Memory Visibility vs. Execution Ordering

The Core Confusion

Most engineers conflate two distinct concepts:

1. **Execution ordering:** The sequence in which operations complete
2. **Memory visibility:** When one CPU sees effects of another CPU's operations

Critical insight: Operations can execute in order but still have visibility problems.

What is Memory Visibility?

Memory visibility is about **when writes by one goroutine become observable by reads in another goroutine.**

```
var a int

// CPU 1 (Goroutine 1)
a = 42  // Write to L1 cache

// CPU 2 (Goroutine 2)
print(a)  // Read from L1 cache

// Question: Does CPU 2 see a=42?
// Answer: Maybe. Depends on cache coherency protocol and memory barriers.
```

Without synchronization, CPU 2 might read stale value from its cache indefinitely.

What is Execution Ordering?

Execution ordering is about **the sequence in which instructions execute within a goroutine.**

```
// Source code order
a := 1  // (1)
b := 2  // (2)
c := a + b  // (3)

// Compiler can reorder to:
b := 2  // (2)
a := 1  // (1)
c := a + b  // (3)

// Why? (1) and (2) are independent.
// Observable behavior within goroutine unchanged.
```

Important: Reordering is legal if single-goroutine semantics are preserved.

Why CPUs and Compilers Reorder

Compiler Reordering

Compilers optimize code by reordering instructions.

Example: Register allocation

```
// Source
a = 1
b = 2
c = a

// Compiler sees: c depends on a, not b.
// Optimized:
a = 1
c = a  // Use a immediately (a might be in register)
b = 2  // Delay b assignment
```

Rule: Compiler preserves **as-if-serial semantics** (behavior as observed within goroutine).

Problem: Other goroutines can see reordered effects.

CPU Reordering (Store Buffer)

Modern CPUs use **store buffers** to improve write performance.

What happens on write:

1. CPU writes to store buffer (fast, ~1 cycle)
2. Value eventually written to L1 cache (~10 cycles)
3. Cache line eventually synced to other CPUs (~100+ cycles)

Implication: Writes are not immediately visible to other CPUs.

```
// CPU 1
WRITE a = 1   // Enter store buffer
WRITE b = 1   // Enter store buffer

// CPU 2 (simultaneously)
READ b        // Might see b=1 (if flushed from store buffer)
READ a        // Might see a=0 (if not yet flushed)

// Result: See b=1 but a=0 (reordered from perspective of CPU 2)
```

CPU Reordering (Out-of-Order Execution)

CPUs execute instructions out-of-order for pipeline efficiency.

```
// Instructions
LOAD  R1, [x]   // (1) Memory load (slow, ~100 cycles)
LOAD  R2, [y]   // (2) Memory load (slow)
ADD   R3, R1, R2  // (3) Depends on (1) and (2)
```

```
// CPU might execute:
(2) before (1) to parallelize memory loads
```

If another CPU reads intermediate states, sees reordered effects.

The Four Types of Memory Reordering

CPUs can reorder memory operations in four ways:

Reordering Type	Description	Example
Store-Store	Write B before Write A	a=1; b=2 → b=2; a=1
Load-Load	Read B before Read A	x=a; y=b → y=b; x=a
Load-Store	Write B before Read A	x=a; b=2 → b=2; x=a
Store-Load	Read B before Write A	a=1; x=b → x=b; a=1

x86/x64 guarantees:

- ✓ Store-Store: No reordering
- ✓ Load-Load: No reordering
- ✓ Load-Store: No reordering
- ✗ Store-Load: **CAN reorder** (most expensive to prevent)

ARM/ARM64:

- ✗ **All four can reorder** (weakly ordered)

Go memory model abstracts over architectures: Assumes weak ordering (like ARM). Code correct on ARM works on x86.

Example: Dekker's Algorithm (Classic Failure)

Dekker's algorithm is a mutual exclusion algorithm without locks.

```go
var flag1, flag2 int

// Goroutine 1
flag1 = 1         // (1) "I want to enter"
if flag2 == 0 {   // (2) "Is other goroutine out?"
    // Critical section
}

// Goroutine 2
flag2 = 1         // (3) "I want to enter"
if flag1 == 0 {   // (4) "Is other goroutine out?"
    // Critical section
}
```

Intended behavior: At least one sees the other's flag.

With reordering:

Time	G1	G2
1	flag2 == 0 (read)	flag1 == 0 (read)
2	flag1 = 1 (write)	flag2 = 1 (write)

Both reads before both writes (load-load and store-store reordering).

Result: Both goroutines enter critical section → race!

Why it fails: No happens-before. Compiler/CPU can reorder independent loads and stores.

Example: Write Buffering Bug

```
var a, b int

// Goroutine 1
a = 1  // (1)
b = 1  // (2)

// Goroutine 2
for b == 0 {}  // (3) Spin until b=1
print(a)       // (4)

// Expected: G2 sees b=1, then a must be 1
// Reality: Can print a=0
```

Execution timeline:

1. CPU 1 executes (1): a=1 enters store buffer
2. CPU 1 executes (2): b=1 enters store buffer
3. **b=1 flushes to memory first** (arbitrary order)
4. CPU 2 sees b=1, exits loop
5. CPU 2 reads a (still 0 in CPU 2's cache, a=1 not yet visible)

Result: G2 sees b=1 but a=0 (reordered visibility).

