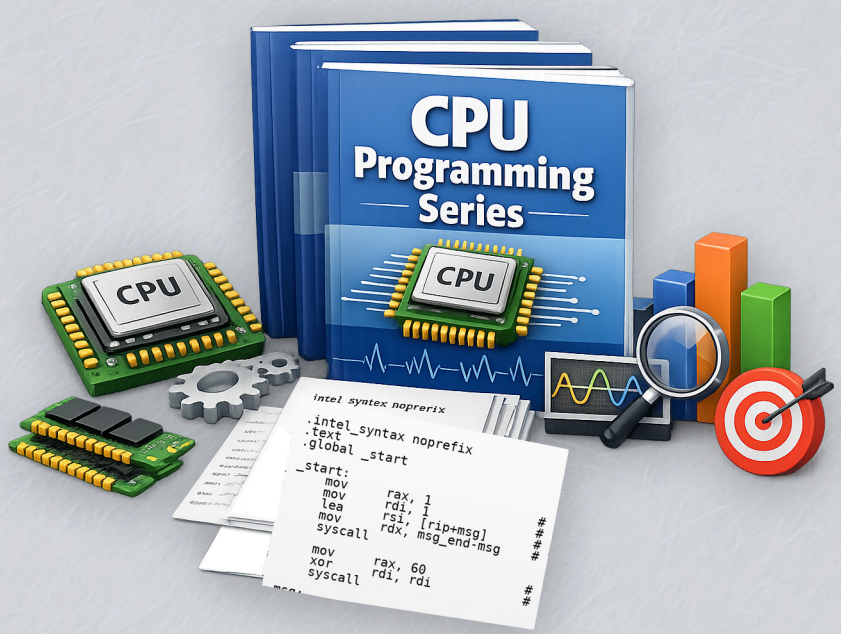
https://simplifycpp.org

CPU Programming Series

Atomics & Memory Ordering

Assembly-Level Concurrency Without Illusions



19

Prepared by Ayman Alheraki

CPU Programming Series

Atomics Memory Ordering

Assembly-Level Concurrency Without Illusions

Prepared by Ayman Alheraki

simplifycpp.org

February 2026

Contents

Preface

Motivation and Scope

This booklet exists to remove a dangerous illusion: that concurrency correctness can be achieved by "writing things in the right order" and trusting that other cores will observe that same order.
Modern CPUs and compilers are explicitly designed to:

- reorder instructions (for latency hiding and throughput),
- buffer and merge stores (for performance),
- speculate and execute out-of-order,
- and propagate writes to other cores on timelines that are not "immediate".

Atomics and memory ordering are not optional sophistication; they are the only precise way to express:

- **atomicity**: what must be indivisible,
- **visibility**: what must become observable to other cores,
- **ordering**: which reads/writes are forbidden from being observed out of order.

Scope. We focus on assembly-level concurrency invariants:

- Atomic load/store and RMW operations as implemented by real ISAs.
- Memory ordering: SC, acquire, release, relaxed (what they *forbid*).
- Fences/barriers: what they guarantee, and what they do *not* guarantee.
- Practical failure modes: reordering bugs, visibility failures, ABA, false sharing, livelock.

Non-scope. This booklet intentionally avoids:

- thread APIs and runtime abstractions,
- lock-free "recipes" without invariants,
- folklore rules not reducible to a memory-model statement.

Example: the classic publish/observe mistake

The most common wrong mental model is:

> If producer writes `data` then `flag`, consumer that sees `flag` must see the new `data`.

Without an acquire/release edge, this is not a guaranteed contract across architectures.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Producer (naive) */
mov     DWORD PTR [data], 123
mov     DWORD PTR [flag], 1
```

```
/* Consumer (naive) */
cmp     DWORD PTR [flag], 1
jne     not_ready
mov     eax, DWORD PTR [data]     /* expects 123 */
not_ready:
```

This booklet teaches how to turn the intended guarantee into a **real** guarantee by building a **synchronizes-with** edge using the minimal ordering required.

Target Architectures and Assumptions

Target architectures. The content is structured to remain correct and portable across:

- **x86-64**: relatively stronger default ordering in common cases, but still not "SC everywhere".
- **AArch64**: weak ordering; acquire/release and barriers are central to correctness.
- **RISC-V RV64**: RVWMO; fences and/or acquire/release annotations define visibility constraints.

Assumptions about the machine.

- Cache-coherent multi-core (coherence is not the same as ordering).
- Out-of-order execution, speculation, store buffers, and non-instant propagation.
- Atomic RMW operations are globally serialized *with respect to the same location*, but not free.

Assumptions about the compiler.

- The compiler may reorder independent loads/stores unless constrained.

- "Volatile" is not a synchronization primitive.
- Only atomics and fences create portable ordering contracts.

Three orders to keep separate.

- **Program order**: what you wrote.
- **Compiler order**: what optimization emitted.
- **Hardware-observed order**: what other cores are allowed to observe.

Example: acquire/release as real instruction contracts

Below are minimal assembly-level sketches showing how the *intent* maps to actual ISA mechanisms. (They are illustrative: the invariant is what matters.)

x86-64: rely on the correct primitive, not folklore A locked RMW is a strong synchronization point for the affected cache line, but it is costly.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Atomic: increment shared counter (locked RMW) */
lock add DWORD PTR [counter], 1
```

AArch64: acquire/release load/store are first-class

```
/* AArch64 shown conceptually in one place */

/* Producer: write data, then release-store flag */
str     w0, [xData]
```

```
stlr    w1, [xFlag]      /* release */

/* Consumer: acquire-load flag, then read data */
ldar    w2, [xFlag]      /* acquire */
ldr     w3, [xData]
```

RISC-V: fences define edges when plain loads/stores are used

```
/* RISC-V shown conceptually */

/* Producer: data then release-like fence then flag */
sw      t0, 0(a0)
fence   rw, w            /* prior reads/writes before following
 ↪  writes */
sw      t1, 0(a1)

/* Consumer: flag then acquire-like fence then data */
lw      t2, 0(a1)
fence   r, rw            /* prior reads before following reads/writes
 ↪  */
lw      t3, 0(a0)
```

The key takeaway: the **same algorithm** must be justified against each architecture's allowed outcomes, not against what "usually works" on one machine.

How to Read This Booklet

Read every section as a contract checklist, not as a narrative.

Step 1: start from an invariant

Every concurrent design begins with a sentence like:

> If thread B observes `flag==1`, then it must observe all writes that thread A performed before publishing `flag`.

If you cannot state the invariant, you cannot prove correctness.

Step 2: identify the publish edge and the observe edge

Correctness requires a cross-thread edge:

- **Publish edge**: a release operation (or stronger) that makes prior writes visible.
- **Observe edge**: an acquire operation (or stronger) that prevents later reads from floating earlier.

Step 3: separate compiler constraints from hardware constraints

- A compiler barrier can stop reordering in generated code.
- A hardware fence can stop visibility reordering across cores.
- Many bugs come from applying only one while needing both.

Step 4: validate with minimal litmus thinking

For each pattern, ask:

- What outcomes are allowed if ordering is **relaxed**?
- Which forbidden outcomes do **acquire/release** eliminate?
- Where would a **fence** be required on weak ISAs if you use plain loads/stores?

Example: reading a spin-wait correctly

Spin-waits are common, but correctness depends on the load semantics and the data dependency that follows.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Invariant: after observing flag==1, read data safely */

/* Spin on flag */
spin:
mov     eax, DWORD PTR [flag]
cmp     eax, 1
jne     spin

/* Now read the published data */
mov     ebx, DWORD PTR [data]
```

When you see this, do not ask “does it work on my PC?” Ask instead:

- What ordering does the **compiler** preserve for these accesses?
- What ordering does the **ISA** guarantee for these operations?
- If targeting weak ordering, what is the minimal acquire/release or fence needed to make the invariant true?

Practical reading strategy

- First pass: build the mental model (ordering, fences, RMW semantics).

- Second pass: treat the booklet as a reference for correct minimal patterns.
- Always prefer: **state invariant** $\rightarrow$ **choose primitive** $\rightarrow$ **prove forbidden outcomes**.

Chapter 1

Atomicity at the CPU Level

1.1 Atomic vs Non-Atomic Instructions

At the machine level, an operation is **atomic** only if other observers (other cores / devices) cannot observe a **torn** or **partially updated** state for that operation on that memory location. Most instructions are **not** atomic in the sense people casually assume: a single instruction can still participate in complex micro-architectural behavior (store buffers, cache coherence traffic, write combining). So we separate three ideas:

- **Single-copy atomicity (SCA)**: observers see either the old value or the new value, never a mix.
- **Indivisible RMW**: a read-modify-write acts as one indivisible operation with respect to the same location.
- **Ordering** (not in this section): whether other locations become visible before/after this operation.

Non-atomic does *not* mean "two instructions"

Even a single store instruction can be **non-atomic** in the presence of:

- misalignment,
- size not naturally supported atomically by the machine,
- crossing a cache-line boundary,
- or being compiled into multiple memory operations.

Example: a plain store is not a synchronization primitive

A regular store updates memory, but it does not establish a cross-thread ordering contract by itself.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Thread A (producer) */
mov     DWORD PTR [data], 123
mov     DWORD PTR [flag], 1

/* Thread B (consumer, naive) */
cmp     DWORD PTR [flag], 1
jne     not_ready
mov     eax, DWORD PTR [data]
not_ready:
```

This code may appear to work often, but the only thing it *obviously* guarantees is that each individual aligned 32-bit store is written. It does *not* define the required ordering/visibility contract for `data` relative to `flag` on all machines and toolchains.

Example: atomic RMW has different semantics than a load+store pair

A non-atomic increment is a read followed by a write, and can lose updates under contention. An atomic RMW makes the modification indivisible with respect to the same location.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Non-atomic increment (racy under contention) */
mov     eax, DWORD PTR [counter]
add     eax, 1
mov     DWORD PTR [counter], eax

/* Atomic increment (indivisible for 'counter') */
lock add DWORD PTR [counter], 1
```

The key distinction is not “one instruction vs three”: it is whether other cores can interleave their updates between your read and your write.

1.2 Indivisibility Guarantees

Indivisible means: for a given memory location, observers cannot see intermediate states of the operation. This guarantee depends on:

- **width**: 8/16/32/64-bit are commonly atomic when aligned on mainstream 64-bit systems,
- **alignment**: misalignment can turn a single access into multiple micro-operations,
- **boundary crossing**: crossing cache-line boundaries is the classic atomicity killer,
- **operation kind**: plain load/store vs locked/atomic RMW.

Example: torn reads/writes (conceptual)

A "torn" observation occurs when a reader sees half old bits and half new bits (or some mixed state). This is easiest to trigger conceptually with:

- misaligned wide stores,
- or wide stores spanning cache lines.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Conceptual example: misaligned 64-bit store can be problematic */
lea     rdi, [buf + 1]              /* deliberately misalign address
 ↪  */
mov     QWORD PTR [rdi], 0x1122334455667788

/* Concurrent reader might observe a mixed value on
 ↪  machines/conditions where this is not atomic */
mov     rax, QWORD PTR [rdi]
```

Engineering rule: if you need atomicity guarantees, ensure:

- natural alignment for the width,
- no cache-line crossing for the width,
- and use an atomic primitive when doing RMW.

Example: CAS loop is indivisible per attempt, not "atomic algorithm" by magic

Compare-and-swap makes one update attempt indivisible for the location, but the *algorithm* can still spin, starve, or require additional ordering rules.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Atomic: try to set *ptr = new if *ptr == expected */
mov     rax, QWORD PTR [expected]    /* rax = expected (cmpxchg
 ↪  compares against rax) */
mov     rbx, QWORD PTR [newval]      /* rbx = new value */
lock cmpxchg QWORD PTR [ptr], rbx    /* if [ptr]==rax then [ptr]=rbx
 ↪  else rax=[ptr] */
```

The single `cmpxchg` attempt is indivisible for `[ptr]` with respect to other cores modifying `[ptr]`. But correctness still depends on how you use the result and what ordering you need for adjacent data.

Example: LL/SC attempt is indivisible if SC succeeds (conceptual)

Load-linked / store-conditional works as an atomic update attempt: if any interfering write occurs, SC fails.

