The Go Memory Model

What is a Memory Model?

A **memory model** defines the conditions under which one goroutine's writes to a variable are guaranteed to be visible to reads from another goroutine.

In simpler terms: The memory model tells you: "When does goroutine B see the effects of what goroutine A did?"

Without this specification, concurrent programs are undefined.

Why You Need to Understand This

```
var x, y int

// Goroutine A
x = 1
y = 1

// Goroutine B
if y == 1 {
    fmt.Println(x)  // Can this print 0? YES!
}
```

Shocking truth: Without synchronization, goroutine B can see `y==1` but `x==0`.

Why? Compiler and CPU can reorder operations. Memory model defines when reordering is forbidden.

The Central Concept: Happens-Before

Happens-before is a partial ordering of memory operations. If event A happens-before event B, then:

1. A completes before B starts (temporal ordering)
2. A's effects are visible to B (memory visibility)

Without happens-before, there's NO GUARANTEE of visibility.

```
// Notation: A →hb B means "A happens-before B"

write(x) →hb read(x)  // Read sees write
```

The Fundamental Guarantee (Sequential Consistency Within Goroutine)

Within a single goroutine, operations execute in program order:

```
func example() {
    x := 1  // (1)
```

```
    y := 2  // (2)
    z := x + y  // (3) Sees x=1, y=2
}

// Guaranteed: (1) →hb (2) →hb (3)
```

But this guarantee does NOT extend across goroutines without synchronization.

The Problem: No Happens-Before Across Goroutines

```
var a, b int

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

// Goroutine 2
if b == 2 {  // (3)
    print(a)  // (4)
}
```

Within G1: (1) →hb (2)
Within G2: (3) →hb (4)
Across goroutines: NO RELATIONSHIP

Possible: (2) →hb (3) →hb (4) but (1) NOT visible to (4)
Result: Print 0, even though G2 sees b==2

This is not a bug. This is how CPUs and compilers work.

Synchronization Operations That Create Happens-Before

Go provides specific operations that establish happens-before relationships across goroutines.

1. Goroutine Creation

```
var a string

func f() {
    print(a)  // (2)
}

func main() {
    a = "hello"  // (1)
    go f()       // Start goroutine
}

// Guarantee: (1) →hb (2)
// "go f()" synchronizes: operations before "go" visible inside f()
```

Rule: Everything before `go` happens-before the start of the goroutine.

2. Goroutine Exit (No Guarantee!)

```
var a string

func f() {
    a = "hello"  // (1)
}

func main() {
    go f()
    print(a)  // (2)
}

// NO GUARANTEE: (1) NOT →hb (2)
// "a" might be "", might be "hello"
```

Rule: Goroutine exit is NOT synchronized with parent. Use channels or sync primitives.

3. Channel Communication

Unbuffered channel:

```
var c = make(chan int)
var a string

// Goroutine 1
func sender() {
    a = "hello"  // (1)
    c <- 0       // (2) Send
}

// Goroutine 2
func receiver() {
    <-c          // (3) Receive
    print(a)     // (4)
}

// Guarantees:
// (1) →hb (2) (same goroutine)
// (2) →hb (3) (send →hb receive)
// (3) →hb (4) (same goroutine)
// Chain: (1) →hb (4)
// Result: Always prints "hello"
```

Rule (Unbuffered): Send on channel happens-before the corresponding receive completes.

Buffered channel:

```
var c = make(chan int, 1)
var a string

// Goroutine 1
a = "hello"  // (1)
c <- 0       // (2)

// Goroutine 2
<-c          // (3)
print(a)     // (4)

// Guarantees:
// (2) →hb (3) (send →hb receive)
// Does (1) →hb (4)? YES, because:
// (1) →hb (2) →hb (3) →hb (4)
```

Rule (Buffered): Send on channel happens-before receive from that channel **begins**.

Channel close:

```
var c = make(chan int)
var a string

// Goroutine 1
a = "hello"  // (1)
close(c)     // (2)

// Goroutine 2
<-c          // (3) Receive from closed channel
print(a)     // (4)

// Guarantee: (2) →hb (3)
// Chain: (1) →hb (2) →hb (3) →hb (4)
```

Rule: Close of channel happens-before receive of zero value from closed channel.

