.
- Between each pair of philosophers is a single **chopstick** (shared resource).
- To **eat**, a philosopher needs to pick up **both** the chopstick to their left and the chopstick to their right.

Classical Problem 1: The Dining Philosophers Problem II

The Challenge:

- Design a synchronization scheme that allows the philosophers to eat concurrently without:
  - **Deadlock:** A situation where all philosophers are forever waiting for a chopstick that another philosopher holds.
  - **Starvation:** A situation where a particular philosopher is never able to eat (continuously out-competed).
  - While maximizing **concurrency** (allowing as many as possible to eat simultaneously).

Dining Philosophers Problem (Visual Representation)



- Five philosophers (‘P1-P5‘) are seated around a table.
- There are five chopsticks (‘C1-C5‘), with one placed between each pair of philosophers.
- A philosopher P_i needs **both their left** C_i **and right** $C_{(i+1)\%N}$ **chopsticks** to eat.
- The central issue arises when all five philosophers simultaneously pick up their left chopstick. In this state, no one can pick up their right chopstick, leading to a deadlock.

Dining Philosophers: A Simple Semaphore Solution I

Basic (and Flawed) Semaphore Solution:

- Use one binary semaphore for each chopstick, initialized to 1 (available).
- 'chopstick[i]' represents the chopstick to the left of philosopher 'i'.

```
semaphore chopstick[5] = {1, 1, 1, 1, 1}; // One semaphore per chopstick, initially
     available

void philosopher(int i) { // i = philosopher ID (0 to 4)
  while (true) {
    // Phase 1: Thinking
    think(); // Philosopher is not hungry

    // Phase 2: Hungry - Attempt to pick up chopsticks
    // Try to pick up left chopstick
    wait(chopstick[i]);
    // Try to pick up right chopstick
    wait(chopstick[(i + 1) % 5]);

    // Phase 3: Eating (Critical Section)
    eat(); // Philosopher has both chopsticks and is eating

    // Phase 4: Done Eating - Put down chopsticks
    // Put down left chopstick
    signal(chopstick[i]);
    // Put down right chopstick
    signal(chopstick[(i + 1) % 5]);
  }
}
```

Dining Philosophers: A Simple Semaphore Solution II

Issue with this Solution: Potential for DEADLOCK

- If all 5 philosophers simultaneously pick up their **left** chopstick:
  - P0 holds C0, waits for C1
  - P1 holds C1, waits for C2
  - P2 holds C2, waits for C3
  - P3 holds C3, waits for C4
  - P4 holds C4, waits for C0
- This forms a **circular wait** condition, a classic sign of deadlock. No philosopher can acquire their second chopstick, and thus none can release their first. All are stuck.

This highlights that simply using semaphores for mutual exclusion isn't enough; the acquisition order matters for multiple resources.

Dining Philosophers: Deadlock Prevention Strategies I

To prevent deadlock, we must break one of the four necessary conditions for deadlock (Mutual Exclusion, Hold and Wait, No Preemption, Circular Wait). For Dining Philosophers, we typically break "Hold and Wait" or "Circular Wait".

Common Strategies to Prevent Deadlock:

1. **Limit the Number of Philosophers:**
   - Allow at most $N - 1$ philosophers to be hungry (and attempt to pick up chopsticks) at any time.
   - This can be implemented using a counting semaphore initialized to $N - 1$ (e.g., 'waiter = N-1'). Philosophers 'wait(waiter)' before picking up any chopstick and 'signal(waiter)' after putting both down.
   - **Prevents:** Circular Wait (ensures at least one philosopher can always pick up both).
2. **Asymmetric Pickup Order:**
   - Introduce an ordering rule:
     - ★ Even-indexed philosophers (P0, P2, P4) pick up their **left** chopstick first, then their **right**.

Dining Philosophers: Deadlock Prevention Strategies II

        - ★ Odd-indexed philosophers (P1, P3) pick up their **right** chopstick first, then their **left**.
    - This breaks the symmetry and thus the circular wait condition.
    - **Prevents:** Circular Wait.