```
/* Conceptual LL/SC pattern (ISA-specific mnemonics differ) */

/* old = LL(ptr) */
 /* ... */

/* new = old + 1 */
 /* ... */

/* if (!SC(ptr, new)) retry */
 /* ... */
```

Important: LL/SC can fail for reasons other than true contention (implementation-defined), so forward progress requires careful design.

1.3 Multi-Core Atomicity

Multi-core atomicity is about what **other cores can observe** when many agents access the same locations. Two facts dominate reality:

- **Coherence is per cache line.** It ensures a single location has a consistent order of writes as seen by all cores.
- **Atomicity is per location per operation.** A write to one location does not automatically order or publish writes to other locations.

Example: atomicity on one location does not protect adjacent locations

Even if updates to `flag` are atomic, the visibility/order of `data` relative to `flag` is a separate contract.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Thread A */
mov     DWORD PTR [data], 777
mov     DWORD PTR [flag], 1          /* flag update is atomic as a
 ↪  store, but ordering is a separate issue */

/* Thread B */
mov     eax, DWORD PTR [flag]
cmp     eax, 1
jne     not_ready
mov     ebx, DWORD PTR [data]
not_ready:
```

To make **flag** serve as a publish signal for **data**, you need explicit ordering (acquire/release or equivalent), which is handled in later chapters.

Example: atomic RMW serializes updates to the same location

When multiple cores increment the same counter, atomic RMW creates a single global modification order for that location.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Core 0 */
lock add DWORD PTR [counter], 1

/* Core 1 */
lock add DWORD PTR [counter], 1

/* All cores observe counter updates as a single serial order for
 ↪  [counter] */
```

This does *not* imply the code scales well: the cache line containing `counter` will bounce between cores, and throughput collapses under contention (performance is discussed later).

Example: per-location atomicity does not give multi-location atomicity

There is no "atomic update of two independent memory locations" using plain atomics. If you update two fields, other cores can observe them in different interleavings unless you design a protocol.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix
```

```
/* Thread A: updates two locations */
mov     DWORD PTR [x], 1
mov     DWORD PTR [y], 1

/* Thread B might observe: x==1 and y==0 (or vice versa) depending on
 ↪  timing/propagation */
mov     eax, DWORD PTR [x]
mov     ebx, DWORD PTR [y]
```

Engineering takeaway: If you need multi-location consistency, use a protocol:

- publish/observe with acquire/release,
- versioning (sequence counters),
- single-writer designs,
- or a lock when the invariant requires it.

Minimal checklist for "CPU-level atomicity"

- Do you require **no torn reads/writes**? Ensure supported width + alignment + no line crossing.
- Do you require **no lost updates**? Use an atomic RMW (CAS or LL/SC) on the same location.
- Do you require **cross-location visibility/order**? That is memory ordering, not atomicity; do not conflate them.

Chapter 2

Cache Coherency and Visibility

2.1 Cache Lines and Ownership

Modern multicore CPUs do not communicate by "sharing memory" directly. They communicate by moving **cache lines** (fixed-size blocks, commonly 64 bytes on mainstream systems) between cores. Your program reads/writes *addresses*, but the hardware tracks and transfers *lines*.

Key facts that matter for atomics

- **Coherency is line-based.** Two independent variables in the same line are not independent in performance, and sometimes not independent in observed interleavings under contention.
- **Ownership is exclusive for writes.** To write a line, a core must obtain a writable state for that line.
- **The line is the unit of coherence traffic.** If one byte changes, the entire line

participates in ownership changes.

Example: false sharing caused by line ownership ping-pong

Two threads update different variables, but if they live on the same cache line, every write forces ownership transfer.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Assume 'a' and 'b' are in the same 64B cache line */

/* Thread 0 (Core 0): repeatedly increments a */
L0:
add     DWORD PTR [a], 1
jmp     L0

/* Thread 1 (Core 1): repeatedly increments b */
L1:
add     DWORD PTR [b], 1
jmp     L1
```

Result: the cache line bounces between cores (ownership ping-pong). Throughput collapses even though `a` and `b` are logically unrelated.

Example: atomic RMW amplifies ownership pressure

Atomic RMW instructions tend to serialize updates to the line. Under contention, they can be far more expensive than plain loads/stores because they force exclusive ownership and cannot be freely combined or reordered.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Contended atomic increment (same line, same location) */
lock add DWORD PTR [counter], 1
```

Takeaway: correctness is about ordering/atomicity, but performance is dominated by cache-line ownership and traffic.

2.2 Coherency Protocol Concepts

A **coherency protocol** ensures that for each cache line, cores agree on a single order of writes and that reads see a value consistent with that order. A practical mental model is:

- A line can be in a **readable shared** state on multiple cores.
- A line can be in a **writable exclusive** state on only one core at a time.
- To transition to writable, a core must **invalidate** other cores' copies (or force them to downgrade).

This is often explained with MESI-like states (Modified/Exclusive/Shared/Invalid). You do not need every state name; you need the invariants:

- **Single writer or many readers.** Not many writers simultaneously.
- **Writes require invalidations.** Other cores' cached copies must stop being valid for reading.
- **Reads can be satisfied locally** if the line is present and valid; otherwise the line is fetched.

Example: write requires obtaining exclusive ownership

Even a normal store must acquire a writable state for the line. This can trigger invalidations.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Core 0 writes X: must own line(X) in a writable state */
mov     DWORD PTR [X], 1

/* Core 1 reads X: may trigger fetch or may hit if it already has a
 ↪  shared copy */
mov     eax, DWORD PTR [X]
```

Example: why “read-only sharing” scales but “write-sharing” collapses

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Many cores can repeatedly read shared data with minimal coherence
 ↪  traffic */
mov     eax, DWORD PTR [read_only_value]
```

```
/* But many cores writing the same line continuously force constant
 ↪  invalidations */
mov     DWORD PTR [hot_write_value], 1
```

The difference is not the instruction count; it is the coherency protocol forcing ownership transfers.

What coherency guarantees (and what it does not)

Coherency guarantees:

- For a single location, all cores agree on a single order of writes to that location's line.
- A core will not read an arbitrarily old value forever; its line will eventually be invalidated/refreshed.

Coherency does not guarantee:

- any ordering relationship between different cache lines,
- any cross-location "publication" semantics by itself,
- any immediate visibility without the appropriate ordering mechanisms.

This is why coherent caches alone do not replace memory ordering.

2.3 Visibility and Propagation

Visibility answers: *when can another core observe a write?* **Propagation** answers: *how do writes spread through the coherence fabric?*
Even with coherence, visibility is not "instant" and not "in program order" for independent locations. Several mechanisms introduce delay and reordering of visibility:

- store buffers (stores become globally visible later),
- speculative execution and replays,
- write combining and merging,
- independent cache-line propagation.

Example: store-buffer window (conceptual symptom)

A core can execute a store, then later code reads as if the store has “happened”, while another core may still not observe that store yet.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Core 0 */
mov     DWORD PTR [X], 1          /* enters store buffer, later
 ↪  becomes globally visible */
mov     eax, DWORD PTR [Y]        /* continues executing */

/* Core 1 */
mov     ebx, DWORD PTR [X]        /* may still read 0 for a short
 ↪  window */
```

The key point: program order inside a core is not the same thing as global visibility order across cores.

Example: independent lines propagate independently

Publishing a flag and publishing data are often two different lines. Without an ordering edge, another core can observe them in an unexpected interleaving.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Assume 'data' and 'flag' are on different cache lines */

/* Producer */
```

```
mov     DWORD PTR [data], 777
mov     DWORD PTR [flag], 1

/* Consumer */
cmp     DWORD PTR [flag], 1
jne     not_ready
mov     eax, DWORD PTR [data]
not_ready:
```

Coherence ensures each line is coherent, but it does not force a cross-line visibility contract. To make `flag` a reliable publication mechanism for `data`, you must add acquire/release ordering (later chapters).

Example: invalidation-based visibility (conceptual)

When a core writes a line, other cores' copies are invalidated. Those cores will only see the new value when they reload and fetch the updated line.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Core 0: write forces others to drop stale shared copies */
mov     DWORD PTR [X], 2

/* Core 1: if it had a cached copy, it becomes invalid and must
 ↪  refetch on next load */
mov     eax, DWORD PTR [X]
```

Practical rules for engineers

- Treat the cache line as the unit of sharing: design data layout to avoid write-sharing.

- Coherence makes single-location reads/writes consistent; it does not publish multi-location invariants.
- If an algorithm depends on “seeing writes in order” across different locations, you need explicit memory ordering.

Chapter 3

Atomic Load and Store Operations

3.1 Alignment Requirements

Atomicity at the CPU level is conditional. The most common hidden condition is **alignment**. If you violate alignment requirements, a "single load/store instruction" can become multiple micro-operations, and other cores may observe a **torn** value.

What alignment protects

- **Single-copy atomicity**: observers see either the old value or the new value, not a mix.
- **Coherence granularity**: coherence operates on cache lines; crossing boundaries is risky.
- **Implementation freedom**: many ISAs allow unaligned access, but do not promise atomicity for it.

Practical rules (portable mental model)

- Use **natural alignment** for the width: 4-byte alignment for 32-bit, 8-byte for 64-bit.
- Avoid **cache-line crossing** for the width (especially for wide objects or packed structs).
- Do not assume that "works" implies "atomic".

Example: misalignment destroys atomicity (conceptual)

This sketch demonstrates how easy it is to create an access that is not guaranteed atomic.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* rdi = &buf[0] */
lea     rsi, [rdi + 1]                  /* deliberately misalign */

/* write 64-bit value at misaligned address */
mov     QWORD PTR [rsi], 0x1122334455667788

/* concurrent reader could observe a torn value if atomicity is not
 ↪  guaranteed */
mov     rax, QWORD PTR [rsi]
```

Engineering takeaway: if your correctness depends on atomicity, misalignment is a correctness bug, not a performance bug.

Example: packed layout can create accidental misalignment

Packed structures can place a naturally wide field at an unaligned offset.

```
/* Conceptual layout consequence:
   struct __attribute__((packed)) P { char c; uint64_t x; };
   x may be placed at offset 1 => misaligned 64-bit access possible
*/
```

When a field is misaligned, you have no safe basis to claim atomic loads/stores for that field on all targets.

3.2 Atomic Loads

An **atomic load** (in the hardware sense) means the load is not torn and participates correctly in the cache coherence protocol. But correctness in concurrent algorithms depends on two separate properties:

- **Atomicity of the load**: no tearing for the loaded location.
- **Ordering semantics of the load**: whether later reads/writes can be observed before the load (acquire vs relaxed).

In this chapter we focus on the first property: the load reads a single coherent value.

Example: plain aligned loads are typically atomic for their width

On mainstream 64-bit systems, aligned 32-bit and 64-bit loads are commonly single-copy atomic. But do not confuse that with synchronization.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* atomicity (no tear) for aligned 32-bit load is typical */
```

```
mov     eax, DWORD PTR [x]

/* atomicity (no tear) for aligned 64-bit load is typical */
mov     rax, QWORD PTR [y]
```

This ensures you do not read a torn value for `x` or `y`. It does *not* ensure anything about other data being visible.

Example: the correctness trap (atomic load does not mean ordered visibility)

A consumer can read a flag atomically but still read stale data without an ordering edge.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Consumer (naive): atomic load of flag, but no acquire contract is
 ↪  stated here */
mov     eax, DWORD PTR [flag]
cmp     eax, 1
jne     not_ready
mov     ebx, DWORD PTR [data]
not_ready:
```

The `mov [flag]` load is atomic as a load of that location. But the algorithm requires **publication**: seeing `flag==1` must imply seeing `data`'s writes. That implication is not provided by plain atomicity alone.

Example: repeated atomic loads amplify coherence traffic

Hot polling is not just a loop cost; it generates coherence reads and can starve the writer.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

spin:
mov     eax, DWORD PTR [flag]
test    eax, eax
je      spin
```

Correctness is still not guaranteed without proper ordering, and performance can be catastrophic if the flag shares a line with other writes.

3.3 Atomic Stores

An **atomic store** means other cores will not observe a torn value for that location. But just like loads, stores have two distinct concerns:

- **Atomicity of the store**: observers see old or new value for that location, not a mix.
- **Ordering semantics of the store**: whether prior writes must become visible before the store (release vs relaxed).

Here we focus on atomicity and coherence participation.

Example: plain aligned store is not torn but is not a publish primitive

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Producer: two independent stores */
mov     DWORD PTR [data], 123
mov     DWORD PTR [flag], 1
```

Each store (if aligned) is typically single-copy atomic for its width. Yet the algorithm's intent is: `flag==1` publishes `data`. That is an **ordering** requirement, not an atomicity requirement.