Memory Barriers (How Synchronization Works)

Memory barrier: CPU instruction that enforces ordering.

Types of Barriers

1. **Store barrier (SFENCE on x86):** All stores before barrier complete before stores after barrier
2. **Load barrier (LFENCE on x86):** All loads before barrier complete before loads after barrier
3. **Full barrier (MFENCE on x86):** All memory ops before barrier complete before ops after barrier

Go synchronization primitives insert barriers automatically.

How Mutex Inserts Barriers

```
mu.Lock()      // Inserts ACQUIRE barrier
a = 1          // Protected write
mu.Unlock()    // Inserts RELEASE barrier
```

ACQUIRE barrier (Lock):

- Prevents operations AFTER Lock from moving BEFORE Lock
- Ensures you see all writes from previous Unlock

RELEASE barrier (Unlock):

- Prevents operations BEFORE Unlock from moving AFTER Unlock
- Ensures your writes visible to next Lock

Result: Lock/Unlock create happens-before.

How Channel Send Inserts Barriers

```
a = 1      // (1)
ch <- 0    // (2) RELEASE barrier

<-ch       // (3) ACQUIRE barrier
print(a)   // (4)
```

RELEASE (send): Flushes store buffer, ensures (1) visible before (2) completes.
ACQUIRE (receive): Ensures all writes before send are visible after receive.

Result: (1) →hb (4)

Example: Double-Checked Locking (Why It Fails)

```
type Singleton struct {
    instance *Instance
    mu       sync.Mutex
}

func (s *Singleton) Instance() *Instance {
    if s.instance != nil {  // (1) Read without lock
        return s.instance   // (2) Return
    }

    s.mu.Lock()
    if s.instance == nil {
        s.instance = &Instance{  // (3) Write
            Field1: value1,      // (4)
            Field2: value2,      // (5)
        }
    }
    s.mu.Unlock()  // (6) RELEASE barrier
```

```go
    return s.instance
}
```

The bug:

Goroutine A: (3) writes pointer, (4)(5) initialize fields, (6) unlocks.
Goroutine B: (1) reads pointer (not protected by lock).

Problem: Even if B sees pointer non-nil, no happens-before between (4)(5) and (1).

Reordering scenario:

1. A allocates memory for Instance
2. A writes pointer to s.instance (pointer now non-nil) — (3)
3. **B reads pointer at (1), sees non-nil**
4. **B returns partially constructed instance** — (2)
5. A writes Field1 — (4)
6. A writes Field2 — (5)

Result: B uses instance with zero-value fields → crash or incorrect behavior.

Why? No ACQUIRE barrier at (1). Compiler/CPU can reorder (3) to be visible before (4)(5).

Visibility vs. Ordering in Practice

Case 1: Sequential Consistency (Strong Guarantee)

Sequential consistency: All operations appear in some global order consistent with each goroutine's program order.

Go does NOT guarantee sequential consistency (too expensive).

Example that violates sequential consistency:

```go
var a, b, x, y int

// Goroutine 1
a = 1
x = b

// Goroutine 2
b = 1
y = a

// Possible: x=0 and y=0
// Violates sequential consistency
```

If sequentially consistent, one of these must be true:

- G1's writes before G2's: x=0, y=1
- G2's writes before G1's: x=1, y=0
- Interleaved: x=1, y=1 or x=0, y=1 or x=1, y=0

But x=0, y=0 is impossible with sequential consistency.

With reordering:

- G1 reads b before writing a (load-store reorder)
- G2 reads a before writing b (load-store reorder)
- Result: Both reads see 0

This is legal in Go's memory model (no synchronization).

Case 2: Release-Acquire Ordering (What Go Provides)

Go's synchronization primitives guarantee release-acquire ordering:

- **Release:** Stores before Unlock/Send/Done visible to acquirer
- **Acquire:** Acquirer sees all stores from releaser

This is weaker than sequential consistency but sufficient for correctness.

```
var a, b int
var mu sync.Mutex

// Goroutine 1
mu.Lock()
a = 1
b = 1
mu.Unlock()  // RELEASE

// Goroutine 2
mu.Lock()    // ACQUIRE
print(a, b)  // Sees a=1, b=1
mu.Unlock()
```

Real-World Failure: Linux Kernel RCU Bug (2006)

RCU (Read-Copy-Update): Lock-free data structure in Linux kernel.

Bug: Missing memory barrier in RCU update path.

```
// Simplified
void publish(node *new_node) {
    new_node->data = value;  // (1)
    list->head = new_node;   // (2) Publish
}

node *read() {
    node *n = list->head;    // (3) Read pointer
    return n->data;          // (4) Read data
}
```

Without barrier between (1) and (2): CPU can make (2) visible before (1).

Result: Reader sees pointer to uninitialized node → crash.