3. **Resource Hierarchy (Chopstick Numbering):**
    - Assign a unique number to each chopstick (e.g., C0, C1, C2, C3, C4).
    - Require all philosophers to pick up the lower-numbered chopstick first, then the higher-numbered one.
    - Example: P2 (needs C2, C3) picks C2 then C3. P3 (needs C3, C4) picks C3 then C4.
    - **Prevents:** Circular Wait (as resources are acquired in a strict order).
    - **Drawback:** This solution can be less intuitive and might lead to reduced concurrency in some cases.

Classical Problem 2: The Readers–Writers Problem I

Setup:

- A shared data resource (e.g., a file, a database record).
- Multiple **Reader** processes/threads that only read the data.
- Multiple **Writer** processes/threads that modify the data.

Concurrency Rules:

- **Multiple Readers Allowed:** Any number of readers can access the shared data concurrently. (Reading doesn't change data, so it's safe).
- **Exclusive Writer:** Only one writer can access the shared data at a time.
- **No Reader During Write:** No reader can access the data if a writer is accessing it.
- **No Writer During Read:** No writer can access the data if any reader is accessing it.

The Challenge:

Classical Problem 2: The Readers–Writers Problem II

- Design a synchronization mechanism that satisfies these rules while avoiding **starvation** (either readers or writers perpetually waiting) and maximizing concurrency.

Two Main Variants:

1. **Reader-Priority:** Favors readers; a writer may be forced to wait indefinitely if a steady stream of readers arrives.
2. **Writer-Priority:** Favors writers; new readers are blocked if a writer is waiting, potentially leading to reader starvation.

Readers–Writers Problem: Reader-Priority Solution I

Shared Variables:

- ‘semaphore rw = 1;‘: Binary semaphore for controlling access to the **shared data** itself. Writers ‘wait(rw)‘ and ‘signal(rw)‘. The first reader ‘wait(rw)‘ and the last reader ‘signal(rw)‘.
- ‘semaphore mutex = 1;‘: Binary semaphore for protecting ‘readcount‘ (ensuring ‘readcount‘ updates are atomic).
- ‘int readcount = 0;‘: Keeps track of how many readers are currently accessing the data.

Readers–Writers Problem: Reader-Priority Solution II

```c
semaphore rw = 1;     // Controls access to the shared resource (read/write lock)
semaphore mutex = 1; // Protects ‘readcount‘ variable
int readcount = 0;   // Number of active readers

// Reader process/thread
void Reader() {
  while (true) {
    // Entry Section
    wait(mutex);          // Lock to protect readcount
    readcount++;          // Increment active readers count
    if (readcount == 1) { // If this is the first reader, acquire rw lock
      wait(rw);           // Blocks writers, allows other readers
    }
    signal(mutex);        // Unlock mutex for readcount

    // Reading Section (multiple readers allowed here)
    read_data(); // Access shared data

    // Exit Section
    wait(mutex);          // Lock to protect readcount
    readcount--;          // Decrement active readers count
    if (readcount == 0) { // If this is the last reader, release rw lock
      signal(rw);         // Unblocks waiting writers
    }
    signal(mutex);        // Unlock mutex for readcount

    // Remainder Section
    // ...
  }
}
```

Readers–Writers Problem: Writer Code (Reader-Priority) I

Writer Process/Thread Code:

```
// Writer process/thread
void Writer() {
  while (true) {
    // Entry Section
    wait(rw); // Acquire exclusive lock on the shared resource
              // Blocks if readers are active (readcount > 0)
              // or another writer is active.

    // Writing Section (only one writer allowed here)
    write_data(); // Modify shared data

    // Exit Section
    signal(rw); // Release exclusive lock on the shared resource

    // Remainder Section
    // ...
  }
}
```

Problem with Reader-Priority Solution:

- **Writer Starvation:** If readers continuously arrive, ‘readcount‘ may never drop to 0, meaning ‘rw‘ is never ‘signal()‘ed, and a waiting writer may **never get a chance to write**.

Readers–Writers Problem: Writer Code (Reader-Priority) II

- This is a common issue when prioritizing one class of processes over another.

Towards a Writer-Priority Solution:

- A writer-priority solution would typically involve additional semaphores and a ‘writecount‘ or ‘read_block‘ semaphore.
- When a writer wants to enter, it would block new readers from starting, allowing waiting writers to proceed before any new readers.
- This often involves more complex logic to manage queues for both readers and writers and ensure fairness or strict priority.