Example: contended stores cause ownership bouncing

Even non-atomic stores can cause high contention if multiple cores write the same line.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Core 0 */
mov     DWORD PTR [hot], 0

/* Core 1 */
mov     DWORD PTR [hot], 1
```

Each store forces the cache line into writable ownership on the writing core, invalidating the other core's copy. This is coherence-level serialization.

Example: atomic store to a flag still needs the right ordering contract

A flag store must usually be a **release** publication point in the algorithmic proof. At the assembly level, that release contract is expressed differently per ISA (later chapters), but the invariant is always the same.

- Producer: write data *then* publish flag (release).
- Consumer: observe flag (acquire) *then* read data.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Producer (pattern skeleton; ordering details handled later) */
mov     DWORD PTR [data], 777
mov     DWORD PTR [flag], 1          /* publication point */

/* Consumer (pattern skeleton; ordering details handled later) */
cmp     DWORD PTR [flag], 1          /* observation point */
jne     not_ready
mov     eax, DWORD PTR [data]
not_ready:
```

Checklist: when a load/store is safe to treat as atomic

- Width is supported and intended (avoid exotic wide objects without explicit atomic mechanisms).
- Address is naturally aligned for that width.
- Access does not cross a cache-line boundary.
- You are not relying on atomicity to provide ordering (ordering must be stated separately).

Chapter 4

Read-Modify-Write Instructions

4.1 RMW Instruction Semantics

A **read-modify-write (RMW)** operation performs three logical steps on a single memory location:

1. **Read** the current value.
2. **Compute** a new value from it.
3. **Write** the new value back.

The critical property is that these steps are **indivisible with respect to the same location**: no other core can interleave its own update between the read and the write of the RMW for that location.

Non-RMW vs RMW: the lost-update failure

A plain load+add+store is not indivisible and loses updates under contention.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Non-atomic increment: can lose updates */
mov     eax, DWORD PTR [counter]     /* read */
add     eax, 1                       /* modify */
mov     DWORD PTR [counter], eax     /* write */
```

Two cores can read the same old value, compute the same new value, and both store it, losing one increment.

Example: x86 locked RMW

On x86, the `lock` prefix turns the instruction into a coherent atomic RMW with strong serialization for that cache line.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Atomic increment */
lock add DWORD PTR [counter], 1

/* Atomic exchange (swap) */
xchg    DWORD PTR [x], eax           /* xchg with memory is atomic on
 ↪  x86 */

 /* Atomic bit test/set */
lock bts DWORD PTR [mask], 5
```

Example: why RMW is stronger than "atomic store" for coordination

Many coordination patterns require "observe old state and update to new state" atomically. A plain store can overwrite concurrently written state.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Naive: overwrite flag (not conditional, races with other writers)
 ↪  */
mov     DWORD PTR [flag], 1

/* RMW: conditional update (compare-and-swap) */
mov     eax, 0                      /* expected */
mov     ebx, 1                      /* desired */
lock cmpxchg DWORD PTR [flag], ebx  /* if *flag==0 => *flag=1 else
 ↪  eax=*flag */
```

RMW is the minimal primitive for many lock-free state machines.

4.2 Atomicity Boundaries

RMW provides indivisibility for **one memory location**. It does not magically make a multi-location invariant atomic. To reason correctly, define the boundary precisely:

- **Per-location atomicity**: the RMW is indivisible for the addressed object.
- **Per-cache-line coherence**: the whole line participates in ownership and serialization effects.
- **No multi-location atomicity**: updating two addresses is not one atomic action.

Boundary 1: RMW is atomic only with respect to the same location

RMW serializes with other RMW and stores to the *same* location, but it does not impose atomicity on adjacent variables.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Atomic update of counter */
lock add DWORD PTR [counter], 1

/* Independent update of data: not part of the same atomic action */
mov     DWORD PTR [data], 999
```

If your algorithm requires “counter and data must change together”, RMW is insufficient by itself.

Boundary 2: alignment and size still matter

Atomic RMW requires the object to be a supported width and correctly aligned. Violations can produce faults on some ISAs, or degrade to non-atomic sequences on others.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Conceptual: do not assume atomic RMW on misaligned or unsupported
 ↪  widths */
lea     rdi, [buf + 1]              /* misalign */
lock add QWORD PTR [rdi], 1         /* correctness depends on
 ↪  alignment guarantees */
```

Boundary 3: atomicity is not the same as ordering

RMW is indivisible for a location, but memory ordering (what other cores may observe before/after) must still be stated explicitly in the algorithm proof.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Atomic RMW on flag */
lock xchg DWORD PTR [flag], eax

/* Does not by itself prove that other writes (to other locations)
 ↪  are now visible as required.
   Publication/visibility must be reasoned with
    ↪  acquire/release/fences (later chapters). */
```

Example: multi-location invariant needs a protocol (sequence counter sketch)

A common pattern is to bracket multi-field writes with a version counter updated atomically.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Writer: increment seq to odd, write fields, increment seq to even
 ↪  */
lock add DWORD PTR [seq], 1          /* seq becomes odd => write in
 ↪  progress */
mov     DWORD PTR [field1], 10
mov     DWORD PTR [field2], 20
```

```
lock add DWORD PTR [seq], 1          /* seq becomes even => write
↪  complete */
```

Readers retry if `seq` is odd or changes during the read. This shows the boundary: RMW protects the counter, and the protocol protects the multi-field invariant.

4.3 Contention Effects

RMW instructions are correctness tools, but their performance cost is dominated by contention and coherence traffic. Under contention, RMW can become the **serialization point** of the entire system.

Why contended RMW is expensive

- The core must obtain **exclusive ownership** of the cache line.
- The operation creates a **single global modification order** for that location.
- Other cores attempting the same RMW must wait for ownership and completion.

Example: contended counter (scalability collapse)

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Many cores running this loop produce heavy cache-line bouncing */
L:
lock add DWORD PTR [counter], 1
jmp     L
```

Even though each increment is correct, throughput often degrades as core count grows because the cache line containing `counter` becomes a hot shared resource.

Example: CAS loop under contention (retry storm)

CAS-based algorithms can amplify contention: failures trigger retries, and retries trigger more traffic.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Sketch: CAS loop (conceptual) */
retry:
mov     eax, DWORD PTR [expected]        /* expected in eax */
mov     ebx, DWORD PTR [desired]         /* desired in ebx */
lock cmpxchg DWORD PTR [ptr], ebx        /* try update */
jne     retry                             /* retry on failure */
```

Observation: under high contention, most attempts fail, turning the loop into a coherence storm.

Example: test-and-test-and-set reduces traffic

Instead of RMW on every iteration, read first, then RMW only when it looks free.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* lock variable: 0 = free, 1 = held */

/* Test-and-test-and-set spin */
spin:
mov     eax, DWORD PTR [lockv]            /* read (shared) */
test    eax, eax
```

```
jne     spin

mov     eax, 1
lock xchg DWORD PTR [lockv], eax          /* RMW only when likely
 ↪  free */
test    eax, eax
jne     spin
```

This reduces RMW frequency and therefore reduces ownership bouncing while still remaining correct.

Example: per-core counters to avoid contended RMW

A scalable alternative is to avoid one shared hot location entirely.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Each core increments its own counter; aggregation is done
 ↪  occasionally */
add     DWORD PTR [local_counter], 1
```

Performance checklist for RMW

- Avoid contended single-location RMW in hot paths.
- Prefer protocols that shift work to **per-core** or **sharded** state.
- If spinning is required, avoid RMW-on-every-iteration.
- Measure: contention cost is microarchitecture- and topology-dependent.

Chapter 5

Compare-and-Swap (CAS)

5.1 CAS Execution Model

Compare-and-Swap (CAS) is the fundamental conditional RMW primitive used to build lock-free algorithms. It atomically performs:

> **If** *ptr == expected **then** *ptr = desired **else** return observed value (or report failure).

CAS gives you one crucial power: **it couples observation and update into a single indivisible attempt** for one memory location.

Why CAS exists (what it prevents)

A non-atomic "check then write" is always vulnerable to an interleaving writer.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix
```

```
/* Non-atomic check-then-set (racy) */
mov     eax, DWORD PTR [state]
cmp     eax, 0
jne     fail
mov     DWORD PTR [state], 1          /* another core could have
 ↪  changed state after the cmp */
fail:
```

CAS makes the check and the write a single atomic step.

x86-64 CAS instruction model: CMPXCHG

On x86-64, `cmpxchg` compares memory with an implicit accumulator register (EAX/RAX). With `lock`, it becomes an atomic RMW visible to all cores.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Attempt: if (*ptr == expected) *ptr = desired */
/* Inputs:
   rdi = ptr
   eax = expected (implicit compare operand)
   ebx = desired
*/
lock cmpxchg DWORD PTR [rdi], ebx     /* if [rdi]==eax then [rdi]=ebx
 ↪  else eax=[rdi] */
```

After the instruction:

- On success: memory updated; EAX remains the expected value.
- On failure: memory unchanged; EAX is overwritten with the observed memory value.

CAS loop: the lock-free "retry until it sticks" pattern

Most lock-free algorithms use a loop:

1. Load current value.
2. Compute desired new value.
3. CAS expected → desired.
4. If fail: repeat using the newly observed value.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* ptr in rdi, increment the 32-bit value atomically via CAS loop */
retry:
mov     eax, DWORD PTR [rdi]          /* expected = *ptr */
lea     ebx, [eax + 1]                /* desired  = expected + 1 */
lock cmpxchg DWORD PTR [rdi], ebx     /* attempt */
jne     retry                         /* failure => eax now holds
 ↪  observed *ptr, retry */
```

This loop is correct for the single-location update. Ordering/publishing of other data is a separate contract handled by acquire/release/fences in later chapters.

5.2 Success and Failure Semantics

CAS has two distinct outcomes, and correctness depends on treating both correctly.

Success semantics

When CAS succeeds:

- the location transitions from `expected` to `desired` **atomically**,
- the successful CAS defines a single step in the location's **modification order**,
- and the algorithm may treat that CAS as the moment it "won" the state transition.

Failure semantics (what failure actually means)

When CAS fails:

- it does **not** mean "no progress"; it usually means **someone else updated the location first**,
- the caller must read the newly observed value and recompute the desired transition,
- repeated failures can occur from contention, not from bugs.

Example: using the observed value on failure (x86 EAX overwrite)

A common bug is to retry with the old expected value. Correct usage consumes the newly observed value.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* ptr in rdi, try to change 0 -> 1 */

mov     eax, 0
mov     ebx, 1
```

```
lock cmpxchg DWORD PTR [rdi], ebx
jne     failed                          /* eax now holds *ptr
 ↪  (observed), not 0 */

success:
 /* state transitioned to 1 */
jmp     done

failed:
 /* eax contains the current value; decision should be based on eax
  ↪  */
 /* e.g., if eax != 0 someone already owns it */
done:
```

Spurious failure vs real failure

Some platforms expose **weak CAS** (allowed to fail even when the value matches), typically to enable LL/SC implementations. Even when not using weak CAS explicitly, engineers must design CAS loops assuming:

- failure is normal under contention,
- forward progress is not guaranteed for a single thread (lock-free vs wait-free distinction).

Example: contention creates a retry storm

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix
```

```
/* Many cores running this CAS loop on the same location can spend
 ↪  most time failing */
retry2:
mov     eax, DWORD PTR [ptr]
/* compute desired from eax */
mov     ebx, eax
add     ebx, 1
lock cmpxchg DWORD PTR [ptr], ebx
jne     retry2
```

This is still correct, but performance can collapse because CAS concentrates traffic on one cache line.