Fix: Insert `smp_wmb()` (write memory barrier) between (1) and (2).

```
new_node->data = value;  // (1)
smp_wmb();               // Write barrier
list->head = new_node;   // (2)
```

In Go: This would be prevented by using channels or mutexes (automatic barriers).

Real-World Failure: Java HashMap Race (Common)

Java's HashMap is not thread-safe. Concurrent access without synchronization causes visibility problems.

```
// Thread 1
map.put("key", "value");  // Writes to internal array

// Thread 2
String v = map.get("key");  // Reads internal array
// Can return null even after put() completed
```

Why? No happens-before. Thread 2's CPU cache might not see Thread 1's writes.

Same in Go with non-concurrent map:

```
m := make(map[string]int)

go func() {
    m["key"] = 42  // Write
}()

go func() {
    v := m["key"]  // Read - can see 0 or panic
}()
```

Go detects this with race detector:

```
WARNING: DATA RACE
Write by goroutine 1
Read by goroutine 2
```

How to Reason About Visibility

Rule 1: Synchronization Provides Visibility

If operations are synchronized (happens-before), writes are visible.

```
var a int
var mu sync.Mutex

// Writer
mu.Lock()
a = 1
mu.Unlock()  // RELEASE: a written to memory
```

```go
// Reader
mu.Lock()    // ACQUIRE: a read from memory
v := a       // Sees a=1
mu.Unlock()
```

Rule 2: Without Synchronization, No Visibility Guarantee

```go
var a int

// Writer
a = 1  // Might stay in CPU cache

// Reader
v := a  // Might read from different CPU's cache (stale value)
```

Rule 3: Atomics Provide Visibility

```go
var a int
var flag int32

// Writer
a = 1
atomic.StoreInt32(&flag, 1)  // RELEASE: a visible

// Reader
for atomic.LoadInt32(&flag) == 0 {}  // ACQUIRE
v := a  // Sees a=1
```

Performance Implications

Memory barriers are expensive:

Operation	x86 Latency	ARM Latency
Register read	~1 cycle	~1 cycle
L1 cache read	~4 cycles	~4 cycles
Store buffer flush	~10 cycles	~10 cycles
Memory barrier	~20-100 cycles	~50-200 cycles
Cache line sync	~100-300 cycles	~100-500 cycles

Why slow? Barrier flushes pipelines, waits for pending memory ops, ensures cache coherency.

Implication: Minimize synchronization in hot paths.

Interview Traps

Trap 1: "Reads are free, only writes need synchronization"

Wrong. Both reads and writes need synchronization to ensure visibility.

Correct: "Without synchronization, reads might see stale values indefinitely due to CPU caching. Both directions (write→read and read→write) need happens-before."

Trap 2: "volatile keyword in other languages is like atomic"

Wrong. Volatile in C/C++/Java only prevents compiler reordering, not CPU reordering (mostly).

Go has no volatile. Use sync/atomic for visibility guarantees.

Correct: "In Go, use sync/atomic for atomic operations with memory visibility guarantees. sync/atomic operations include necessary memory barriers to ensure visibility across goroutines."

Trap 3: "If execution order is A then B, reads see A before B"

Wrong. Execution order within a goroutine doesn't guarantee visibility order in another goroutine.

Correct: "Execution order (program order) is only guaranteed within a single goroutine. For visibility across goroutines, must establish happens-before through synchronization. CPUs can have different views of memory due to caching."

Trap 4: "x86 has strong memory model, so my code is safe"

Wrong. Code might work on x86 but fail on ARM.

Correct: "Go's memory model assumes weak ordering (like ARM). Code must be correct under weak ordering to be portable. Never rely on architecture-specific behavior. Use race detector to find such bugs."

Key Takeaways

1. **Visibility ≠ Ordering:** Operations can execute in order but have visibility problems
2. **CPUs cache memory:** Without barriers, writes stay in cache
3. **CPUs reorder operations:** For performance, within as-if-serial constraint
4. **Compilers reorder operations:** Within single-goroutine semantics
5. **Synchronization inserts barriers:** Lock/Channel/Atomic enforce ordering and visibility
6. **Go assumes weak memory model:** Correct code works on x86 and ARM
7. **Always use race detector:** Catches visibility bugs

What You Should Be Thinking Now

- "How do I know if I need synchronization?"
- "What are the common misconceptions about memory?"
- "Why do tests pass but production fails?"

Next: common-misconceptions.md - Debunking dangerous myths about memory and concurrency.

Exercises

1. Explain why this can print "y=1, x=0":

```go
var x, y int
go func() { x=1; y=1 }()
go func() { if y==1 { print(x) } }()
```

2. Draw the store buffer timeline for:

```go
a=1; b=1
// Other goroutine reads b then a
```

3. Explain why double-checked locking fails using visibility arguments (not ordering).

4. What memory barriers does this insert?

```go
mu.Lock()
data = value
mu.Unlock()
```

5. Why does x86 code sometimes work without barriers but fail on ARM?

Don't continue until you can: "Distinguish between execution ordering and memory visibility in concurrent code."