This illustrates the trade-off in synchronization: prioritizing one group can lead to starvation for another.

Bounded-Buffer Problem (Producer-Consumer Review) I

Recall from Last Week:

- A producer creates items and puts them into a shared, fixed-size buffer.
- A consumer takes items from the buffer and consumes them.
- This is a fundamental pattern in concurrent systems (e.g., message queues, print spoolers).

Synchronization Needs:

1. **Mutual Exclusion:** Only one process (producer or consumer) can access the buffer at a time to modify its contents or pointers.
2. **Synchronization (Coordination):**
    - Producer must wait if the buffer is full.
    - Consumer must wait if the buffer is empty.

Semaphore-Based Solution (Recap):

- ‘semaphore mutex $= 1$;‘ (for mutual exclusion on buffer access)

Bounded-Buffer Problem (Producer-Consumer Review) II

- ‘semaphore empty = N;‘ (counting semaphore, initially ‘N‘, counts empty slots)
- ‘semaphore full = 0;‘ (counting semaphore, initially ‘0‘, counts full slots)

Producer Logic	**Consumer Logic**
`wait(empty);` `wait(mutex);` `add_item_to_buffer();` `signal(mutex);` `signal(full);`	`wait(full);` `wait(mutex);` `remove_item_from_buffer();` `signal(mutex);` `signal(empty);`

This problem is crucial as its solution pattern forms the basis for many real-world message queueing systems.

Mutexes vs. Semaphores: Choosing the Right Tool I

While a binary semaphore **can** be used for mutual exclusion, dedicated mutex objects offer additional semantic clarity and features in most modern programming environments.

Feature	**Binary Semaphore**	**Mutex (Mutual Exclusion Lock)**
Type	Integer counter (0 or 1)	Binary state (locked/unlocked)
Purpose	General-purpose signaling	Strictly for mutual exclusion
Initial Value	0 or 1	Unlocked (ready to be acquired)
Ownership	No concept of ownership. Any process can ‘signal()‘.	**Owned** by the thread that acquired it. Only the owner can release it.
Operations	‘wait()‘ (P), ‘signal()‘ (V)	‘lock()‘, ‘unlock()‘
Use Case	Resource counting, signaling events, Producer-Consumer (full/empty)	Protecting critical sections, ensuring atomicity of operations
Complexity	Lower-level primitive	Higher-level primitive built on top of binary semaphores or atomic ops
Recursion	No inherent support	Often supports recursion (a thread can acquire its own mutex multiple times)
Priority Inversion Safety	No inherent protection	Many implementations have priority inheritance/ceiling protocols

Mutexes vs. Semaphores: Choosing the Right Tool II

When to Use Which:

- Use a **Mutex** when you need to protect a critical section where only one thread can execute at a time (e.g., shared variable updates). It's simpler and safer due to ownership.
- Use a **Counting Semaphore** when you need to control access to a pool of resources (e.g., 'N' printers) or for general signaling (e.g., producer signaling consumer that data is ready).
- A **Binary Semaphore can** be used as a mutex, but usually, a dedicated Mutex object is preferred for its features and clearer semantics.

Key Takeaways I

- **Classical Synchronization Problems** (Dining Philosophers, Readers-Writers, Bounded Buffer) serve as benchmarks for understanding and solving complex concurrency challenges like deadlock and starvation.
- **Dining Philosophers** highlights how resource acquisition order and circular dependencies can lead to deadlock. Solutions involve breaking one of the deadlock conditions.
- **Readers-Writers** demonstrates the need for nuanced access control (multiple readers vs. single writer) and the trade-offs in prioritizing one group, which can lead to starvation for another.
- **Bounded Buffer** is a foundational pattern for inter-process/thread communication, requiring both mutual exclusion and coordination.
- **Mutexes** are specifically for mutual exclusion (ownership, RAII-friendly), while **Semaphores** are more general for counting resources or signaling. Choose based on specific needs.

Key Takeaways II

- Real-world operating systems and libraries (e.g., POSIX, C++ Standard Library) provide high-level, robust synchronization primitives (mutexes, semaphores, condition variables, RW locks) that abstract away low-level complexities.

