Common Misconceptions About Concurrency and Memory

The "Works On My Machine" Fallacy

Misconception: "My code runs fine in tests, so it's correct."

Why it's wrong:

1. **Timing-dependent:** Race conditions are non-deterministic
2. **CPU-dependent:** x86 has stronger memory ordering than ARM
3. **Load-dependent:** Low contention hides races
4. **Optimization-dependent:** Debug builds can hide races present in optimized builds

```
var ready bool
var data int

func writer() {
    data = 42
    ready = true
}

func reader() {
    for !ready {}  // Spin
    print(data)
}

// Runs fine 99.99% of time
// But CAN print 0 (data race)
```

In tests: Usually prints 42 (lucky timing).
In production: Occasionally prints 0 (different CPU, load, optimization).

Real example: A trading system ran for 6 months without issues, then a single race condition caused $2.3M loss when market volatility caused different timing.

Truth: Tests cannot prove absence of races. Must reason about happens-before.

The "Atomic Types" Fallacy

Misconception: "Reads and writes to small types (bool, int) are atomic, so no synchronization needed."

Why it's wrong:

Even if individual reads/writes are atomic (no tearing), **without synchronization there's no visibility guarantee.**

```
var flag bool
```

```go
// Goroutine 1
flag = true

// Goroutine 2
for !flag {}  // Can spin forever!
print("done")
```

What happens:

1. G1 writes `flag=true` to its CPU cache
2. G2 reads `flag` from its own CPU cache (cached `false`)
3. **G2's cache never invalidated** → spins forever

This is NOT a bug in Go. This is how modern CPUs work (cache coherency is not instantaneous).

Fix: Use `atomic.LoadBool` / `atomic.StoreBool` .

```go
var flag atomic.Bool

// Goroutine 1
flag.Store(true)  // Inserts memory barrier

// Goroutine 2
for !flag.Load() {}  // Inserts memory barrier
print("done")
```

Truth: Small types != atomic operations. Atomicity ≠ visibility. Use sync/atomic.

Real example: Video streaming service had goroutines spin indefinitely waiting for boolean flag, causing memory leak. Worked in dev (fewer CPUs), failed in production.

The "Single Writer" Fallacy

Misconception: "If only one goroutine writes, I don't need synchronization."

Why it's wrong:

Even with single writer, **readers need synchronization to see writes.**

```go
var config *Config

// Only this goroutine writes (single writer)
func reloadConfig() {
    newConfig := loadFromFile()  // (1)
    config = newConfig           // (2) Write
}

// Many goroutines read
func handleRequest() {
    cfg := config  // (3) Read
    if cfg != nil {
        use(cfg)   // Might see partially constructed Config
    }
}
```

Problem: No happens-before between (2) and (3).

- Reader can see pointer but not fields (partial construction)
- Reader might see old pointer indefinitely (caching)

Fix: Use atomic.Pointer or mutex.

```go
var config atomic.Pointer[Config]

func reloadConfig() {
    newConfig := loadFromFile()
    config.Store(newConfig)  // Atomic store
}

func handleRequest() {
    cfg := config.Load()  // Atomic load
    if cfg != nil {
        use(cfg)
    }
}
```

Truth: Multiple readers need synchronization with writer even if writer is single.

The "Immutable Data" Fallacy

Misconception: "Immutable data doesn't need synchronization."

Why it's wrong:

Even if data is immutable after construction, **publication of the data needs synchronization.**

```go
type Config struct {
    Host string
    Port int
}

var config *Config

func init() {
    config = &Config{  // (1) Construct
        Host: "localhost",
        Port: 8080,
    }
}

func main() {
    go func() {
        if config != nil {
            connect(config.Host, config.Port)  // (2) Use
        }
    }()
}
```

Problem: No happens-before between init and goroutine.

Wait, doesn't init happen-before main?

Yes: init →hb main starts.
But: main starts →hb go func()... does NOT mean init →hb inside func().

