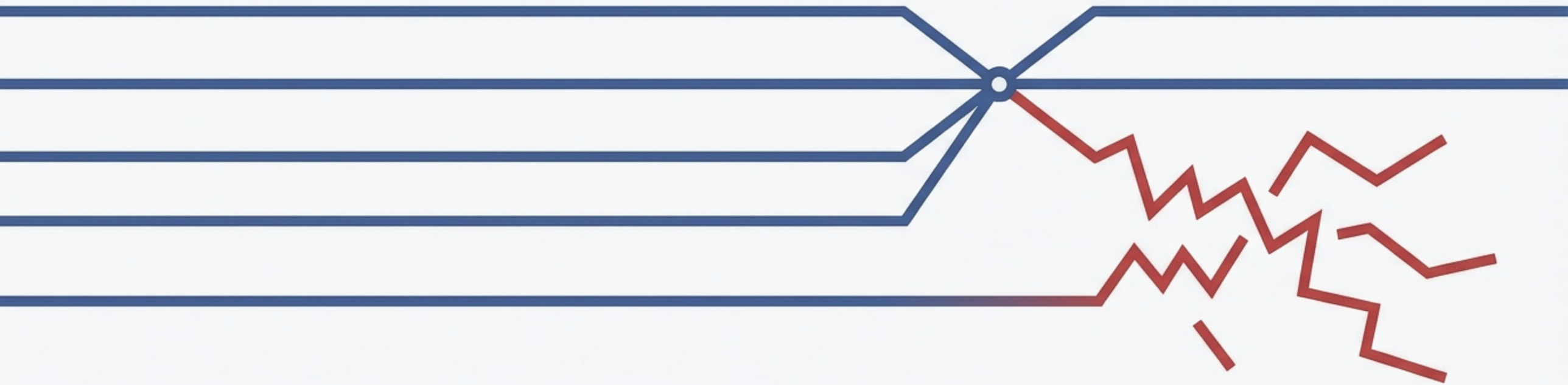
The Peril of Parallelism: Understanding Shared Data in Concurrent Code

Dr. Talgat Turanbekuly

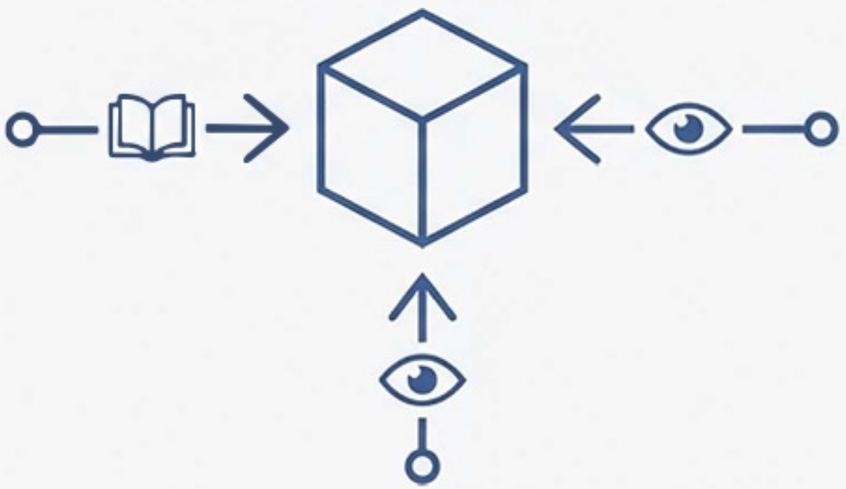
Lecture-3-1



The One Core Problem: Modifying Shared Data

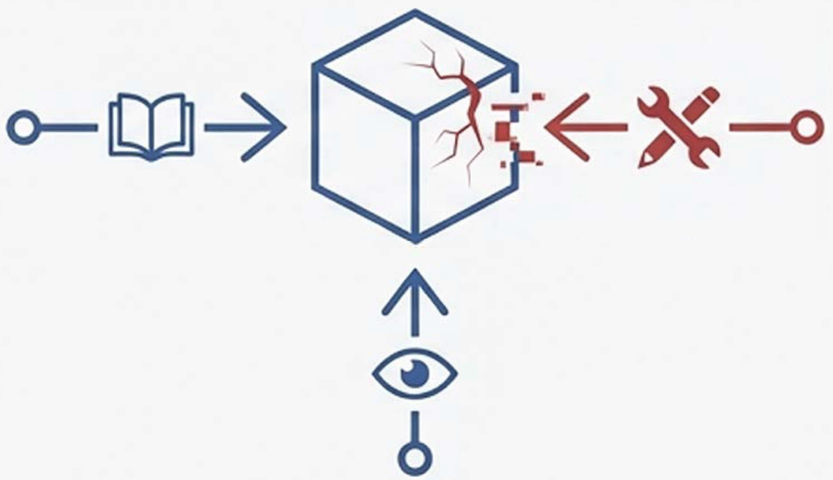
If all shared data is read-only:

There is no problem. A thread's view of the data is unaffected by other threads reading it simultaneously.



If one or more threads modify shared data:

There is immense potential for trouble. You must actively engineer for safety.



The act of changing data that other threads can see is where our focus must be.

Invariants: The Rules That Keep Data Structures Sane

Definition

An invariant is a statement that is always true about a particular data structure. It's a promise about its state.

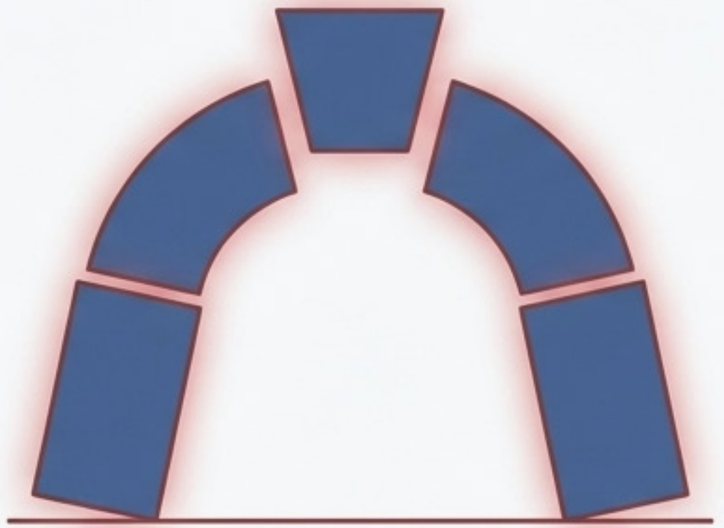
Examples

- "This variable contains the number of items in the list."
- "This pointer is never null."

The Critical Vulnerability

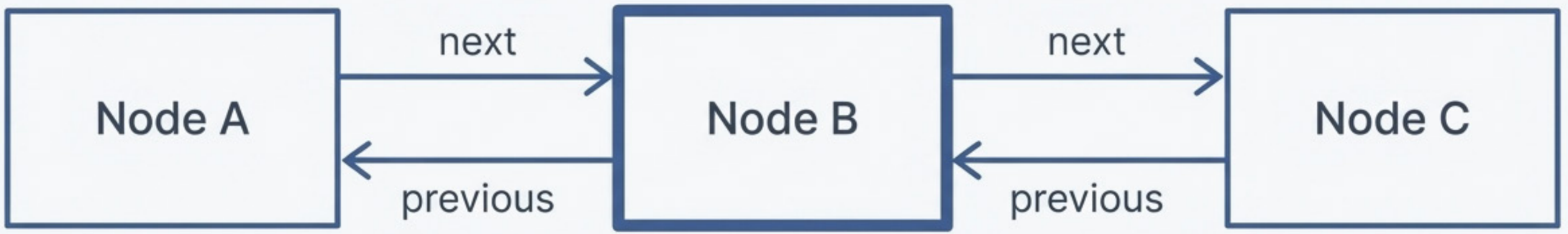
Invariants are often broken during an update, especially in complex operations that require modifying more than one value. This temporary broken state is a window of danger.

INVARIANT HOLDS



INVARIANT TEMPORARILY BROKEN

Case Study: The Doubly Linked List



The Key Invariant

‘If you follow a `next` pointer from one node (A) to another (B), the `previous` pointer from that node (B) must point back to the first node (A).’

The Scenario

We will analyze the steps required to delete the middle node from this list.

