You Don’t Know Jack

Shared variables in concurrent programs appear deceptively simple: multiple threads read and write the same memory. In reality, without an explicit and well-understood memory model, shared variables have no reliable semantics.

At the source-code level, shared variables look like ordinary variables accessed by multiple threads. Programmers often assume that writes by one thread will eventually be seen by others, and that operations occur in program order. These assumptions are false unless enforced by the language’s memory model. This is due to reordering. Modern compilers reorder instructions to improve performance. CPUs execute instructions out of order, speculatively, and in parallel and as a result, the actual execution order may differ substantially from the program order. Without synchronization, there is no guarantee that another thread observes operations in the order written in the source code. Each core typically maintains private caches and store buffers. A write performed by one thread may remain invisible to others for an unbounded period of time. Cache coherence ensures eventual consistency, but not ordering or timeliness. Two threads can simultaneously observe different values of the same variable. A data race occurs when two threads access the same variable concurrently and at least one access is a write, without synchronization. In languages such as C and C++, the presence of a data race renders program behavior undefined. This means the compiler is permitted to assume the race never occurs and optimize accordingly, often producing results that appear nonsensical. Atomic operations prevent partial reads and writes, but they do not automatically provide ordering or visibility guarantees. Memory orders (e.g., relaxed, acquire, release, sequentially consistent) define which operations become visible and in what order. Misusing atomics often leads to programs that are “mostly correct” until they fail under optimization, scale, or different hardware. Synchronization primitives—mutexes, condition variables, barriers, fences—do more than prevent concurrent access. They establish happens-before relationships, which are the only mechanism by which one thread’s actions become meaningfully visible to another.