Correct reasoning:

- init →hb main starts (guaranteed)
- main starts →hb "go func()" statement (guaranteed)
- "go func()" statement →hb func executes (guaranteed)
- **Chain: init →hb (2) ✓ Actually safe!**

But this is NOT safe:

```go
var config *Config

func Load() {
    config = &Config{...}  // (1)
}

func Use() {
    if config != nil {
        use(config)  // (2)
    }
}

// In main:
go Load()
go Use()

// NO happens-before between (1) and (2) → race
```

Truth: Immutable data still requires synchronized publication.

The "Happens-Before Means Time" Fallacy

Misconception: "Happens-before means happens earlier in wall-clock time."

Why it's wrong:

Happens-before is about **causality and visibility**, not time.

```go
var a, b int
ch := make(chan int, 1)

// Goroutine 1
```

```
ch <- 0       // (1) Happens at time T
a = 1         // (2) Happens at time T+10

// Goroutine 2
<-ch          // (3) Happens at time T+1
print(a)      // (4) Happens at time T+2
```

In wall-clock time: (3) and (4) happen BEFORE (2).

But happens-before: (1) →hb (3), NOT (2) →hb (4).

Result: (4) can print 0 even though it executed after (2) in real time.

Truth: Happens-before is about synchronization, not time. Operations can happen later in time but not be visible.

The "Volatile Is Enough" Fallacy (From Other Languages)

Misconception: "Go doesn't have `volatile`, but I can simulate it."

Why it's wrong:

In C/C++:

```
volatile int flag = 0;
```

- Prevents compiler from optimizing away reads/writes
- Does NOT insert memory barriers (usually)
- Does NOT guarantee visibility across CPUs

In Java:

```
volatile int flag = 0;
```

- Inserts memory barriers (acquire/release semantics)
- Guarantees visibility

In Go:

- No `volatile` keyword
- Must use sync/atomic for visibility

Attempting to simulate volatile:

```
//go:noinline
func readFlag() bool { return flag }

// WRONG: This prevents inlining, not caching
```

Truth: Go requires sync/atomic for visibility. No way to simulate volatile.

The "Mutexes Protect Data" Fallacy

Misconception: "Mutexes protect data structures."

Why it's misleading:

Mutexes don't know about data. **You protect data by establishing conventions.**

```go
var mu sync.Mutex
var count int

func increment() {
    mu.Lock()
    count++
    mu.Unlock()
}

func read() int {
    // WRONG: Forgot to lock
    return count  // Data race!
}
```

Mutex didn't protect count. You must lock in ALL accesses.

Correct version:

```go
func read() int {
    mu.Lock()
    defer mu.Unlock()
    return count  // Safe
}
```

Truth: Mutexes protect critical sections, not data. You must ensure all accesses are inside critical sections.

Real example: Payment processor had mutex protecting balance updates but forgot to protect balance reads in logging. Rare race caused double-spend when reader saw partial update.

The "Race-Free Equals Correct" Fallacy

Misconception: "If my code has no data races, it's correct."

Why it's wrong:

Race-free doesn't mean **correct logic.**

```go
var (
    balance int
    mu      sync.Mutex
)

func withdraw(amount int) bool {
    mu.Lock()
    defer mu.Unlock()

    if balance >= amount {
        // Simulated processing time
        time.Sleep(time.Millisecond)
        balance -= amount
        return true
    }
    return false
}

// Goroutine 1: withdraw(100)
// Goroutine 2: withdraw(100)
// Balance: 150

// Both withdrawals succeed! Balance = -50
```

No data race (all accesses protected by mutex).
But incorrect (allows overdraft).

Fix: Check-then-act must be atomic at business-logic level, not just memory level.

Truth: Race-freedom is necessary but not sufficient for correctness.

The "Buffered Channels Don't Block" Fallacy

Misconception: "Buffered channels never block, unbuffered channels always block."