4. Locks (Mutex, RWMutex)

```
var mu sync.Mutex
var a string

// Goroutine 1
func writer() {
    mu.Lock()
    a = "hello"  // (1)
    mu.Unlock()  // (2)
}

// Goroutine 2
func reader() {
```

```
    mu.Lock()    // (3)
    print(a)     // (4)
    mu.Unlock()
}

// Guarantee: (2) →hb (3)
// Chain: (1) →hb (2) → hb (3) →hb (4)
```

Rule: Unlock of mutex M happens-before any subsequent Lock of M.

RWMutex:

- RUnlock happens-before Lock (reader → writer)
- Unlock happens-before Lock (writer → writer)
- Unlock happens-before RLock (writer → reader)

5. sync.WaitGroup

```
var wg sync.WaitGroup
var a string

func worker() {
    a = "hello"  // (1)
    wg.Done()    // (2)
}

func main() {
    wg.Add(1)
    go worker()
    wg.Wait()    // (3)
    print(a)     // (4)
}

// Guarantee: (2) →hb (3)
// Chain: (1) →hb (2) →hb (3) →hb (4)
```

Rule: Done() happens-before Wait() returns (if it caused counter → 0).

6. sync/atomic Operations

```
var a int
var flag int32  // atomic

// Goroutine 1
a = 42                          // (1)
atomic.StoreInt32(&flag, 1)     // (2)

// Goroutine 2
for atomic.LoadInt32(&flag) == 0 {}  // (3) Spin until flag set
print(a)                             // (4)
```

```
// Guarantee: (2) →hb (3)
// Chain: (1) →hb (2) →hb (3) →hb (4)
```

Rule: Atomic write happens-before atomic read of the same variable that observes the write.

7. sync.Once

```
var a string
var once sync.Once

func setup() {
    a = "hello"  // (1)
}

// Multiple goroutines
once.Do(setup)  // (2)
print(a)        // (3)

// Guarantee: (1) →hb (3)
// setup() completes before once.Do() returns
```

Rule: once.Do(f) call that executes f() happens-before any once.Do(g) returns.

Operations That DO NOT Provide Happens-Before

1. Time

```
// WRONG: Assuming time provides ordering
func goroutine1() {
    a = 1
}

func goroutine2() {
    time.Sleep(time.Second)  // "Let goroutine1 finish"
    print(a)  // Might still see 0!
}
```

No guarantee. Time is not synchronization.

2. Goroutine Yield/Gosched

```
// WRONG
a = 1
runtime.Gosched()  // Yield to scheduler
```

No guarantee. Yielding doesn't synchronize.

3. Finalizers

```
// WRONG: Using finalizer for synchronization
runtime.SetFinalizer(obj, func(o *Object) {
    // Runs when obj is garbage collected
    // NO synchronization with main program
})
```

4. Defer, Panic, Recover

```
// WRONG: Assuming defer creates happens-before
func main() {
    defer print(a)
    go func() { a = 1 }()
}
// No guarantee defer sees a=1
```

Example: The Double-Checked Locking Bug

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

// WRONG: Broken double-checked locking
func (s *Singleton) Instance() *Instance {
    if s.instance != nil {  // (1) Read without lock
        return s.instance
    }

    s.mu.Lock()
    defer s.mu.Unlock()

    if s.instance == nil {
        s.instance = &Instance{}  // (2) Write
    }

    return s.instance
}

// Problem: (1) and (2) are not synchronized
// Goroutine A: Creates instance, releases lock
// Goroutine B: Sees s.instance != nil (pointer is non-nil)
//              But instance FIELDS might not be visible!
//              Using partially-constructed instance → crash
```

Fix: Use sync.Once

```
var (
    instance *Instance
```

```
    once     sync.Once
)

func Instance() *Instance {
    once.Do(func() {
        instance = &Instance{}
    })
    return instance  // Guaranteed fully constructed
}
```

Example: The Init Function Trap

```
var a string

func init() {
    a = "hello"
}

func main() {
    go func() {
        print(a)  // Guaranteed "hello"? YES
    }()
}
```

Guarantee: Package init functions happen-before main.main starts.
Chain: init writes a →hb main starts →hb go f() →hb print(a)

Common Misconceptions

Misconception 1: "Volatile" Reads and Writes

Go does not have "volatile" keyword like C/C++/Java.

Wrong assumption: "Regular reads/writes are atomic and visible."
Truth: Without synchronization, visibility is not guaranteed.

Misconception 2: "Reads Can't See Partial Writes"

```
type T struct {
    x, y int
}

var t T

// Goroutine 1
t = T{1, 2}

// Goroutine 2
fmt.Println(t)  // Can print {1, 0}, {0, 2}, or other partial state
```

Without synchronization, non-atomic writes can be observed partially.

Misconception 3: "If Tests Pass, It's Correct"