5.3 ABA Problem

CAS checks only one thing: **the current value equals expected**. It does not know whether the value has changed *in between*, as long as it returned to the same bits. This is the **ABA problem**:

> A reads value A. Another thread changes it to B, then back to A. CAS sees A and succeeds, even though the world changed in between.

ABA is not a theoretical corner case. It is a concrete failure mode in pointer-based lock-free stacks, queues, and free lists.

Classic scenario: lock-free stack pop

- Thread T1 reads `head = A` and plans to CAS `head` to `A->next`.
- Thread T2 pops A (head becomes B), then frees A, then allocates a new node whose address is also A, and pushes it back (head becomes A again).

- T1 CAS sees `head == A` and succeeds, but it is not the same logical node A anymore.

Example sketch (pointer CAS concept)

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* ptr in rdi points to head (a pointer-sized location) */
/* expected in rax, desired in rbx */
lock cmpxchg QWORD PTR [rdi], rbx
```

If the address bits match, CAS succeeds. ABA exploits that addresses can repeat.

How ABA is mitigated (core techniques)

Real systems use one or more of these:

- **Tagged pointers / version counters**: store (pointer, counter) together and CAS both.
- **Hazard pointers**: prevent reclamation while another thread may still hold a reference.
- **Epoch-based reclamation (EBR)**: defer frees until all threads have passed a safe point.

Example: tagged pointer idea (conceptual 64-bit packing)

Pack a small counter with the pointer so the CAS detects A→B→A.

```
/* Conceptual: head = (ptr | tag) packed into 64 bits.
   CAS checks both pointer bits and tag bits, so ABA changes the tag.
*/
```

Example: minimal version-counter protocol (sequence tag)

Even without packing, the invariant is simple:

- Each logical update increments a tag.
- CAS compares both value and tag.

Engineering takeaway: If you use CAS on pointers and allow memory reclamation/reuse, you must address ABA explicitly. Ignoring it yields rare, catastrophic correctness failures that escape testing and appear only under real contention.

Chapter 6

Load-Linked / Store-Conditional

6.1 LL/SC Operation Flow

Load-Linked / Store-Conditional (LL/SC) is an atomic update mechanism used by several ISAs to implement conditional RMW without a single "CAS" instruction. The model is:

1. **LL (load-linked)** reads the value and establishes a **reservation** on the addressed location (or its granule).
2. You compute the desired new value in registers.
3. **SC (store-conditional)** attempts to store the new value *only if* the reservation is still valid.
4. SC reports **success** or **failure**. On failure, you retry from LL.

The crucial property is: if another core performs a conflicting write to the reservation granule, your reservation is cleared, and SC fails. This prevents interleaving updates between your read and write.

LL/SC is conditional RMW, not “two ordinary instructions”

A naive “load then store” has a race window. LL/SC closes that window by making the store conditional on the reservation.

```
/* Conceptual LL/SC pseudocode in assembly-like form.
   Real mnemonics differ across ISAs. */

/* retry:
      old = LL(ptr)
      new = f(old)
      ok  = SC(ptr, new)
      if (!ok) goto retry
*/
```

Example: atomic increment via LL/SC (conceptual)

```
/* Conceptual LL/SC atomic increment */

/* retry: */
 /* old = LL([ptr]) */
 /* new = old + 1 */
 /* ok  = SC([ptr], new) */
 /* if (!ok) goto retry */
```

This implements the same logical contract as a CAS loop: **read current value → compute new value → attempt conditional write**.

Important detail: reservation granularity

Reservations are not always “one byte” or “one word”; they are often tied to an implementation-defined granule (e.g., a cache line or sub-line region). Therefore, unrelated writes within the granule can also cause SC to fail, even if the exact word you care about did not change.

```
/* Conceptual: reservation is on a granule, not necessarily on the
 ↪  exact word.
   A write to a nearby address may invalidate the reservation and
    ↪  force SC to fail.
*/
```

6.2 Spurious Failure

Spurious failure means: SC can fail even when no other thread performed a logically conflicting update to the target location.
This is not a bug in the ISA; it is an explicit design property that allows efficient microarchitectural implementations. Common causes include:

- interrupts, exceptions, context switches,
- preemption / migration to another core,
- cache eviction of the reserved line,
- internal coherence events affecting the reservation granule,
- implementation limits on how many reservations can be tracked.

Consequence: LL/SC loops must always be written as retry loops

You may never write LL/SC logic assuming “if no one touches it, SC must succeed”.

```
/* Conceptual robust LL/SC loop */

/* retry: */
 /* old = LL([ptr]) */
 /* new = f(old) */
 /* ok  = SC([ptr], new) */
 /* if (!ok) goto retry */
```

Example: why “SC should succeed” is a wrong assumption

Even with a single thread, SC can fail due to events unrelated to contention (e.g., preemption).

```
/* Conceptual timeline:
   T: LL(ptr) -> reservation set
   OS: preempts T
   Other events: invalidate reservation (migration/eviction)
   T: resumes, SC(ptr,new) -> fails (spurious)
*/
```

Engineering rule

Treat SC failure as a normal event. The algorithm must remain correct and terminate under the intended progress guarantees (lock-free vs wait-free), not under wishful assumptions about success rates.

6.3 Forward Progress

LL/SC enables lock-free algorithms, but **forward progress** is not automatically guaranteed for every thread. You must be precise about what your algorithm promises:

- **Lock-free**: system as a whole makes progress (some thread completes) even under contention.
- **Wait-free**: every thread completes in a bounded number of steps (stronger, harder).

LL/SC primitives usually guarantee that:

- If a thread repeatedly retries and contention eventually stops, it will eventually succeed.
- Under heavy contention, one thread may starve while others succeed.

Example: starvation under contention (conceptual)

Two threads compete on the same location. One repeatedly invalidates the other's reservation. Both are correct, but one might lose repeatedly and make no progress.

```
/* Conceptual: T0 and T1 run the same LL/SC loop on [ptr].
   T0's updates keep invalidating T1's reservation, causing T1's SC
    ↪  to fail repeatedly.
*/
```

Practical forward-progress techniques

To improve progress behavior under contention:

- **Backoff**: pause after failures (exponential or randomized).

- **Reduce granule conflicts**: avoid placing frequently written independent variables in the same line/granule.
- **Sharding**: replace one hot location with per-core or per-bucket locations.
- **Bound retries**: if retries exceed a threshold, fall back to a lock (hybrid design).

Example: backoff on repeated SC failure (conceptual)

```
/* Conceptual LL/SC with backoff */

/* retry:
      old = LL([ptr])
      new = f(old)
      if (SC([ptr], new)) done
      pause/backoff
      goto retry
*/
```

Key takeaway

LL/SC gives a clean **conditional update mechanism** but forces you to design with:

- retry as a first-class control flow,
- spurious failure as normal behavior,
- and progress guarantees as an explicit design goal, not an assumption.

Chapter 7

Memory Reordering Fundamentals

7.1 Sources of Reordering

Memory reordering is not “the CPU being random”. It is the predictable result of performance mechanisms. To write correct concurrent code, you must accept this rule:

> **Within one thread, your source order is not a visibility contract.**

Reordering comes from two independent engines:

- **Compiler reordering**: legal transformations that preserve single-thread behavior.
- **Hardware reordering**: microarchitectural execution/visibility behavior across cores.

Main hardware sources

- **Out-of-order execution**: later operations may execute before earlier ones.
- **Store buffers**: stores become globally visible later than the core that issued them.

- **Invalidate queues / coherence latency**: other cores observe writes after coherence traffic completes.
- **Speculation**: loads may be issued before control-flow is resolved, then squashed/replayed.
- **Independent cache lines propagate independently**: writes to different lines can become visible in different orders.

Main compiler sources

- **Common subexpression elimination** and **load hoisting**: reuse/hoist loads across code motion.
- **Store sinking** and **dead store elimination**: delay or remove stores if single-thread semantics permit.
- **Instruction scheduling**: reorder independent memory ops for latency.
- **Inlining** and **loop transformations**: change structure and ordering of memory operations.

Example: why CPUs reorder at all (store buffer window)

The core may proceed after issuing a store while the store waits to become globally visible.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Thread A */
mov     DWORD PTR [X], 1          /* may sit in store buffer before
 ↪  other cores see it */
```

```
mov     DWORD PTR [Y], 1          /* another store, same story */

/* Thread B */
mov     eax, DWORD PTR [Y]        /* could see Y==1 */
mov     ebx, DWORD PTR [X]        /* and still see X==0 on weak
 ↪  models (and in some patterns) */
```

This is the fundamental reason “I wrote X then Y” is not a cross-core guarantee unless you enforce ordering.

7.2 Compiler vs Hardware Effects

Correct reasoning requires separating:

- what the **compiler** is allowed to do before the code becomes machine instructions,
- and what the **CPU** is allowed to do when executing those instructions and making results visible to other cores.

Compiler reordering: the as-if rule meets concurrency

Compilers preserve the observable behavior of a **single thread**. Without atomics/fences, the compiler is allowed to reorder memory operations in ways that are catastrophic under concurrency.

Example: load hoisting (conceptual)

A polling loop that reads a flag can be transformed if the compiler sees no synchronization.

```
/* Conceptual transformation risk:
```

```
   while(flag==0) { }  may be treated as an infinite loop if 'flag'
    ↪  is not atomic/volatile in the source model.
   The resulting machine code may load once, then loop forever.
*/
```

Engineering rule: use real atomics/fences to constrain the compiler, not hope.

Hardware reordering: execution order vs visibility order

Even if the compiler emits instructions in a particular order, the CPU can:

- execute them out-of-order internally,
- and make their effects visible to other cores in an order permitted by the ISA memory model.

Example: store-load ordering is the classic weak point

The most dangerous pattern is when a store in one thread and a later load in that thread can be observed out of order. Some architectures allow this aggressively; some restrict it, but not all cases are eliminated.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Thread A */
mov     DWORD PTR [X], 1
mov     eax, DWORD PTR [Y]

/* Thread B */
mov     DWORD PTR [Y], 1
mov     ebx, DWORD PTR [X]
```

The surprising outcome (on weak models) is: `eax==0` and `ebx==0` even though each thread stored 1 before loading. This is the classic demonstration that visibility order is not program order.

Two different fixes, two different targets

- To stop **compiler** motion: use language-level atomics / compiler barriers.
- To stop **hardware** visibility reordering: use acquire/release or fences as required by the ISA.

Do not confuse them: a compiler barrier does not flush store buffers, and a hardware fence does not stop the compiler from reordering around it unless the fence is expressed in a way the compiler respects.

7.3 Observable Anomalies

Reordering becomes real only when it produces outcomes that violate naive expectations. These outcomes are not hypothetical: they are the expected results of allowed reorderings.

Anomaly 1: store buffering (both read 0)

Two threads store then load. Each may read 0, contradicting "I stored first" intuition.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Shared: X=0, Y=0 */

/* Thread A */
```

```
mov     DWORD PTR [X], 1
mov     eax, DWORD PTR [Y]

/* Thread B */
mov     DWORD PTR [Y], 1
mov     ebx, DWORD PTR [X]
```

Allowed on weak memory models: `eax==0 && ebx==0`.