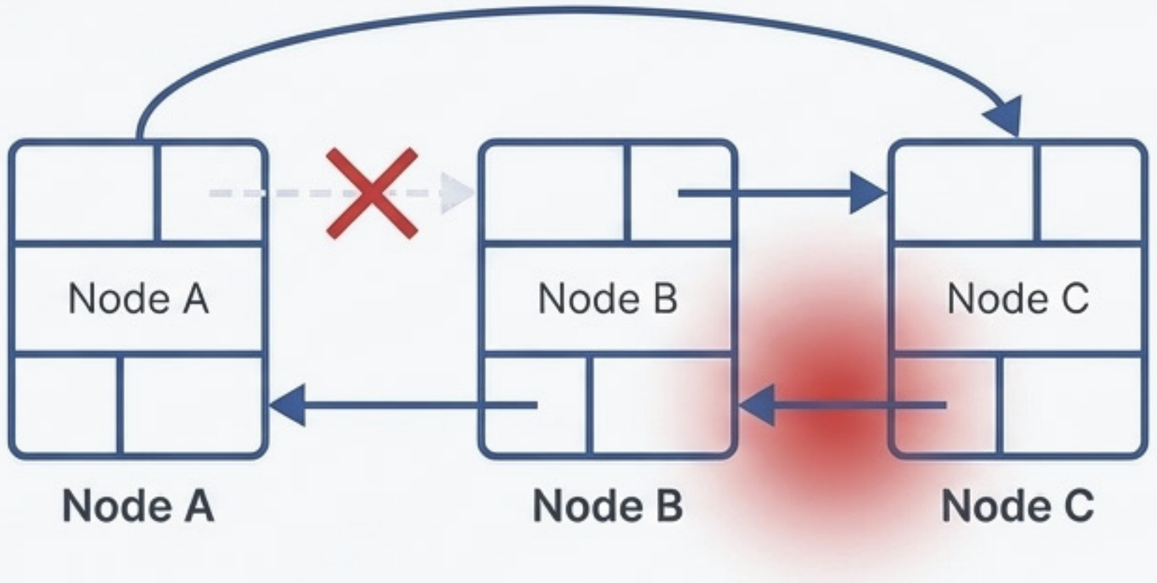
Anatomy of a Deletion: The Invariant Breaks

Step 1: The First Link is Changed

The process to delete node ‘B’:

a) Identify node to delete: B.

b) Update the `next` pointer from the node *before* B (Node A) to point to the node *after* B (Node C).



The invariant is now BROKEN. The links are inconsistent.

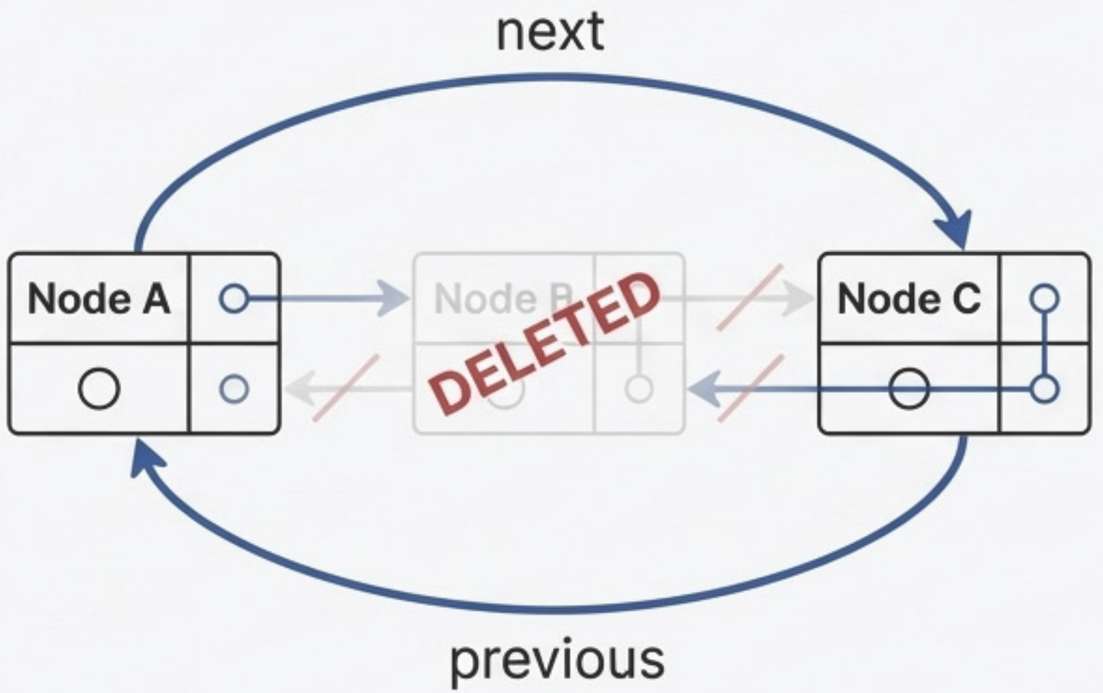
Anatomy of a Deletion: Restoring the Invariant