```
var x, y int

go func() {
    x = 1
    y = 1
}()

time.Sleep(time.Millisecond)  // "Wait"
if y == 1 && x == 0 {
    panic("impossible!")  // This CAN happen
}
```

Tests can pass 1000 times and still have races.

Detecting Violations: The Race Detector

```
go run -race main.go
go test -race
go build -race
```

What it detects:

- Unsynchronized access to shared memory
- At least one access is a write
- No happens-before relationship

What it doesn't detect:

- Races that don't execute during the test
- Deadlocks, livelocks, resource leaks

Always run tests with `-race` in CI/CD.

The Safe Patterns

Pattern 1: Message Passing (Channel)

```
type Message struct {
    Data []int
}

ch := make(chan Message)

// Sender owns Data before send
msg := Message{Data: []int{1,2,3}}
ch <- msg  // Happens-before receive
```

```
// Don't use msg.Data after send (ownership transferred)

// Receiver
received := <-ch
// Can safely use received.Data (ownership gained)
```

Pattern 2: Mutex Protection

```
var (
    mu   sync.Mutex
    data map[string]int
)

// All access protected
mu.Lock()
data["key"] = value
mu.Unlock()

mu.Lock()
v := data["key"]
mu.Unlock()
```

Pattern 3: Atomic Operation

```
var counter int64

// All access atomic
atomic.AddInt64(&counter, 1)
atomic.LoadInt64(&counter)
```

Pattern 4: Write Once, Read Many

```
var config *Config

func init() {
    config = loadConfig()  // Write once in init
}

// Many goroutines can read without synchronization
func handler() {
    use(config)  // Safe: init →hb main →hb handler
}
```

Interview Traps

Trap 1: "This code is fine—it works in tests"

Wrong. Absence of observed race ≠ absence of race.

Correct answer:
"Tests can pass even with data races due to timing, compiler optimizations, or CPU reordering. The Go memory model specifies that without synchronization (happens-before relationship), reads can see stale or inconsistent values. I must use proper synchronization primitives."

Trap 2: "Adding time.Sleep fixes the race"

Wrong. time.Sleep is not syncchronization.

Correct answer:
"Time-based coordination doesn't provide happens-before guarantees. Even with sleep, compiler/CPU reordering can cause reads to see stale values. Proper synchronization (channels, mutexes, atomics) is required to establish memory visibility."

Trap 3: "Reading a bool doesn't need synchronization"

Wrong. All unsynchronized access to shared data is a race.

Correct answer:
"Even single-byte reads/writes can race. While the value might not be corrupted, without synchronization there's no happens-before guarantee, so reads might see stale values indefinitely due to CPU caching. Use atomic.LoadUint32/StoreUint32 for flags."

Trap 4: "The race detector proves my code is correct"

Wrong. Race detector finds races in executed code paths only.

Correct answer:
"The race detector is a dynamic analysis tool—it only detects races that occur during execution. Code paths not executed, rare interleavings, or races that require high contention might not be caught. I must reason about happens-before relationships to ensure correctness."

Key Takeaways

1. **Happens-before defines memory visibility** (not just time ordering)
2. **Within goroutine: program order guaranteed**
3. **Across goroutines: no guarantee without synchronization**
4. **Synchronization operations create happens-before**:
    - Channel send/receive
    - Mutex lock/unlock
    - WaitGroup done/wait
    - Atomic operations
    - sync.Once
5. **Time, sleep, gosched do NOT synchronize**
6. **Race detector is necessary but not sufficient**
7. **Always reason about happens-before, not time**

What You Should Be Thinking Now

- "What exactly is the happens-before relationship?"
- "How do I prove my code has proper synchronization?"
- "What's the difference between visibility and ordering?"
- "Why can compilers and CPUs reorder operations?"

Next: happens-before.md - Deep dive into the happens-before partial order.

Exercises (Do These Before Moving On)

1. Write a program with a data race on a boolean flag. Run with `-race`. Fix it with atomic.
2. Create two goroutines: write `x=1; y=1` in one, read `y; x` in the other. Prove the reader can see `y==1` but `x==0` (run billions of times or use CPU-specific barriers).
3. Explain why `time.Sleep` doesn't synchronize using happens-before.
4. Find 3 synchronization points in your own code that create happens-before relationships.

Don't continue until you can explain: "Why can one goroutine see partial effects of another goroutine's operations without synchronization?"