Anomaly 2: load buffering (both read 0)

Two threads load then store. Both can read 0 even though both later store 1.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Shared: X=0, Y=0 */

/* Thread A */
mov     eax, DWORD PTR [X]
mov     DWORD PTR [Y], 1

/* Thread B */
mov     ebx, DWORD PTR [Y]
mov     DWORD PTR [X], 1
```

The key lesson: the reads can happen before the stores become visible to the other core.

Anomaly 3: message passing without ordering (flag seen, data stale)

The flag is observed as set, but the data is still the old value.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Shared: data=0, flag=0 */

/* Producer */
mov     DWORD PTR [data], 777
mov     DWORD PTR [flag], 1

/* Consumer */
cmp     DWORD PTR [flag], 1
jne     not_ready
mov     eax, DWORD PTR [data]     /* could still read 0 without a
 ↪  publish/observe ordering edge on weak models */
not_ready:
```

Anomaly 4: “impossible” interleavings across independent locations

Without a defined ordering edge, observing one write does not imply observing another write, even if both came from the same thread.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Thread A writes two variables */
mov     DWORD PTR [A], 1
mov     DWORD PTR [B], 1

/* Thread B reads in opposite order */
mov     eax, DWORD PTR [B]        /* sees 1 */
```

```
mov     ebx, DWORD PTR [A]          /* sees 0 (allowed on weak models)
↪  */
```

What this chapter establishes

- Reordering is expected, not exceptional.
- Correctness requires explicit ordering contracts.
- “Atomic” alone prevents tearing for a location; it does not define cross-location visibility.

Chapters 8 and 9 convert these anomalies into controllable outcomes using memory ordering and fences.

Chapter 8

Memory Ordering Models

8.1 Sequential Consistency

Sequential Consistency (SC) is the strongest widely used ordering model. The mental model is:

> All threads observe all SC atomic operations as if they occur in **one single global order**, and each thread's operations appear in that order consistent with program order.

SC does *not* mean "no reordering inside the CPU"; it means the **observable behavior** matches that single-order model for the SC operations involved.

What SC buys you

- A single global story for the relevant atomic operations.
- Litmus outcomes that vanish under weaker models become forbidden.
- Reasoning becomes closer to interleaving-based proofs.

What SC does not buy you

- It does not make non-atomic data-race accesses safe.
- It does not automatically publish unrelated non-atomic data unless you build a correct publish/observe pattern.
- It is often more expensive than acquire/release on weak ISAs because it can require stronger barriers.

Example: SC forbids the classic “both read 0” outcome (conceptual)

With SC atomics on X and Y, the store-buffering anomaly is forbidden for those SC operations.

```
/* Conceptual litmus with SC atomics (not syntax, but contract) */

/* Shared: X=0, Y=0 */

/* Thread A */
 /* SC_store(X,1) */
 /* r1 = SC_load(Y) */

/* Thread B */
 /* SC_store(Y,1) */
 /* r2 = SC_load(X) */

/* Under SC, r1==0 && r2==0 is forbidden. */
```

Example: SC atomic flag publish (strong but often overkill)

```
/* x86-64 (GAS) shown with Intel syntax enabled */
```

```
.intel_syntax noprefix

/* Producer (treat flag store as SC atomic publication point) */
mov     DWORD PTR [data], 777
mov     DWORD PTR [flag], 1          /* conceptually SC store */

/* Consumer (treat flag load as SC) */
cmp     DWORD PTR [flag], 1          /* conceptually SC load */
jne     not_ready
mov     eax, DWORD PTR [data]
not_ready:
```

SC can make reasoning simpler, but it may impose stronger constraints than necessary. Most real designs use acquire/release for publication and relaxed for counters/statistics.

8.2 Acquire and Release

Acquire and **Release** are the workhorses of scalable concurrency. They do not impose one global total order; instead, they build **edges** that connect threads:

- A **release** operation prevents earlier memory operations in the same thread from moving after it and serves as a **publication point**.
- An **acquire** operation prevents later memory operations in the same thread from moving before it and serves as an **observation point**.

The core guarantee is:

> If thread B performs an **acquire load** that reads the value written by thread A's **release store**, then B will observe A's prior writes (as required by the model).

Message passing pattern (the canonical acquire/release use)

This is the minimal contract for safe publication of data behind a flag.

- Producer: write data, then release-store flag.
- Consumer: acquire-load flag, then read data.

Example: AArch64 uses acquire/release instructions directly

```
/* AArch64 shown conceptually */

/* Producer */
str     w0, [xData]          /* data = w0 */
stlr    w1, [xFlag]          /* flag = 1   (release) */

/* Consumer */
ldar    w2, [xFlag]          /* acquire load of flag */
cbz     w2, not_ready
ldr     w3, [xData]          /* now ordered after acquire */
not_ready:
```

Example: RISC-V can express edges with fences (conceptual)

```
/* RISC-V shown conceptually */

/* Producer: release-like */
sw      t0, 0(a0)            /* *data = t0 */
fence   rw, w                /* release edge: prior R/W before
 ↪  following W */
sw      t1, 0(a1)            /* *flag = 1 */
```

```
/* Consumer: acquire-like */
lw      t2, 0(a1)             /* t2 = *flag */
beq     t2, x0, not_ready
fence   r, rw                 /* acquire edge: prior R before
 ↪  following R/W */
lw      t3, 0(a0)             /* t3 = *data */
not_ready:
```

Example: x86-64 often needs no explicit fence for basic message passing, but the contract is still acquire/release

x86-64 has stronger default ordering for many load/store cases, but the correct way to reason is still:

- treat the flag store as release,
- treat the flag load as acquire,
- and never rely on “it works on my machine” folklore.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Producer */
mov     DWORD PTR [data], 777
mov     DWORD PTR [flag], 1          /* conceptually release */

/* Consumer */
mov     eax, DWORD PTR [flag]        /* conceptually acquire */
test    eax, eax
```

```
je      not_ready
mov     ebx, DWORD PTR [data]
not_ready:
```

Acquire/Release is about ordering, not atomicity width

Acquire/release does not increase the size of atomicity; it constrains reordering/visibility. You still need proper alignment and supported widths for atomicity of the operations themselves.

8.3 Relaxed Ordering

Relaxed ordering provides atomicity for the location (no torn reads/writes, no lost updates for RMW), but provides **no ordering guarantees** with respect to other memory operations. Relaxed is correct when:

- you need an atomic counter/statistic where only the final value matters,
- you need uniqueness (e.g., fetch-add ID allocator) but not ordering of other data,
- you are building higher-level protocols where ordering is established elsewhere.

Example: relaxed counter (correct and scalable)

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Atomic increment for statistics: ordering against other data is
 ↪  irrelevant */
lock add DWORD PTR [stats_counter], 1
```

The increment is atomic for `stats_counter`, but it does not publish or order other memory.

Example: relaxed fetch-add ID allocation (uniqueness without publication)

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* rax = old; memory = old + 1 (conceptual fetch-add using xadd) */
mov     eax, 1
lock xadd DWORD PTR [next_id], eax   /* eax receives old value;
 ↪  [next_id] increases */
```

Each thread gets a unique old value (ID) atomically. This does not provide ordering for anything else.

Example: relaxed flag is a bug in message passing

If you try to publish data with a relaxed flag, you have not created the required edge.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Producer (wrong if 'flag' is treated as relaxed publication) */
mov     DWORD PTR [data], 777
mov     DWORD PTR [flag], 1          /* relaxed store: no publish
 ↪  guarantee */

/* Consumer (wrong if 'flag' is treated as relaxed observation) */
mov     eax, DWORD PTR [flag]        /* relaxed load: no acquire
 ↪  guarantee */
test    eax, eax
```

```
je      not_ready
mov     ebx, DWORD PTR [data]        /* may observe stale data on
 ↪  weak models */
not_ready:
```

Minimal decision table

- Use **Relaxed** for counters/IDs when only atomicity for that location matters.
- Use **Acquire/Release** for publication and synchronization edges.
- Use **SC** when you need one global order story and accept the cost.

Chapter 9

Memory Barriers and Fences

9.1 Fence Types

A **fence** (memory barrier) is an instruction or mechanism that constrains how memory operations are allowed to be **reordered** and **observed** across cores. Fences are not "slow no-ops": they are explicit constraints on the CPU's optimization freedom.
You should classify fences by the **kind of ordering they constrain**:

- **Load-load (R→R)**: prevents later loads from being observed before earlier loads.
- **Store-store (W→W)**: prevents later stores from being observed before earlier stores.
- **Load-store (R→W)**: prevents later stores from being observed before earlier loads.
- **Store-load (W→R)**: prevents later loads from being observed before earlier stores (often the hardest/most expensive).

Many ISAs expose fences as:

- **Full fence**: constrains both reads and writes in both directions.

- **Acquire fence**: prevents later operations from moving before (R and sometimes W).
- **Release fence**: prevents earlier operations from moving after (R and W before W).
- **Lightweight / domain-specific fences**: separate ordering for reads vs writes, or for inner-shareable domains.

x86-64 fence family (conceptual roles)

x86-64 commonly uses:

- **LFENCE**: order loads (and acts as a speculation barrier in many contexts).
- **SFENCE**: order stores (not always needed for normal WB memory).
- **MFENCE**: full fence (orders loads and stores).
- **LOCKed RMW**: often acts as a strong ordering point for the affected line, and is also an atomic RMW.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

lfence
sfence
mfence

/* Locked RMW often provides a strong ordering point for the line */
lock add DWORD PTR [counter], 1
```

AArch64 fence family (conceptual roles)

AArch64 expresses ordering through:

- **DMB**: data memory barrier (ordering).
- **DSB**: stronger barrier (completion before continuing).
- **Acquire/Release loads/stores**: LDAR/STLR often avoid full barriers.

```
/* AArch64 shown conceptually */

dmb     ish         /* order memory accesses in inner-shareable domain
 ↪  */
dsb     ish         /* stronger completion barrier */

/* Prefer acquire/release instructions when sufficient */
ldar    w0, [x1]
stlr    w2, [x3]
```

RISC-V fence family (conceptual roles)

RISC-V uses:

- **FENCE**: order sets of reads/writes before/after.
- **Acquire/Release annotations** on AMOs and LR/SC in many designs.

```
/* RISC-V shown conceptually */

fence   rw, rw      /* full-like: prior R/W before following R/W */
fence   rw, w       /* release-like: prior R/W before following W */
fence   r, rw       /* acquire-like: prior R before following R/W */
```

Engineering rule: treat each fence as an explicit statement of which reorderings you forbid. Do not use fences as "mystical fixes".

9.2 Ordering Guarantees

Fences create **local** ordering constraints in one thread that become meaningful **globally** when paired with coherence and with an observe/publish protocol.

Release and Acquire: the minimal publication guarantee

The canonical publication pattern is:

- Producer: write data, then **release** publish a flag.
- Consumer: **acquire** observe the flag, then read data.

A release fence (or release store) ensures the producer's prior writes become visible before the flag is visible. An acquire fence (or acquire load) ensures the consumer's later reads cannot be satisfied before observing the flag.

Example: release fence then flag store (conceptual)

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Producer: publish data then flag with an explicit release fence
 ↪  (often overkill on x86) */
mov     DWORD PTR [data], 777
mfence                          /* release-like full fence */
mov     DWORD PTR [flag], 1
```

Example: acquire fence after flag load (conceptual)

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Consumer: observe flag then fence then read data */
mov     eax, DWORD PTR [flag]
test    eax, eax
je      not_ready
mfence                          /* acquire-like full fence */
mov     ebx, DWORD PTR [data]
not_ready:
```

Important: full fences are correct but can be unnecessarily expensive. Prefer acquire/release operations when they express exactly what you need.

Example: AArch64 acquire/release without full barriers

```
/* AArch64 shown conceptually */

/* Producer */
str     w0, [xData]
stlr    w1, [xFlag]           /* release publish */

/* Consumer */
ldar    w2, [xFlag]           /* acquire observe */
cbz     w2, not_ready
ldr     w3, [xData]
not_ready:
```

Example: RISC-V fence-based acquire/release

```
/* RISC-V shown conceptually */

/* Producer */
sw      t0, 0(a0)
fence   rw, w                 /* release */
sw      t1, 0(a1)

/* Consumer */
lw      t2, 0(a1)
beq     t2, x0, not_ready
fence   r, rw                 /* acquire */
lw      t3, 0(a0)
not_ready:
```

What fences do *not* guarantee

- Fences do not make non-atomic races safe at the language level.
- Fences do not provide multi-location atomicity.
- A fence in one thread does nothing unless the other thread uses a compatible observation mechanism.

Litmus outcome control (conceptual)

Fences exist to forbid specific outcomes such as “both read 0” in store-buffering patterns or “flag seen, data stale” in message passing. The correct fence is the one that forbids the unwanted outcome with the minimal constraint.

9.3 Performance Costs

Fences cost performance because they reduce the CPU's ability to overlap and reorder work. They also increase coherence pressure by forcing ordering that may require additional waiting.

Why fences are expensive (mechanisms)

- They can force the core to wait for the store buffer to drain (visibility ordering).
- They can block speculative loads crossing the barrier.
- They can reduce memory-level parallelism by preventing independent misses from being issued early.
- They can serialize instruction retirement around memory ordering points.