Reflection Prompt: Beyond Theory

Consider an online multiplayer game. How might the "Readers-Writers" problem manifest when players update their character's stats (write) or view other players' stats (read)? What are the practical implications of reader-priority vs. writer-priority in such a system?

Next Week Preview: Deadlocks: Detection, Avoidance, Prevention, Recovery

A Deeper Dive into Deadlocks

- **Deadlock Characterization:** The four necessary conditions revisited.
- **Deadlock Prevention:** Strategies to ensure at least one condition never holds.
- **Deadlock Avoidance:** Algorithms (e.g., Banker's Algorithm) to dynamically decide if granting a resource request is safe.
- **Deadlock Detection and Recovery:** Identifying deadlocks once they occur and strategies to break them.
- **Livelock and Starvation** (briefly differentiate from deadlock).

Next Week Preview: Deadlocks: Detection, Avoidance, Prevention, Recovery

A Deeper Dive into Deadlocks

- **Deadlock Characterization:** The four necessary conditions revisited.
- **Deadlock Prevention:** Strategies to ensure at least one condition never holds.
- **Deadlock Avoidance:** Algorithms (e.g., Banker's Algorithm) to dynamically decide if granting a resource request is safe.
- **Deadlock Detection and Recovery:** Identifying deadlocks once they occur and strategies to break them.
- **Livelock and Starvation** (briefly differentiate from deadlock).

Prep Tip for Next Session

Review the four necessary conditions for deadlock from a previous session. Understanding these deeply will be crucial for next week's topic, as all prevention and avoidance strategies target one or more of these conditions.

Outline

Synchronization Primitives in Real-World OS/Library APIs I

POSIX Threads (pthreads) - C/C++

- **Mutexes:** ‘pthread_mutex_t‘, ‘pthread_mutex_init()‘, ‘pthread_mutex_lock()‘, ‘pthread_mutex_unlock()‘. The most common way to achieve mutual exclusion.
- **Semaphores:** ‘sem_t‘, ‘sem_init()‘, ‘sem_wait()‘, ‘sem_post()‘. Can be named (process-shared) or unnamed (thread-shared).
- **Condition Variables:** ‘pthread_cond_t‘, ‘pthread_cond_init()‘, ‘pthread_cond_wait()‘, ‘pthread_cond_signal()‘, ‘pthread_cond_broadcast()‘. Used with mutexes to wait for specific conditions to become true (e.g., buffer not empty/full).
- **Read-Write Locks:** ‘pthread_rwlock_t‘, ‘pthread_rwlock_rdlock()‘, ‘pthread_rwlock_wrlock()‘, ‘pthread_rwlock_unlock()‘. Built-in support for Readers-Writers problem.

Synchronization Primitives in Real-World OS/Library APIs II

C++ Standard Library (C++11 onwards)

- **Mutexes:** ‘std::mutex‘, ‘std::recursive_mutex‘, ‘std::timed_mutex‘. Provide ‘lock()‘ and ‘unlock()‘ methods.
- **Lock Guards/Unique Locks:** ‘std::lock_guard‘, ‘std::unique_lock‘. RAII (Resource Acquisition Is Initialization) wrappers around mutexes to ensure ‘unlock()‘ is called automatically, preventing common errors.
- **Condition Variables:** ‘std::condition_variable‘. Used with ‘std::unique_lock‘ to wait for conditions.
- **Semaphores (C++20):** ‘std::counting_semaphore‘, ‘std::binary_semaphore‘. Native support added recently.

Synchronization Primitives in Real-World OS/Library APIs III

Linux Kernel Synchronization Primitives

- **Spinlocks:** ‘spinlock_t‘. Used for very short critical sections, busy-waits.
- **Mutexes:** ‘struct mutex‘. Blocking, suitable for longer critical sections.
- **Semaphores:** ‘struct semaphore‘. Can be used as counting or binary.
- **Completions:** For threads to wait for specific events.
- **Read-Write Semaphores/Spinlocks:** For Readers-Writers pattern.