Why it's wrong:

Buffered channels block when full:

```go
ch := make(chan int, 2)  // Buffer size 2

ch <- 1  // Doesn't block (buffer: 1/2)
ch <- 2  // Doesn't block (buffer: 2/2)
ch <- 3  // BLOCKS forever (buffer full, no receiver)
```

Unbuffered channels don't always block:

```go
ch := make(chan int)

// Goroutine 1
ch <- 0  // Blocks until receiver ready

// Goroutine 2
<-ch  // Receiver ready, sender unblocks immediately
```

Truth: All channel sends can block if no receiver or buffer full. **All channel receives block** if no sender or buffer empty.

The "Goroutines Are Free" Fallacy

Misconception: "Goroutines are so lightweight, I can create millions."

Why it's wrong:

Goroutines are cheap, not free.

Cost per goroutine:

- 2 KB initial stack (can grow)
- Goroutine struct (~few hundred bytes)
- Scheduler overhead

1 million goroutines:

- Stack: 2 GB minimum
- Runtime structures: ~500 MB
- Scheduler contention

Real example: Web scraper created 1 million goroutines, consumed 3 GB, then crashed with OOM when stacks grew.

Fix: Use worker pool pattern (bounded concurrency).

```go
const maxWorkers = 100

sem := make(chan struct{}, maxWorkers)

for url := range urls {
    sem <- struct{}{}  // Acquire
    go func(u string) {
        defer func() { <-sem }()  // Release
        fetch(u)
    }(url)
}
```

Truth: Goroutines are cheap, but not free. Bound concurrency in production.

The "nil Channel Is Broken" Fallacy

Misconception: "Sending/receiving on nil channel is a bug."

Why it's misleading:

Nil channel behavior is **intentional and useful.**

```go
var ch chan int  // nil

ch <- 0  // Blocks forever
<-ch     // Blocks forever

// In select:
```

```
select {
case ch <- 0:  // Never selected (blocks forever)
case <-ch:     // Never selected
case <-other:  // This can be selected
}
```

Useful pattern: Disabling select cases

```
func merge(a, b <-chan int) <-chan int {
    out := make(chan int)
    go func() {
        defer close(out)
        for a != nil || b != nil {
            select {
            case v, ok := <-a:
                if !ok {
                    a = nil  // Disable this case
                    continue
                }
                out <- v
            case v, ok := <-b:
                if !ok {
                    b = nil  // Disable this case
                    continue
                }
                out <- v
            }
        }
    }()
    return out
}
```

Truth: Nil channels are a feature, not a bug. Use them to disable select cases.

The "Close Detects All Goroutines Done" Fallacy

Misconception: "close(ch) means all senders are done."

Why it's wrong:

You can close a channel while senders are still active → **panic**.

```
ch := make(chan int)

// Goroutine 1
go func() {
    time.Sleep(time.Millisecond)
    ch <- 1  // Might panic if ch closed
}()
```

```go
// Goroutine 2
close(ch)  // Close immediately
```

Panic:

```
panic: send on closed channel
```

Correct pattern:

1. **Only sender closes** (or use sync.WaitGroup to coordinate)
2. Never close from receiver

```go
ch := make(chan int)
var wg sync.WaitGroup

wg.Add(1)
go func() {
    defer wg.Done()
    ch <- 1
}()

go func() {
    wg.Wait()    // Wait for all senders
    close(ch)    // Safe to close
}()
```

Truth: close() doesn't detect completion; you must ensure completion first.

The "Context Cancel Stops Goroutines" Fallacy

Misconception: "ctx.Cancel() stops goroutines."

Why it's wrong:

Context cancellation is **cooperative**. Goroutines must check context.

```go
ctx, cancel := context.WithCancel(context.Background())

go func() {
    for {
        work()  // Doesn't check context → never stops
    }
}()

cancel()  // Goroutine keeps running!
```

Fix: Check context in loop.

```go
go func() {
    for {
        select {
        case <-ctx.Done():
            return  // Goroutine exits
        default:
            work()
        }
    }
}()
```

Truth: Context cancellation is a signal, not a kill command. Goroutines must cooperate.

The "Race Detector Finds All Races" Fallacy

Misconception: "If race detector doesn't report races, my code is race-free."