Example: over-fencing destroys throughput

A common anti-pattern is fencing around every shared access.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Anti-pattern: fence on every iteration */
loop:
mfence
mov     eax, DWORD PTR [flag]
mfence
test    eax, eax
je      loop
```

This can turn a fast polling loop into a pipeline-stalling sequence that consumes bandwidth and blocks useful overlap.

Example: prefer acquire/release instead of full fences

If your goal is publication, use a release store and acquire load (or ISA equivalents), not full fences everywhere.

```
/* AArch64 shown conceptually */
/* Acquire/release is typically cheaper than full barrier-based
 ↪  designs */
stlr    w1, [xFlag]
ldar    w2, [xFlag]
```

Example: locked RMW as an implicit heavy barrier (x86)

Locked RMW operations are correct for atomic updates, but they can be expensive under contention.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

lock add DWORD PTR [counter], 1
```

Under contention, this can serialize multiple cores on one cache line (ownership bouncing), so throughput collapses even if correctness is perfect.

Practical cost rules

- Use the weakest ordering that proves the invariant: relaxed for counters, acquire/release for publication, SC only when needed.

- Avoid full fences in hot paths unless you can justify them with a forbidden-outcome argument.
- Do not rely on fences to compensate for poor data layout (false sharing).
- Measure: fence cost is architecture- and microarchitecture-dependent.

Chapter 10

False Sharing and Cache Contention

10.1 Cache Line Granularity

Modern multicore CPUs share memory through **cache lines**, not individual variables. A cache line is a fixed-size block (commonly 64 bytes on mainstream systems). Coherency ownership is tracked per line. This produces a harsh rule:

> If two cores write different words on the same cache line, they are still contending on the **same coherence resource**.

What "granularity" means in practice

- **Coherence unit**: reads/writes participate in ownership and invalidation at line granularity.
- **Transfer unit**: the interconnect moves whole lines between caches.
- **Serialization unit**: frequent writes force the line to bounce between cores.

Example: two independent counters can still fight

Assume `a` and `b` are 4-byte counters placed within the same 64B line.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Core 0 updates 'a' */
L0:
add     DWORD PTR [a], 1
jmp     L0

/* Core 1 updates 'b' */
L1:
add     DWORD PTR [b], 1
jmp     L1
```

Even though `a` and `b` are different addresses, each store invalidates the other core's cached copy of the *entire* line, forcing constant ownership transfers.

Example: a hot flag can poison nearby read-mostly data

If a frequently written flag shares a line with read-mostly data, readers of the data are forced into coherence churn.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Writer toggles flag frequently */
mov     DWORD PTR [flag], 1
mov     DWORD PTR [flag], 0
```

```
/* Readers frequently load read-mostly config that shares the same
 ↪  line */
mov     eax, DWORD PTR [config_value]
```

The “config” reads now suffer from invalidations caused by the flag, even though config is logically independent.

10.2 False Sharing Patterns

False sharing is performance collapse due to multiple threads writing different variables that happen to reside on the same cache line.
The most common patterns:

Pattern 1: per-thread counters stored in a packed array

Each thread updates `counters[tid]` but the array packs many counters into one line.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Core i increments counters[i] in a tight loop */
add     DWORD PTR [counters + rsi*4], 1
```

If multiple active threads have indices that map into the same line, the line bounces continuously.

Pattern 2: adjacent fields in a shared struct updated by different threads

A “status” field updated by one thread and a “progress” field updated by another share a line.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Thread A updates status */
mov     DWORD PTR [obj_status], 1

/* Thread B updates progress */
mov     DWORD PTR [obj_progress], 55
```

Even though they are different fields, they are the same line: invalidation ping-pong.

Pattern 3: lock word shares a line with protected data

Lock acquisition/release writes the lock word. If lock and data share the line, every lock write invalidates data.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* lock acquisition involves writes/RMW to lockv */
mov     eax, 1
lock xchg DWORD PTR [lockv], eax

/* critical section reads data that shares the line with lockv */
mov     ebx, DWORD PTR [protected_data]
```

This makes the lock far more expensive than necessary and increases cache misses inside the critical section.

Pattern 4: atomics amplify false sharing

Atomic RMW operations concentrate traffic on a line even more than plain stores.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Multiple threads contending on the same line: heavy coherence
 ↪  serialization */
lock add DWORD PTR [shared_atomic], 1
```

This is correct but can be the dominant system bottleneck.

10.3 Mitigation Techniques

False sharing is solved by **layout discipline** and **algorithmic structure**, not by fences.

Technique 1: separate frequently written fields onto different cache lines

The simplest fix: **pad** or **align** hot fields so two writers never share a line.

```
/* Conceptual layout:
   align hot counters/flags to 64 bytes and pad to 64 bytes between
    ↪  them.
*/
```

Rule: ensure each independently-written hot variable occupies its own line.

Technique 2: per-core / sharded counters

Replace one contended counter with an array of per-core counters and aggregate occasionally.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix
```

```
/* Increment per-core counter: no sharing, no bouncing */
add     DWORD PTR [local_counter], 1
```

Aggregation happens on a slower path:

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Sum all local counters occasionally (single thread or infrequent)
 ↪  */
mov     eax, DWORD PTR [local0]
add     eax, DWORD PTR [local1]
add     eax, DWORD PTR [local2]
```

Technique 3: isolate lock words

Keep lock variables on their own line, away from hot read-mostly data and protected data.

```
/* Conceptual:
   align lockv to its own cache line so lock traffic does not
    ↪  invalidate protected data lines.
*/
```

Technique 4: avoid write-sharing by design

Prefer designs where:

- one thread owns a data structure (single-writer),
- readers observe snapshots (publish/observe),
- writers communicate via queues (SPSC/MPSC) rather than shared hot fields.

Technique 5: reduce RMW frequency under contention

If you must spin:

- use test-and-test-and-set (read first, RMW only when likely free),
- add backoff after failures,
- avoid hammering the interconnect with locked RMW every iteration.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Test-and-test-and-set style spin (reduces RMW traffic) */
spin:
mov     eax, DWORD PTR [lockv]
test    eax, eax
jne     spin

mov     eax, 1
lock xchg DWORD PTR [lockv], eax
test    eax, eax
jne     spin
```

Field checklist note

When a program “mysteriously” stops scaling with more cores, check first:

- Are multiple threads writing different fields in the same cache line?
- Is a hot atomic or lock sharing a line with other frequently accessed data?
- Are per-thread structures accidentally packed into the same line?

False sharing is often the highest-leverage performance fix in real concurrent systems.

Chapter 11

Lock-Free and Wait-Free Progress

11.1 Progress Guarantees

Progress guarantees describe what happens **under contention** and **under unfair scheduling**. They are not performance claims; they are **liveness contracts**.

The three practical classes

- **Blocking (locks)**: a stalled thread can prevent others from making progress.
- **Lock-free**: the system as a whole makes progress; in any finite number of steps, *some* thread completes its operation.
- **Wait-free**: every thread completes its operation in a bounded number of steps (regardless of other threads).

What lock-free really means

Lock-free does *not* mean “no waiting”. It means: even if one thread is paused, other threads can still complete.

Example: CAS loop is lock-free (not wait-free)

CAS loops can guarantee system-wide progress but allow individual starvation.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* ptr in rdi: atomic increment using CAS loop (lock-free, not
 ↪  wait-free) */
retry:
mov     eax, DWORD PTR [rdi]          /* expected */
lea     ebx, [eax + 1]                /* desired */
lock cmpxchg DWORD PTR [rdi], ebx     /* attempt */
jne     retry                         /* retry on failure */
```

Under contention, one thread may fail repeatedly while others succeed.

Example: single-writer ring buffer can be wait-free (SPSC idea)

If only one producer and one consumer touch disjoint indices (with correct ordering), operations can complete without retries. The key is eliminating multi-writer contention on a single location.

```
/* Conceptual: SPSC queue progress is driven by single-writer
 ↪  ownership of head/tail indices.
   No multi-writer CAS loops are required in the steady state.
*/
```

Engineering decision rule

- Use **wait-free** only when bounded latency is required and complexity is justified.
- Use **lock-free** when blocking is unacceptable but occasional retries are acceptable.
- Use **locks** when invariants are complex and contention is low or bounded.

11.2 Starvation and Livelock

Starvation

Starvation occurs when a thread fails to make progress for an unbounded time, even though others continue. Lock-free algorithms can starve.

Example: CAS starvation under contention

One thread keeps losing the race, failing CAS forever while others keep winning.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Many cores run this; one unlucky core can fail repeatedly */
retry2:
mov     eax, DWORD PTR [ptr]
mov     ebx, eax
add     ebx, 1
lock cmpxchg DWORD PTR [ptr], ebx
jne     retry2
```

This is lock-free (someone succeeds), but a particular thread can starve.

Livelock

Livelock occurs when threads keep executing but the system makes no useful progress because they repeatedly interfere with each other.

Example: symmetric retry storm (livelock tendency)

Two threads repeatedly invalidate each other (CAS retries or LL/SC failures), amplifying contention.

```
/* Conceptual livelock pattern:
   T0 and T1 run the same retry loop with no backoff.
   Both generate constant coherence traffic and repeated failures.
*/
```

Backoff is not optional engineering

Backoff reduces coherence storms and increases the probability that one thread completes.

```
/* Conceptual backoff structure */

/* retry:
      expected = *ptr
      desired  = f(expected)
      if (CAS(ptr, expected, desired)) done
      pause/backoff
      goto retry
*/
```

Hybrid fallback is often the correct production design

Many high-quality systems use:

- fast lock-free path under low contention,
- fallback to a lock or slow path when retries exceed a threshold.

This bounds worst-case CPU consumption and avoids infinite retry under pathological contention.

Example: bounded retry then fallback (conceptual)

```
/* Conceptual:
   for i in 0..K:
       attempt CAS
       if success return
       backoff
   acquire lock
   perform operation
   release lock
*/
```

11.3 Hardware Limitations

Progress guarantees are limited by real hardware and system behavior. Even perfect algorithms run on imperfect conditions.

Contention is coherence serialization

If many cores contend on one cache line, the hardware turns your algorithm into a serialized stream of ownership transfers.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
```

```
.intel_syntax noprefix

/* Hot line: contended atomic update */
lock add DWORD PTR [hot_counter], 1
```

As cores increase, performance can degrade because the line becomes the bottleneck.

LL/SC has spurious failure and reservation granularity limits

LL/SC-based algorithms must tolerate failure not caused by logical contention (preemption, eviction, migration, granule conflicts).

```
/* Conceptual: SC may fail even without a conflicting write; retry is
 ↪  mandatory */
```

Atomics do not remove scheduling unfairness

Lock-free does not defeat:

- OS preemption and priority inversion,
- SMT sibling interference,
- NUMA latency and remote-line bouncing,
- interrupt storms and timer ticks.

A thread can be delayed indefinitely by the scheduler even if the algorithm is wait-free in theory, because “bounded steps” assumes the thread actually gets CPU time.

Single-word atomicity limits multi-word structures

Hardware commonly provides atomicity for a machine word (and some limited wider cases). Multi-field invariants require protocols (versioning, pointers with tags, or locks).

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Atomic for one location */
lock cmpxchg QWORD PTR [ptr], rbx

/* Not atomic for a pair of independent locations without a protocol
 ↪  */
mov     QWORD PTR [p1], rax
mov     QWORD PTR [p2], rdx
```

Practical hardware-aware rules

- Avoid single hot shared locations in scalable designs (shard or per-core state).
- Use backoff under contention; do not spin with full-speed retries forever.
- Prefer single-writer ownership and message passing when possible.
- When bounded latency is required, prove it and test under worst-case scheduling and contention.

Chapter 12

Hardware-Level Concurrency Bugs

12.1 Lost Updates

Lost update means two or more threads perform updates that should compose, but one update overwrites another. This is the most common hardware-visible concurrency failure and the easiest to reproduce under contention.

Cause

A read-modify-write expressed as **separate** operations is not indivisible:

- Thread A reads old value.
- Thread B reads the same old value.
- Both compute a new value.
- Both store it.

One update disappears.

Example: non-atomic increment loses updates

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Thread A and Thread B both execute this sequence concurrently */
mov     eax, DWORD PTR [counter]     /* read */
add     eax, 1                       /* modify */
mov     DWORD PTR [counter], eax     /* write */
```

If both threads read `counter==100`, both compute `101`, and both store `101`. Correct result should be `102`. One update is lost.