Step 2: The Operation Completes

c) Update the `previous` pointer from the node *after* B (Node C) to point to the node *before* B (Node A).

d) Delete node B.

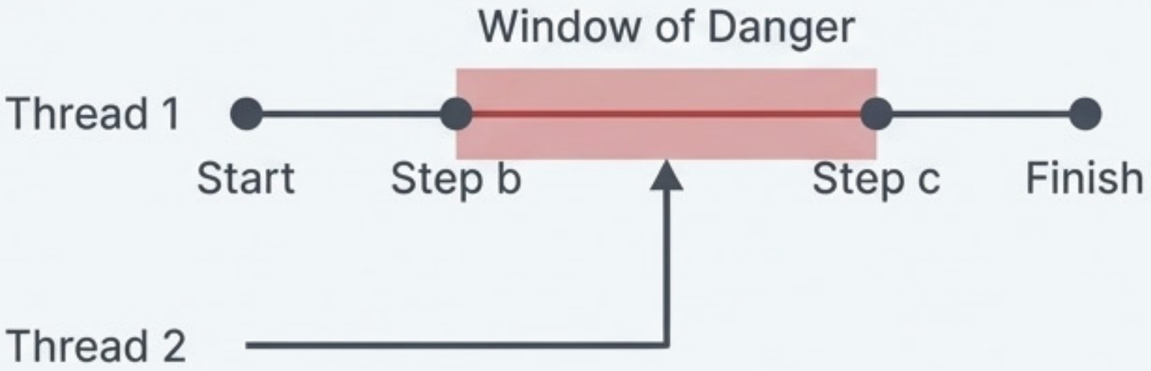
The invariant now HOLDS again. But what could have happened in between?



The Window of Opportunity for Bugs

The Core Question: What if Thread 2 tries to read the list while Thread 1 is between steps (b) and (c)?

The Answer: Thread 2 sees a corrupt data structure where the invariant is broken.



Potential Consequences

Best Case: The reading thread might simply skip over the node being deleted.

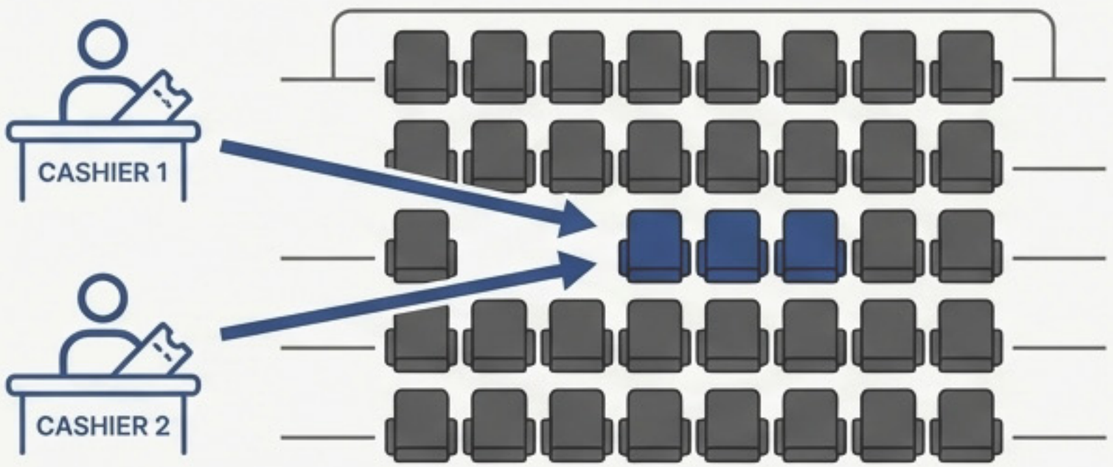
Worst Case: If the second thread is also trying to modify the list (e.g., delete another node), it could permanently corrupt the data structure, leading to memory leaks, incorrect data, and program crashes.