Why it's wrong:

Race detector is **dynamic analysis** → only finds races in executed code paths.

```
func process(condition bool) {
    if condition {
        // Path A (uses shared variable)
    } else {
        // Path B (uses shared variable)
    }
}

// Test only exercises Path A
// Race in Path B not detected
```

Coverage-dependent: Need to execute all code paths under all interleavings (impossible).

Truth: Race detector is necessary but not sufficient. Must also reason about happens-before.

The "Defer Runs Before Return" Fallacy

Misconception: "defer runs before function returns."

Truth: Defer runs before function returns **but after return value is evaluated**.

```
func mystery() (result int) {
    defer func() {
        result++  // Modifies return value
    }()
    return 0  // Returns 0, then defer runs, result becomes 1
}

fmt.Println(mystery())  // Prints 1
```

But:

```go
func mystery2() int {
    result := 0
    defer func() {
        result++  // Modifies local variable, not return value
    }()
    return result  // Returns 0 (value copied before defer)
}

fmt.Println(mystery2())  // Prints 0
```

In concurrency:

```go
func send(ch chan int, v int) {
    defer close(ch)  // Wrong order!
    ch <- v
}

// Receiver might miss value if close() happens first
```

Truth: defer executes in LIFO order after return value evaluation. Be careful with ordering.

Interview Traps

Trap 1: "Code review looks fine, no obvious races"

Wrong assumption: Visual inspection finds races.

Correct answer: "Visual inspection can spot obvious races, but subtle races require reasoning about happens-before relationships. All shared memory accesses need synchronization or happens-before ordering. I would also run race detector and write concurrency tests."

Trap 2: "This worked in 10,000 test runs"

Wrong assumption: Passing tests prove correctness.

Correct answer: "Passing tests increase confidence but don't prove absence of races. Race conditions are non-deterministic and depend on timing, CPU architecture, and load. Must verify happens-before relationships and use race detector."

Trap 3: "I'll just add time.Sleep to fix the timing issue"

Wrong: time.Sleep is not synchronization.

Correct answer: "time.Sleep doesn't provide happens-before guarantees and just masks races. The correct fix is to use proper synchronization: channels for communication, mutexes for shared state, or atomic operations for flags."

Trap 4: "Reads don't need protection, only writes"

Wrong: Readers need synchronization to see writes.

Correct answer: "Both reads and writes need synchronization to establish happens-before. Without it, reads might see stale values indefinitely due to CPU caching, even if writes are completed."

Key Takeaways

1. **Tests cannot prove absence of races** (non-deterministic)
2. **"Atomic" types ≠ sync/atomic operations** (no visibility without barriers)
3. **Single writer still needs reader synchronization** (visibility is bidirectional)
4. **Publication of immutable data needs synchronization** (construction vs. publication)
5. **Happens-before ≠ time ordering** (causality, not clocks)
6. **Mutexes protect critical sections, not data** (you must enforce convention)
7. **Race-free ≠ correct** (no races doesn't mean correct logic)
8. **All channels can block** (buffered when full, unbuffered when no receiver)
9. **Goroutines cost memory** (stack + runtime structures)
10. **Race detector is necessary, not sufficient** (dynamic analysis, not proof)

What You Should Be Thinking Now

- "What mistakes am I making in my current code?"
- "How do I verify my happens-before reasoning?"
- "What patterns should I always use?"

Next: Section 03: Classic Problems - Learn how race conditions manifest in real code.

Exercises

1. Find 5 misconceptions you held before reading this. Write down what you learned.

2. Review your recent code: Find 3 places where you made assumptions about:

   - Visibility without synchronization
   - Mutexes protecting data
   - Tests proving correctness

3. Write a program that "works" in tests but has a race. Run with `-race` to confirm.

4. Explain to a colleague why "it works on my machine" is not proof of correctness.

5. Create a checklist for code review: What questions should you ask about every shared variable?

Don't continue until you can: "Identify and debunk dangerous misconceptions about concurrent code."