Quick Questions: Classical Problems & Synchronization Tools

Test Your Understanding:

1. **Dining Philosophers:** Describe the **deadlock scenario** in the simple semaphore solution for the Dining Philosophers Problem. Which specific deadlock condition is violated?
2. **Readers-Writers:** Explain why the Reader-Priority solution to the Readers-Writers Problem can lead to **writer starvation**. How would a writer-priority solution typically differ to address this?
3. **Mutex vs. Semaphore:** You need to protect a shared linked list from concurrent modifications. Which synchronization primitive would you primarily choose: a `mutex` or a `counting semaphore`? Justify your choice.
4. **Bounded Buffer (Producer-Consumer):** If a producer attempts to add an item to a full buffer, and a consumer attempts to remove an item from an empty buffer, what role do the ‘empty‘ and ‘full‘ semaphores play in ensuring correct behavior without busy-waiting?

Exercise: Applying Readers-Writers & Mutex/Semaphore Choice I

Part 1: The Ticketing System Problem
Imagine a shared online ticketing system for a concert with 100 available tickets. Multiple users (threads/processes) can:

- **View available tickets** (read operation). Many users can do this concurrently.
- **Purchase a ticket** (write operation). This reduces the available tickets and must be exclusive.

Task:

1. Describe how this problem maps to the Readers-Writers problem. Identify the "readers" and "writers".
2. Outline a synchronization solution for this ticketing system using semaphores (or mutexes where appropriate) that **prioritizes buyers (writers)** to ensure that waiting buyers get tickets before new viewers can read. Describe the variables you'd use and the high-level logic for ‘view_tickets()‘ and ‘buy_ticket()‘.

Exercise: Applying Readers-Writers & Mutex/Semaphore Choice II

3. What potential starvation issue could arise from your writer-priority solution for this problem?

Part 2: Choosing the Right Primitive
For each scenario below, decide whether a **mutex** or a **counting semaphore** would be the most appropriate primary synchronization primitive, and briefly explain why.

1. Protecting a linked list that stores active network connections, allowing only one thread to add or remove connections at a time.
2. Controlling access to a shared pool of 8 database connections, ensuring no more than 8 threads can use a connection simultaneously.
3. Signaling between two threads: Thread A produces a data packet, and Thread B needs to process it. Thread B should wait until a packet is available.

Exercise: Applying Readers-Writers & Mutex/Semaphore Choice III

Reminder

Think about the fundamental purpose and properties of each synchronization primitive. Consider not just correctness, but also practicality and common usage patterns.

Appendix: Advanced Topics to Explore I

Deepening Your Understanding of Advanced Synchronization

I. Beyond Semaphores & Mutexes

- **Monitors:** High-level programming language constructs (e.g., in Java, C#) that encapsulate shared data and the synchronization primitives (mutexes and condition variables) needed to protect them.
- **Condition Variables (Detailed):** How ‘wait()‘, ‘signal()‘, and ‘broadcast()‘ work in conjunction with mutexes to enable threads to wait for complex conditions beyond simple resource availability.
- **Rethinking Classical Problems with Monitors/Condition Variables:** How the Dining Philosophers or Readers-Writers problem would be solved using these higher-level constructs, often resulting in cleaner code.

Appendix: Advanced Topics to Explore II

II. Fairness and Liveness Issues

- **Priority Inversion & Priority Inheritance Protocol:** When a high-priority task gets blocked by a low-priority task holding a resource it needs, and how OSes prevent this.
- **Adaptive Mutexes:** Mutexes that start as spinlocks and transition to blocking when contention is high.
- **Readers-Writers Variants (Full Discussion):** More complex solutions that guarantee no starvation for either readers or writers (e.g., using turnstiles or separate queues).

Appendix: Advanced Topics to Explore III

III. Hardware-Software Interface

- **Memory Models (Revisited):** The specifics of how different processor architectures (e.g., x86, ARM) ensure memory visibility and order, impacting concurrent programming correctness.
- **Cache Coherency Protocols:** How multiple CPU cores maintain a consistent view of shared memory in their private caches.
- **Atomic Operations in Detail:** Deep dive into how ‘compare_exchange_strong‘, ‘fetch_add‘, etc., are implemented at the hardware level and their role in lock-free programming.