This is an example of a **race condition**.

Defining the Culprit: Race Condition

Analogy: Buying Movie Tickets

Imagine two people buying tickets for the same movie at the same time from different cashiers. If only a few seats are left, who gets them depends on whose transaction processes first. It's literally a race to see who gets the last tickets.



Technical Definition

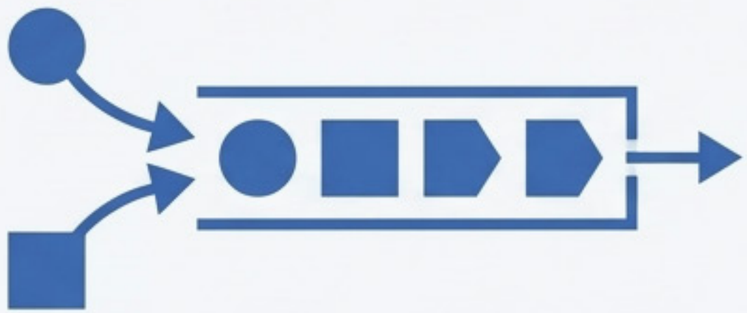
In concurrency, a `race condition` is any situation where the outcome depends on the relative ordering of execution of operations on two or more threads.

Problematic vs. Benign Race Conditions

Most of the time, race conditions are benign. The outcome changes, but all outcomes are acceptable.

Benign Race

Two threads add items to a processing queue. It generally doesn't matter which item gets added first, as long as the queue's invariants are maintained.



Problematic Race

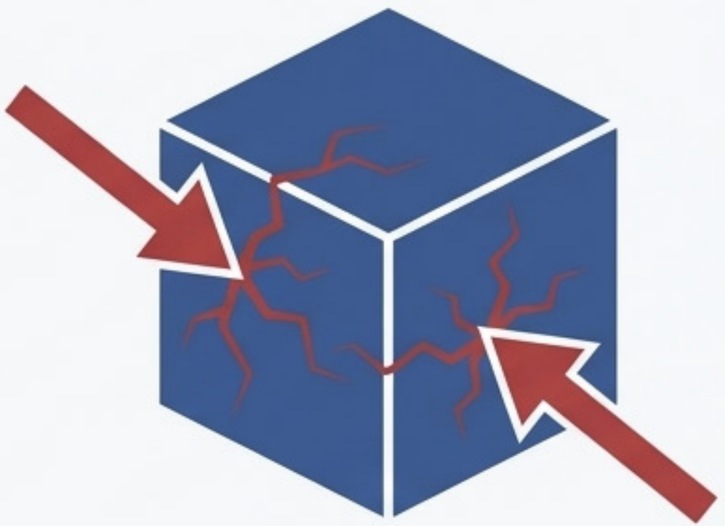
The term "race condition" in practice usually refers to a problematic one, where the race leads to broken invariants, like in our doubly linked list example. These are the bugs we need to prevent.

A Special Case: The Data Race

> **The C++ Standard defines a `data race` as:**
>
> > “The specific type of race condition that arises from concurrent modification to a single object.”

The Consequence is Severe

Data races cause **undefined behavior**. This is one of the most dreaded outcomes in C++, as the program’s state becomes completely unpredictable. It can crash, produce incorrect results, or appear to work correctly until a minor change elsewhere exposes the bug.



Why Race Conditions Are So Hard to Debug

Race conditions are the ghosts in the machine. They are incredibly difficult to find and replicate because:

They are timing-sensitive

The window of opportunity for the problematic ordering of operations can be nanoseconds long.

They depend on load

As system load increases, the chance of a problematic execution sequence occurring increases. The bug may only appear on production servers, not on a developer's machine.

