Synchronization Strategies

> "Chaos is natural. Order is optional — unless you enforce it."

- Multiple threads want the same resource.
- Without discipline, **race conditions**, **deadlocks**, and **starvation** emerge.
- Synchronization ensures fairness, correctness, and sanity.

⚙ The Critical Section Problem

A **critical section** is a piece of code that accesses shared resources.

Goal:

Only one thread executes it at a time.

```
// Critical section (shared counter)
void increment() {
    counter++; // Only one thread allowed here
}
```

Requirements:

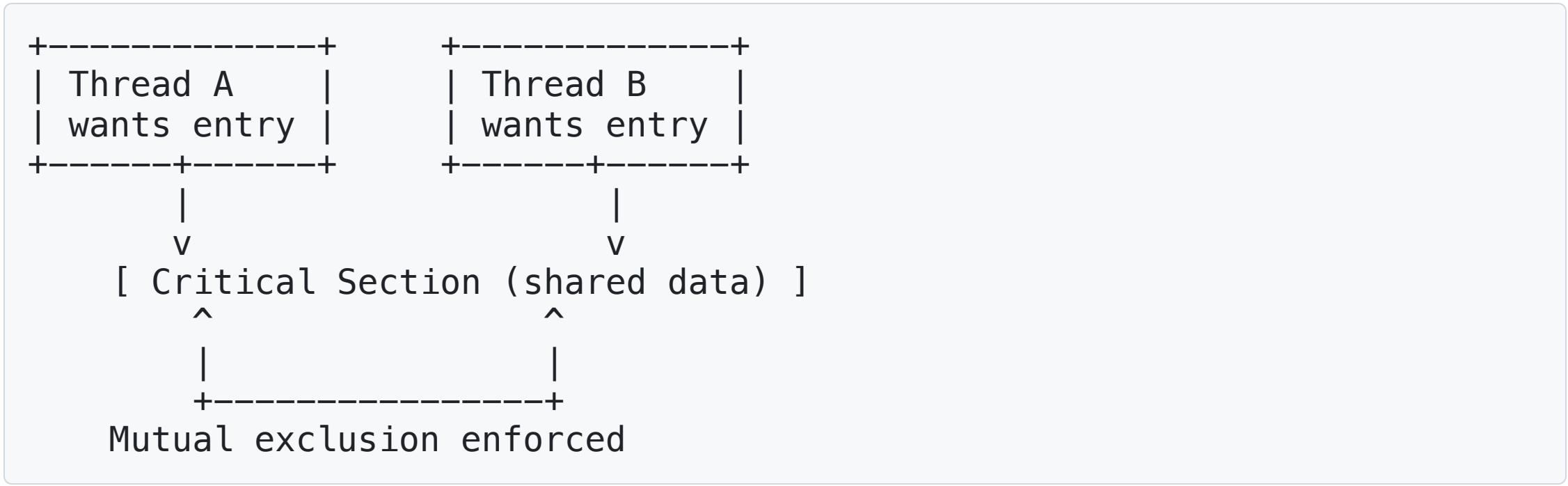
1. **Mutual exclusion** — only one thread inside.
2. **Progress** — someone gets in eventually.
3. **Bounded waiting** — nobody waits forever.

Synchronization Tools

Tool	Description	Example Use
Mutex (Lock)	One thread at a time.	Protecting shared variables.
Semaphore	Counter-based — allows *n* simultaneous threads.	Producer/consumer buffers.
Monitor	Combines lock + condition variables.	Java `synchronized` methods.
Atomic Ops	Hardware-level single-step ops.	`std::atomic<int> counter++`

> "Locks are like traffic lights — ignore them and everything crashes."

Visual: The Critical Section



🧮 Example: The Bank Account Problem

```
void deposit(int amount) {
    balance += amount;
}

void withdraw(int amount) {
    if (balance >= amount)
        balance:-= amount;
}
```

- Without a lock, `balance` can go negative.
- With a **mutex**, it stays consistent:

```
std::lock_guard<std::mutex> guard(mtx);
balance += amount;
```

⚔️ Deadlock

> “A situation where each thread waits forever for the other to release a resource.”

Classic Example:

```
// Thread A
lock(mutex1);
lock(mutex2);

// Thread B
lock(mutex2);
lock(mutex1);
```

💥 Both are now waiting — forever.

🧩 Four Necessary Conditions for Deadlock

Condition	Meaning
Mutual exclusion	Resource held by one process at a time.
Hold and wait	Process holds one, requests another.
No preemption	Resource can’t be forcibly taken away.
Circular wait	Circular chain of waiting processes.

Break any one of these → prevent deadlock.

Deadlock Handling Strategies

Strategy	Idea	Example
Prevention	Design system so at least one condition can’t happen.	Always lock resources in same order.
Avoidance	Detect unsafe states before allocating.	Banker’s Algorithm.
Detection/Recovery	Let it happen, then fix it.	Timeout or kill one process.

⏳ Starvation

> “When a thread waits indefinitely because others keep getting served first.”

Example:

A low-priority thread never gets CPU because higher-priority threads dominate.

Fixes:

- Aging (gradually increase waiting thread’s priority).
- Fair locks (FIFO order).
- Balanced scheduling policies.

🧭 Ordering and Priority Inversion

Even with locks, bad ordering causes chaos.

Priority inversion example:

- Low-priority thread holds a lock.
- High-priority thread waits.
- Medium-priority thread runs endlessly.

💀 High-priority thread starves until low-priority one releases.

Real case:

NASA's *Mars Pathfinder* (1997) reset mid-mission due to priority inversion bug.

🧮 Illustration: Ordering & Hierarchy

```
Resources: R1, R2, R3

Threads must always lock in increasing order:
  lock(R1) → lock(R2) → lock(R3)

If everyone follows this order → no circular wait.
```

Think of it as *“The Locking Chain of Command.”*

🧠 Summary

Concept	Goal	Example Fix
Critical Section	Protect shared resources.	Mutex / atomic ops.
Deadlock	Avoid permanent waiting.	Order locks consistently.
Starvation	Ensure fairness.	Aging / fair locks.
Ordering	Prevent circular waits.	Enforce lock hierarchy.

> “Synchronization isn’t about control — it’s about coordination.”

🎓 Next: The OS as Referee

> “Who gets the CPU, who gets the disk, who gets the lock.”

Next section:

- Scheduling Algorithms (Round-Robin, Priority, Multilevel Queue)
- Context switching
- Cooperative vs Preemptive systems