Correct fix: atomic RMW

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

lock add DWORD PTR [counter], 1
```

This makes the update indivisible for that location.

Example: check-then-set bug

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Buggy: two threads can both see 0 and both set 1, believing they
 ↪  acquired ownership */
mov     eax, DWORD PTR [flag]
```

```
test    eax, eax
jne     fail
mov     DWORD PTR [flag], 1
fail:
```

Correct fix: CAS

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

mov     eax, 0
mov     ebx, 1
lock cmpxchg DWORD PTR [flag], ebx   /* only one thread can
 ↪  transition 0->1 */
```

Lost updates checklist

- Any “read, compute, store” on shared state is suspect.
- If multiple writers exist, you need atomic RMW (CAS, LL/SC, or locked RMW) or a lock.
- Atomicity is per location: multi-location invariants need protocols.

12.2 Visibility Failures

Visibility failure means a thread reads a value that is stale relative to another thread’s write, even though the program logic assumes the write “must already be visible”.
This happens because:

- stores become globally visible later (store buffers),

- cache lines propagate asynchronously,
- and without an ordering edge, observing one write does not imply observing another.

Example: message passing bug (flag visible, data stale)

The algorithm intends: if consumer sees `flag==1`, it must see the updated `data`. Without proper ordering, the consumer can observe the flag but still see stale data (especially on weak models).

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Shared: data=0, flag=0 */

/* Producer */
mov     DWORD PTR [data], 777
mov     DWORD PTR [flag], 1

/* Consumer (naive) */
cmp     DWORD PTR [flag], 1
jne     not_ready
mov     eax, DWORD PTR [data]        /* may observe 0 on weak models
 ↪  without acquire/release */
not_ready:
```

Correct fix: publish/observe edge (conceptual)

- Producer: release-store the flag after writing data.
- Consumer: acquire-load the flag before reading data.

```
/* AArch64 shown conceptually */

/* Producer */
str     w0, [xData]
stlr    w1, [xFlag]                 /* release publish */

/* Consumer */
ldar    w2, [xFlag]                 /* acquire observe */
cbz     w2, not_ready
ldr     w3, [xData]
not_ready:
```

Example: stale reads from read-mostly caches are not a bug; missing ordering is

Coherence ensures eventual consistency for a location, but it does not guarantee your intended timing or cross-location implication. If your design depends on “when I see X, Y must be visible too”, that is a memory-ordering requirement.

Visibility checklist

- Do not use plain loads/stores as publication mechanisms.
- Make the publication point explicit (release) and the observation point explicit (acquire).
- Separate “atomicity for one location” from “visibility/order for related locations”.

12.3 Ordering Violations

Ordering violation means an outcome is observed that contradicts naive program-order assumptions. Nothing is actually "violated" by the hardware; your assumption was not part of the contract.
Ordering bugs appear as:

- "impossible" interleavings,
- reads seeing old values even after apparently observing related new values,
- two threads both reading 0 even though both stored 1.

Example: store buffering anomaly (both read 0)

Two threads store then load. Weak models allow both loads to read 0.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Shared: X=0, Y=0 */

/* Thread A */
mov     DWORD PTR [X], 1
mov     eax, DWORD PTR [Y]

/* Thread B */
mov     DWORD PTR [Y], 1
mov     ebx, DWORD PTR [X]
```

On weak ordering, `eax==0` and `ebx==0` is allowed because each store can be delayed in a store buffer while the load executes.

Fix: fence or stronger ordering (conceptual)

To forbid the "both read 0" outcome, you need to prevent the store→load reordering visibility.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Thread A (conceptual fix with full fence; often not minimal) */
mov     DWORD PTR [X], 1
mfence
mov     eax, DWORD PTR [Y]

/* Thread B */
mov     DWORD PTR [Y], 1
mfence
mov     ebx, DWORD PTR [X]
```

Example: observing writes in different orders across locations

Thread A writes `A` then `B`. Another thread reads `B` then `A`. Without a defined ordering edge, it can observe `B==1` but `A==0` on weak models.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Thread A */
mov     DWORD PTR [A], 1
mov     DWORD PTR [B], 1

/* Thread C */
mov     eax, DWORD PTR [B]        /* sees 1 */
```

```
mov     ebx, DWORD PTR [A]          /* sees 0 (allowed on weak models)
↪  */
```

Fix: use a single publication point

Publish the multi-write state behind one acquire/release edge (flag), or use versioning.

```
/* RISC-V shown conceptually */

/* Producer */
sw      t0, 0(a0)                   /* A = 1 */
sw      t1, 0(a1)                   /* B = 1 */
fence   rw, w                       /* release edge */
sw      t2, 0(a2)                   /* flag = 1 */

/* Consumer */
lw      t3, 0(a2)
beq     t3, x0, not_ready
fence   r, rw                       /* acquire edge */
lw      t4, 0(a0)                   /* read A */
lw      t5, 0(a1)                   /* read B */
not_ready:
```

Ordering checklist

- Identify the forbidden outcome (the anomaly).
- Identify which reordering permits it (compiler or hardware, or both).
- Apply the minimal primitive that forbids that outcome (acquire/release first; fences only when necessary).

- Re-check data layout: false sharing and contention can mask bugs in tests and amplify them in production.

Appendices

Appendix A — Instruction-Level Atomic Patterns

This appendix provides **instruction-level building blocks** that appear repeatedly in real concurrent systems. Each pattern is presented as:

- **Goal** (what invariant it enforces),
- **Mechanism** (the atomic primitive),
- **Notes** (contention and correctness pitfalls).

Atomic Increment and Decrement

Goal. Update a shared counter without lost updates. This is valid for:

- statistics counters,
- reference counts (with additional correctness constraints),
- ticket allocation and sequence numbers (often with relaxed ordering).

Atomic increment (x86-64 locked add)

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* counter++ : atomic RMW */
lock add DWORD PTR [counter], 1
```

Atomic decrement (x86-64 locked sub) and test-for-zero

A common use is reference counting: decrement and check if it reached zero. Correctness still requires memory ordering rules around object lifetime (beyond this snippet).

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* counter-- : atomic RMW */
lock sub DWORD PTR [counter], 1

/* Optional: observe zero using flags (ZF set if result==0) */
jz      became_zero
```

Fetch-add style (x86-64 xadd)

Goal. Obtain a unique old value and update atomically (IDs, tickets).

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

mov     eax, 1
lock xadd DWORD PTR [next_id], eax     /* eax := old; [next_id] :=
↪  old+1 */
```

Contention note

If many cores update the same counter, the cache line becomes a serialization point. Prefer sharding/per-core counters for scalability.

Spin-Based Synchronization

Spin synchronization is not “just a loop”. It is a coherence protocol interaction. Correct and scalable spins:

- avoid locked RMW on every iteration,
- reduce coherence traffic,
- and include backoff when contention is possible.

Pattern 1: naive spin (high traffic)

This pattern repeatedly reads a shared flag. It can saturate the interconnect.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

spin:
mov     eax, DWORD PTR [flag]
test    eax, eax
je      spin
```

Pattern 2: test-and-test-and-set (reduced RMW traffic)

Goal. Use shared reads while locked, perform an atomic RMW only when it looks free.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* lockv: 0 = free, 1 = held */
acquire:
mov     eax, DWORD PTR [lockv]          /* shared read */
test    eax, eax
jne     acquire                         /* still held */

mov     eax, 1
lock xchg DWORD PTR [lockv], eax        /* atomic acquire attempt */
test    eax, eax
jne     acquire                         /* if previous value != 0,
 ↪  someone else won */
```

Pattern 3: bounded retry with backoff (contention control)

Goal. Prevent livelock and reduce coherence storms.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Conceptual backoff loop (pause count may be tuned) */
acquire2:
mov     ecx, 64                          /* backoff counter */
try:
mov     eax, DWORD PTR [lockv]
test    eax, eax
jne     delay
```

```
mov     eax, 1
lock xchg DWORD PTR [lockv], eax
test    eax, eax
je      got_lock

delay:
pause
loop    delay
jmp     try

got_lock:
```

Release (simple store)

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

mov     DWORD PTR [lockv], 0
```

Correctness note. Release/acquire ordering is required for protected data visibility. The exact mechanism differs by ISA; the pattern above is the structural skeleton.

Atomic Flag Patterns

Flags are the simplest synchronization currency, but they fail if you confuse:

- atomicity of the flag itself,
- with publication/visibility of other data behind the flag.

Pattern 1: publish/observe (message passing skeleton)

Invariant. If consumer observes `flag==1`, it must observe the published data.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Producer */
mov     DWORD PTR [data], 777
mov     DWORD PTR [flag], 1             /* publication point
 ↪  (requires release semantics) */

/* Consumer */
mov     eax, DWORD PTR [flag]           /* observation point
 ↪  (requires acquire semantics) */
test    eax, eax
je      not_ready
mov     ebx, DWORD PTR [data]
not_ready:
```

Pattern 2: atomic state transition with CAS

Goal. One thread transitions state 0-¿1, others observe failure.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* if (*state == 0) *state = 1 */
mov     eax, 0
mov     ebx, 1
lock cmpxchg DWORD PTR [state], ebx     /* success if ZF=1 */
```

```
jnz     already_set
/* acquired state */
already_set:
```

Pattern 3: bit flag set/clear (atomic bit operations)

Goal. Manage multiple boolean flags packed into one word.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Set bit k */
lock bts DWORD PTR [flags], 5

/* Clear bit k */
lock btr DWORD PTR [flags], 5

/* Test bit k (value returned in CF) */
lock bt  DWORD PTR [flags], 5
```

Pitfall checklist for flags

- A flag on one cache line does not automatically publish data on another line.
- Busy-wait loops must avoid RMW-on-every-iteration under contention.
- If a flag is contended, isolate it on its own cache line to avoid false sharing.

Appendix B — Architectural Differences

This appendix summarizes the **practical ordering differences** that matter when you move the same algorithm between **x86-64**, **AArch64**, and **RISC-V**. The goal is not to memorize rules, but to map an invariant to the minimal hardware mechanism that enforces it.

x86 TSO Characteristics

x86-64 is relatively strong, commonly described by a TSO-like model:

- Loads are not reordered with other loads (R→R is preserved).
- Stores are not reordered with other stores (W→W is preserved).
- Loads are not reordered with older stores to the **same** location.
- The primary weak point is **Store→Load** visibility (W→R), which can appear reordered to other cores due to store buffers.
- Locked RMW operations form a strong serialization point for the involved line and are fully atomic.

Canonical anomaly: store buffering (W→R)

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Shared: X=0, Y=0 */

/* Thread A */
mov     DWORD PTR [X], 1
mov     eax, DWORD PTR [Y]
```

```
/* Thread B */
mov     DWORD PTR [Y], 1
mov     ebx, DWORD PTR [X]
```

On weaker models this permits `eax==0 && ebx==0`. On x86, the common intuition is "strong ordering", but the correct mental model is still: store buffers can delay global visibility of stores.

Practical mapping rules on x86

- Acquire/release message passing often maps to plain loads/stores in practice, but you must reason in acquire/release terms.
- When x86 needs explicit barriers, `mfence` (or a locked op) is the heavy tool.
- A locked RMW is correct but can be extremely costly under contention.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Strong fence */
mfence

/* Strong atomic RMW */
lock add DWORD PTR [counter], 1
```

ARM Weak Ordering

ARM (AArch64) is weakly ordered: many reorderings that are "unlikely" on x86 are normal and architecturally allowed. Key practical points:

- Loads and stores can be observed out of program order across cores unless constrained.
- Acquire and release are first-class primitives: `LDAR` (acquire load) and `STLR` (release store).
- Barriers such as `DMB` are used to enforce ordering when acquire/release instructions are insufficient.

Why ARM forces discipline: message passing needs explicit edges

```
/* AArch64 shown conceptually */

/* Producer: publish data then flag */
str     w0, [xData]
stlr    w1, [xFlag]          /* release publish */

/* Consumer: observe flag then data */
ldar    w2, [xFlag]          /* acquire observe */
cbz     w2, not_ready
ldr     w3, [xData]
not_ready:
```

Barrier when needed (ordering multiple operations)

```
/* AArch64 shown conceptually */

/* Full ordering within a shareability domain */
dmb     ish
```

Practical mapping rules on ARM

- Never assume a plain flag load/store publishes other data.

- Prefer `LDAR/STLR` (acquire/release) over full barriers for publication.
- Use `DMB` when you need ordering beyond a single acquire/release edge (e.g., complex protocols).

RISC-V Memory Model

RISC-V uses a weak model (RVWMO) where ordering must be expressed explicitly using:

- **FENCE** instructions to order reads/writes before and after,
- and (in many designs) **acquire/release annotations** on AMOs and LR/SC to express synchronization edges.

Fence as an explicit ordering statement

RISC-V fences are written as "before, after" sets:

- `fence rw, w` is commonly used as a release-like ordering step.
- `fence r, rw` is commonly used as an acquire-like ordering step.
- `fence rw, rw` is full-like ordering.