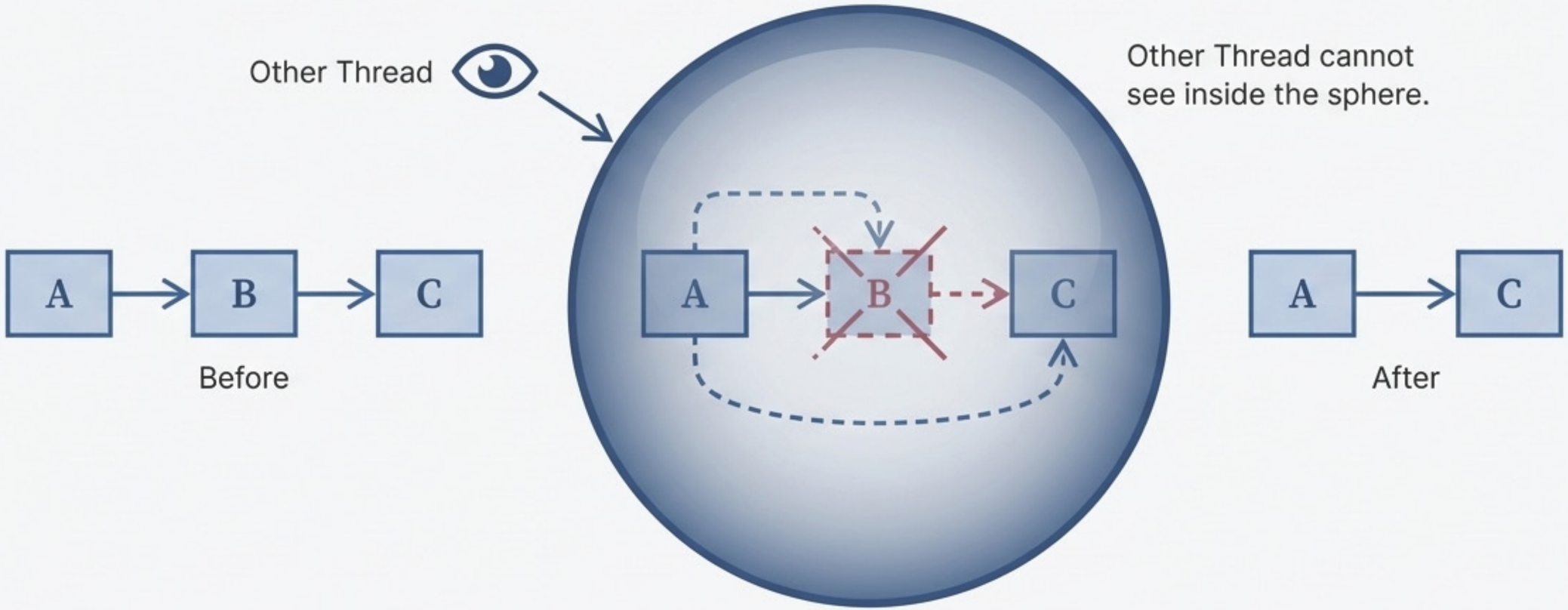
They can be "Heisenbugs"

They often disappear when the application is run under a debugger, because the act of debugging itself affects the program's timing.

A Framework for Avoiding Problematic Races

The goal is to prevent other threads from seeing the intermediate states where invariants are broken.

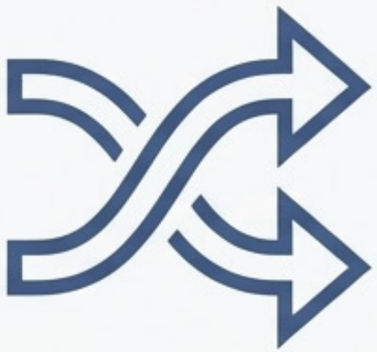
- From the point of view of an outside thread, a modification should be **atomic**: it either hasn't started, or it has completed entirely. There is no in-between.
- The C++ Standard Library provides several mechanisms to achieve this.



Three High-Level Strategies

1. Protection Mechanisms

Wrap the data structure with a protection mechanism (like a mutex) to ensure only one thread can perform a modification at a time.



2. Lock-Free Programming

Redesign the data structure and its invariants so that modifications are done as a series of indivisible (atomic) changes, each of which preserves the invariants. This is extremely difficult to get right.

3. Software Transactional Memory (STM)

Treat updates like a database transaction. Log the required changes and commit them in a single step. If another thread interferes, the transaction is rolled back and restarted. (An active area of research).

Where We Begin: The Mutex

Of these strategies, the most fundamental mechanism for protecting shared data provided by the C++ Standard is the **mutex**.

It is the primary tool for implementing the “Protection Mechanism” strategy.

Understanding how to use mutexes correctly is the first and most critical skill in writing safe concurrent code.

Next Up: A deep dive into mutexes.