Introduction

Hello everyone! Today, we'll explore a real-world concurrency problem through a simple **Voting Service**.
Understanding concurrency early is crucial because **race conditions** and **thread safety** issues can sneak into production code unexpectedly.

The Voting Service Problem

We have a web service where users vote for proposals. For example:

- Users submit votes.
- The server increments the `voteCount`.
- We expect the final count to reflect all votes accurately, but something goes wrong.

Here's what users see across browsers. The vote count seems inconsistent. **Why is this happening?**

The Code Issue

The `incrementVoteCount()` method looks simple.
But what happens if **two threads** execute this at the same time?

When multiple threads access `voteCount` simultaneously:

1. Both threads read the **same value**.
2. They increment separately.
3. One increment overwrites the other.
   Result: Lost votes!

The Root Cause

The increment operation consists of **three separate steps**:

- Get, Add, and Put.

This is problematic because **two CPU cores** can execute these operations **in parallel**.
For example:

- **CPU 1** might perform the "add" operation but not complete the "put."
- **CPU 2** simultaneously executes the "get" operation on the same memory address.

As a result, one update can overwrite the other, leading to inconsistencies.

Testing the Problem

Our test confirms the issue: **Expected 1000 votes**, but **only 985 are recorded**.

Duplicate logs also reveal **suspicious counts**, where two threads update the same value simultaneously.

Detecting Race Conditions

To detect race conditions, we:

1. **Parse each log line** to extract the "Before count: X" and "After count: Y" values.
2. **Keep track** of all the "After" values along with the lines where they occurred.
3. If the **same "After" value** appears more than once, we print those lines as **suspicious** to highlight potential issues.

Solution: Thread Synchronization

The `synchronized` keyword ensures only **one thread** can access `incrementVoteCount()` at a time.

```
public synchronized void incrementVoteCount() {
    int before = voteCount;
    voteCount++;
    int after = voteCount;

    logger.info("Before count: " + before + " After count: " + after);
}
```

Thread Management: Scheduler

Let's talk about how threads are managed in a program.

At the top, we have a **Java process**. A process is a running instance of a program, and within it, we have **threads**.
Threads are lightweight units of execution that **share the same memory**.

When multiple threads are running, the **OS Thread Scheduler** decides which thread gets access to the CPU at any given moment.

Key Points:

- **Shared Memory**: All threads within the process can access the same variables and data structures.
- **Thread Execution**: The OS scheduler maps these threads to different **CPU cores**.
  For example, Thread 1 may execute on **CPU 1**, while Thread 2 executes on **CPU 2**.

The scheduler ensures fairness and efficiency when multiple threads compete for CPU time.
This is why **thread management** is crucial for performance and avoiding conflicts.

Thread Lifecycle: States

Now, let's look at the **lifecycle of a thread**. A thread can be in one of six states:

State	Description
NEW	The thread is created but hasn't started running yet.
RUNNABLE	The thread is ready to run or actively executing on the CPU.
BLOCKED	The thread is waiting for a monitor lock to access a resource.
WAITING	The thread is waiting indefinitely for another thread to notify it.
TIMED_WAITING	The thread is waiting for a specific period (e.g., `sleep()` or `wait(timeout)`).
TERMINATED	The thread has finished executing or has been stopped.

Understanding **thread states** helps us debug and manage thread behavior.
For example, if a thread is stuck in **BLOCKED** or **WAITING** states, it might indicate a resource contention problem.

Compare-And-Swap (CAS): Thread Safety without Locks

The **Compare-And-Swap (CAS)** mechanism ensures **thread safety** by performing atomic updates **without explicit locks**.

How It Works:

1. CAS compares the **current value** of a variable to an **expected value**.
2. If the two values match, the variable is **swapped** with a **new value**.
3. If they don't match, the operation fails, and the old value is retained.

In Java, `AtomicInteger` uses CAS to implement methods like `incrementAndGet()`.
This allows the increment operation to be performed **atomically**, even in a multi-threaded environment, without locks.

Comparing Approaches: Locks vs. CAS

Top Diagram: Lock-Based Approach

- **Thread 1** attempts to access the shared data structure but is **blocked** because another thread holds the lock.
- While Thread 1 is blocked, it **wastes time** waiting for the lock to be released.
- This adds overhead due to **context switching** and thread scheduling.

Bottom Diagram: CAS-Based Approach

- CAS uses **hardware-level atomic operations** to update the variable.
- **Thread 1** performs the operation **without being blocked**.
- CAS repeatedly checks and updates the shared variable in a **non-blocking manner**.

Why CAS Is More Efficient:

- **No Blocking**: Threads don't waste time waiting for locks.
- **Fewer Context Switches**: The scheduler avoids managing paused and resumed threads.
- **Hardware Optimization**: CAS is optimized by hardware, making it much faster.

Concurrency in Modern Applications

1. **IV. Backing Services**:
   Treat backing services, such as databases, message queues, or caches, as **attached resources**. In our case, the **vote count** should not rely on in-memory storage, which is prone to race conditions and data loss. Instead, votes should be **persisted in a database** or external backing service to ensure reliability and consistency.

2. **VI. Stateless Processes**:
   Design processes to be **stateless** and **share nothing**. A stateless process does not store any state (e.g., vote counts) in its memory. Instead, all state should be managed by backing services, such as a database or external storage. Stateless processes can scale horizontally without conflicts, ensuring scalability and reliability.

3. **VII. Concurrency**:
   Processes should scale **horizontally** by running multiple instances that do not share state. This can be achieved by partitioning workloads across processes to prevent contention. For example, the vote-counting process can be **horizontally partitioned** so that each instance handles a specific subset of votes independently, avoiding race conditions.

Why It Matters

By adhering to these principles:

- We avoid **shared state** across threads or processes.
- Votes are consistently counted in a **backing service** like a database.
- Processes are **stateless** and can scale horizontally, ensuring that concurrency is reliable and efficient.

This approach aligns with the **12-Factor App** methodology, making the system robust, scalable, and ready for modern cloud environments.

Key Takeaways

1. Race conditions can occur in simple operations like `voteCount++`.
2. Use **synchronized** methods or atomic classes like `AtomicInteger` to ensure **thread safety**.
3. CAS-based approaches are more efficient because they avoid blocking and reduce scheduler overhead.
4. Concurrency must be **reliable** and **efficient** in modern applications.

Conclusion

Thank you for your time! Concurrency is challenging, but understanding these basics helps you prevent race conditions and thread safety issues early.