```
/* RISC-V shown conceptually */

fence   rw, w      /* release-like */
fence   r, rw      /* acquire-like */
fence   rw, rw     /* full-like */
```

Message passing with fences (conceptual)

```
/* RISC-V shown conceptually */

/* Producer */
sw      t0, 0(a0)           /* *data = t0 */
fence   rw, w               /* release */
sw      t1, 0(a1)           /* *flag = 1 */

/* Consumer */
lw      t2, 0(a1)
beq     t2, x0, not_ready
fence   r, rw               /* acquire */
lw      t3, 0(a0)           /* read *data */
not_ready:
```

Practical mapping rules on RISC-V

- Treat ordering as explicit: if you did not write a fence or use acquire/release AMOs, you likely did not order anything.
- Minimize fences: use the weakest fence that forbids the unwanted outcome.
- Watch granularity and false sharing: contention on one line dominates performance regardless of ordering choice.

Cross-architecture summary (what to remember)

- x86: stronger defaults, but do not confuse that with “no reordering”; store buffers exist and RMW contention is costly.

- ARM: weak ordering; use acquire/release instructions as the standard publication mechanism.
- RISC-V: weak ordering; fences (and/or acquire/release annotated atomics) are how you state the contract.

Appendix C — Practical Rules of Thumb

This appendix is a **decision guide**. The goal is to choose the **weakest correct mechanism** that proves your invariant:

- **Relaxed atomics** for pure atomicity (no lost updates / no tearing) without publication.
- **Acquire/Release** for publication and synchronization edges.
- **Fences** only when you must order multiple operations beyond what acquire/release provides.
- **Locks** when invariants are multi-location, complex, or require bounded behavior under contention.

When Atomics Are Sufficient

Atomics are sufficient when the invariant can be expressed as:

- **Single-location atomicity**: no torn reads/writes for that location, and/or no lost updates for that location.
- **Single edge synchronization**: a clear publish/observe relation can be expressed with acquire/release.
- **Monotonic or commutative updates**: counters, IDs, bitsets, reference counts (with correct lifetime rules).

Case 1: counters and statistics (relaxed or equivalent)

If correctness does not require ordering with other data, use relaxed-like atomic RMW.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Atomic counter++ : ordering with other memory is irrelevant */
lock add DWORD PTR [stats_counter], 1
```

Case 2: unique ticket / ID allocation (fetch-add)

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

mov     eax, 1
lock xadd DWORD PTR [next_ticket], eax   /* eax := old ticket; memory
 ↪  := old+1 */
```

Case 3: one-bit state transitions (CAS)

Use CAS when exactly one thread should win a transition (0-¿1).

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

mov     eax, 0
mov     ebx, 1
lock cmpxchg DWORD PTR [state], ebx
jnz     lost_race
/* won transition */
lost_race:
```

Case 4: publication behind a single flag (acquire/release edge)

Atomics are sufficient when a **single release** publish is paired with a **single acquire** observe.

```
/* AArch64 shown conceptually */

/* Producer: write payload then release-publish flag */
str     w0, [xData]
stlr    w1, [xFlag]

/* Consumer: acquire-observe flag then read payload */
ldar    w2, [xFlag]
cbz     w2, not_ready
ldr     w3, [xData]
not_ready:
```

Atomic-sufficient checklist

- The shared state you update is a **single machine word** (or supported atomic width).
- You need either:
  - atomicity only (relaxed), or
  - one clear publish/observe edge (acquire/release).
- You are not trying to update multiple independent locations as one atomic action.

When Fences Are Required

Fences are required when acquire/release on a single operation is not enough to express your intended ordering. Use a fence only after you identify the forbidden outcome and the reordering that permits it.

Case 1: you must order multiple operations with one barrier

If you use plain loads/stores for data and then publish via a separate mechanism, some ISAs require an explicit fence.

```
/* RISC-V shown conceptually */

/* Producer: multiple writes, then a release fence, then publish */
sw      t0, 0(a0)           /* data1 */
sw      t1, 0(a1)           /* data2 */
fence   rw, w               /* release fence */
sw      t2, 0(a2)           /* flag=1 */

/* Consumer: observe, then acquire fence, then consume */
lw      t3, 0(a2)
beq     t3, x0, not_ready
fence   r, rw               /* acquire fence */
lw      t4, 0(a0)
lw      t5, 0(a1)
not_ready:
```

Case 2: you must forbid a specific litmus outcome (store-buffering style)

To forbid "both read 0" outcomes (where allowed), you need to prevent store→load reordering visibility.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Conceptual: full fence forbids store->load visibility reordering
 ↪  (heavy tool) */
```

```
mov     DWORD PTR [X], 1
mfence
mov     eax, DWORD PTR [Y]
```

Case 3: you need ordering but cannot use acquire/release forms

Some low-level interfaces or hand-written assembly paths may require an explicit fence instruction.

```
/* AArch64 shown conceptually */

/* Order prior data writes before later publication store */
dmb     ish
```

Fence-required checklist

- You need to order **a set** of operations (not just one load or one store).
- The ISA does not provide the needed ordering with the primitives you are using (plain loads/stores, non-annotated ops).
- You can name the **forbidden outcome** and show that the fence forbids it.

When Locks Are Inevitable

Locks are inevitable when:

- the invariant spans multiple independent memory locations,
- the operation must appear atomic as a **transaction**,
- fairness / bounded waiting is required,
- or the algorithm becomes too complex and fragile under contention.

Case 1: multi-field invariants (transactional update)

If you must update `a`, `b`, and `c` together such that readers never see mixed states, atomics alone are usually insufficient without a complex protocol (seqlock/versioning/RCU/hazard pointers).

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Without a protocol, another core can observe partial updates */
mov     DWORD PTR [a], 1
mov     DWORD PTR [b], 2
mov     DWORD PTR [c], 3
```

A lock provides the simplest correct atomicity boundary for such multi-location updates.

Case 2: memory reclamation with pointer CAS (ABA + lifetime)

If you CAS pointers and reclaim nodes, you must solve lifetime safety (ABA, use-after-free). Solutions exist (hazard pointers, epochs), but the complexity is real. Locks are often the pragmatic choice.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Pointer CAS alone cannot guarantee safe reclamation */
lock cmpxchg QWORD PTR [head], rbx
```

Case 3: bounded latency / fairness requirements

Lock-free often permits starvation. If you require bounded waiting, choose a lock or a wait-free design. In production, locks are frequently the simplest correct way to guarantee bounded

behavior.

Case 4: contention collapses performance

A contended atomic hot spot can serialize the whole program. A lock can sometimes outperform naive lock-free designs by reducing coherence storms and providing structured waiting.

```
/* x86-64 (GAS) shown with Intel syntax enabled */
.intel_syntax noprefix

/* Hot contended atomic update (can collapse scaling) */
lock add DWORD PTR [hot_counter], 1
```

Lock-inevitable checklist

- You need multi-location atomicity or complex invariants.
- You need bounded progress or fairness.
- You require safe memory reclamation under concurrent mutation and cannot afford protocol complexity.
- Your lock-free approach becomes a coherence storm under real contention.

References

CPU Architecture Manuals

These manuals define the **architectural contract** between software and hardware: instruction semantics, atomicity guarantees, cache coherency behavior, and ordering rules. They are the primary source of truth for what the CPU *must* guarantee, independent of compilers.

- **x86-64 Architecture Manuals**
  - Definition of atomic instructions (`LOCK`-prefixed RMW, `CMPXCHG`, `XADD`).
  - Total Store Order (TSO) properties and the precise role of store buffers.
  - Fence instructions (`LFENCE`, `SFENCE`, `MFENCE`) and their ordering scope.
  - Cache coherency guarantees and single-copy atomicity for aligned accesses.
- **ARM Architecture Reference Manuals (AArch64)**
  - Weakly ordered memory model with explicit acquire/release semantics.
  - Load-acquire (`LDAR`) and store-release (`STLR`) instructions.
  - Barrier instructions (`DMB`, `DSB`) and shareability domains.
  - LL/SC behavior and reservation granularity constraints.
- **RISC-V Unprivileged and Privileged Specifications**

  - RVWMO memory model and explicit ordering requirements.
  - `FENCE` instruction semantics (before/after read/write sets).
  - Atomic Memory Operations (AMOs) and LR/SC guarantees.
  - Architectural separation between ISA semantics and microarchitectural freedom.

Rule: If a behavior is not guaranteed by the architecture manual, it is not guaranteed at all.

Memory Model Specifications

Memory model specifications define **which executions are allowed** when multiple threads interact. They formalize ordering, visibility, and atomicity beyond informal intuition.

- **x86 TSO Formal Models**
  - Formalization of store-buffering behavior.
  - Guarantees on load-load and store-store ordering.
  - Explicit allowance of store-to-load visibility reordering.
- **ARM Memory Model (AArch64)**
  - Explicit modeling of weak ordering and propagation delays.
  - Acquire/release edges as first-class ordering constructs.
  - Litmus-test–based validation of allowed and forbidden outcomes.
- **RISC-V Memory Model (RVWMO)**
  - Minimal guarantees by default, explicit ordering by fences.
  - Separation between atomicity and ordering.
  - Formal treatment of fence scopes and AMO annotations.

- **Language-Level Memory Models (Conceptual Alignment)**
  - Mapping of hardware ordering to language constructs (SC, acquire/release, relaxed).
  - Distinction between atomicity of a location and visibility across locations.

Rule: Reason about concurrency by identifying *forbidden outcomes*, not by assuming intuitive interleavings.

Compiler and Toolchain Documentation

Compilers are active participants in concurrency. They may reorder, eliminate, or duplicate memory operations unless explicitly constrained.

- **Compiler Memory Models**
  - As-if rule: preservation of single-thread semantics only.
  - Allowed reordering of non-atomic loads and stores.
  - Interaction between compiler barriers and hardware fences.
- **Atomic Code Generation**
  - Mapping of language-level atomics to ISA instructions.
  - When acquire/release lowers to plain loads/stores vs explicit barriers.
  - Use of locked instructions or LL/SC loops for RMW operations.
- **Inline Assembly Constraints**
  - Memory clobbers as compiler ordering barriers.
  - Difference between preventing compiler motion and enforcing hardware visibility.

  - Correct expression of dependencies in inline assembly.

- **Toolchain Guarantees**

  - Optimizer assumptions under undefined behavior and data races.
  - Effects of link-time optimization on memory ordering.

Rule: A hardware fence that the compiler does not understand is not a fence.

Academic Concurrency Literature

Academic work provides the **formal vocabulary** and proof techniques used to reason about concurrency. Much of what is considered "folklore" today originated here.

- **Formal Memory Models**

  - Axiomatic and operational models for weak memory.
  - Litmus tests as minimal counterexamples.
  - Happens-before and synchronization relations.

- **Lock-Free and Wait-Free Algorithms**

  - Definitions of progress guarantees and liveness.
  - CAS- and LL/SC-based algorithm construction.
  - Starvation, livelock, and contention analysis.

- **ABA and Memory Reclamation**

  - Hazard pointers, epoch-based reclamation, reference counting.
  - Proof obligations for safe reclamation under concurrency.

- **Performance and Scalability Studies**
  - False sharing and cache-line contention analysis.
  - Impact of fences and atomic hot spots on throughput.
  - Empirical studies on real multicore systems.

Rule: If an algorithm has no proof or formal argument, it is an experiment—not a guarantee.