Trick Questions and Gotchas

Closure Traps

Trap 1: Loop Variable Capture

```
// TRAP
var wg sync.WaitGroup
for i := 0; i < 5; i++ {
    wg.Add(1)
    go func() {
        defer wg.Done()
        fmt.Println(i)  // What prints?
    }()
}
wg.Wait()
```

Expected: 0, 1, 2, 3, 4
Actual: 5, 5, 5, 5, 5 (or mix if timing varies)

Why: Goroutines capture `i` by reference. Loop finishes before goroutines start, all see final value.

Fix:

```
go func(i int) {  // Pass by value
    defer wg.Done()
    fmt.Println(i)
}(i)
```

Trap 2: Range Variable Reuse

```
// TRAP
items := []Item{{ID: 1}, {ID: 2}, {ID: 3}}

for _, item := range items {
    go func() {
        process(item)  // All goroutines see last item!
    }()
}
```

Why: `item` is reused each iteration, all goroutines see final value.

Fix:

```
for _, item := range items {
    item := item  // Shadow to create new variable
    go func() {
        process(item)
    }()
}
```

Channel Gotchas

Trap 3: Closing Channel Multiple Times

```go
// TRAP
ch := make(chan int)
close(ch)
close(ch)  // Panic: close of closed channel
```

Prevention:

```go
var once sync.Once
once.Do(func() { close(ch) })
```

Trap 4: Sending to Closed Channel

```go
// TRAP
ch := make(chan int)
close(ch)
ch <- 42  // Panic: send on closed channel
```

Pattern: Only sender should close.

Trap 5: Buffered Channel Doesn't Block

```go
// TRAP: Thinking this waits
ch := make(chan int, 10)
ch <- 42
// Continues immediately! Buffer not full.
```

Clarification: Buffered channel blocks only when full.

Trap 6: Nil Channel Blocks Forever

```go
// TRAP
var ch chan int  // nil
ch <- 42  // Blocks forever, not panic!
```

Use case: Disable select case:

```go
var stopCh chan struct{}
if !shouldStop {
    stopCh = nil  // Disables this case in select
}

select {
case <-stopCh:  // Never chosen if nil
    return
case work := <-workCh:
    process(work)
}
```

Trap 7: Range on Closed Channel

```go
ch := make(chan int, 3)
ch <- 1
ch <- 2
ch <- 3
close(ch)

// Does this exit?
for val := range ch {
    fmt.Println(val)  // Prints 1, 2, 3, then exits
}
```

Clarification: `range` exits when channel closed and drained. Not a trap, but often misunderstood as hanging.

WaitGroup Pitfalls

Trap 8: Add Inside Goroutine

```go
// TRAP
var wg sync.WaitGroup

for i := 0; i < 10; i++ {
    go func() {
        wg.Add(1)  // Race! Main may call Wait() before this executes
        defer wg.Done()
        process(i)
    }()
}

wg.Wait()  // May return too early
```

Fix: `Add()` before spawning:

```go
for i := 0; i < 10; i++ {
    wg.Add(1)  // Before go
    go func() {
        defer wg.Done()
        process(i)
    }()
}
```

Trap 9: Negative Counter

```
// TRAP
var wg sync.WaitGroup
wg.Done()  // Panic: sync: negative WaitGroup counter
```

Why: `Done()` without matching `Add()` .

Trap 10: Reusing WaitGroup

```
// TRAP
var wg sync.WaitGroup

// First batch
wg.Add(5)
for i := 0; i < 5; i++ {
    go func() { defer wg.Done(); work() }()
}
wg.Wait()

// Second batch - safe or trap?
wg.Add(5)  // Actually OK! Can reuse after Wait() returns
for i := 0; i < 5; i++ {
    go func() { defer wg.Done(); work() }()
}
wg.Wait()
```

Clarification: WaitGroup can be reused after `Wait()` returns. Counter must be zero.

Mutex Mistakes

Trap 11: Copying Mutex

```
// TRAP
type Counter struct {
    mu    sync.Mutex
    value int
}

func (c Counter) Inc() {  // Value receiver!
    c.mu.Lock()  // Locks copy, not original
    c.value++
    c.mu.Unlock()
}
```

Fix: Pointer receiver:

```go
func (c *Counter) Inc() {  // Pointer
    c.mu.Lock()
    c.value++
    c.mu.Unlock()
}
```

Trap 12: Recursive Lock

```go
// TRAP
var mu sync.Mutex

func outer() {
    mu.Lock()
    defer mu.Unlock()
    inner()  // Deadlock!
}

func inner() {
    mu.Lock()  // Already locked by outer
    defer mu.Unlock()
    // work
}
```

Why: sync.Mutex not reentrant (can't lock twice in same goroutine).

Fix: Refactor to not call `inner()` while holding lock.

Trap 13: Defer Unlock After Return

```go
// TRAP
func getValue() int {
    mu.Lock()
    defer mu.Unlock()

    if condition {
        return computeValue()  // Executes BEFORE defer!
        // No, defer executes before return
    }
    return defaultValue
}
```

Clarification: This is actually **safe**! `defer` executes before function returns, so unlock happens correctly. Not a trap, but often assumed to be one.

Select Statement Tricks

Trap 14: Select Doesn't Loop

```go
// TRAP: Thinking this processes all messages
select {
case msg := <-ch1:
    handle(msg)
case msg := <-ch2:
    handle(msg)
}
// Only handles ONE message, then exits
```

Fix: Wrap in loop:

```go
for {
    select {
    case msg := <-ch1:
        handle(msg)
    case msg := <-ch2:
        handle(msg)
    }
}
```

Trap 15: Default Always Chosen

```go
// TRAP
for {
    select {
    case work := <-workCh:
        process(work)
    default:
        // Infinite busy loop!
        fmt.Println("No work")
    }
}
```

Why: `default` chosen immediately if no other case ready. CPU spins at 100%.

Fix: Remove default or add sleep:

```go
for {
    select {
    case work := <-workCh:
        process(work)
    case <-time.After(100 * time.Millisecond):
        // Periodic check
    }
}
```

Trap 16: Select Non-Determinism

```
ch := make(chan int, 2)
ch <- 1
ch <- 2

// Which value is received?
select {
case val := <-ch:
    fmt.Println(val)  // 1? 2? Random!
case val := <-ch:
    fmt.Println(val)
}
```

Answer: Undefined! If multiple cases ready, Go chooses randomly. This is intentional (prevents starvation).

Context Confusion

Trap 17: Context in Struct

```
// TRAP (anti-pattern)
type Server struct {
    ctx context.Context  // Don't do this!
}

func (s *Server) Handle(req Request) {
    // Use s.ctx?
}
```

Why wrong: Context lifetime tied to request, not struct. Should pass as parameter.

Fix:

```
func (s *Server) Handle(ctx context.Context, req Request) {
    // Use ctx parameter
}
```

Trap 18: Not Calling cancel()

```
// TRAP: Memory leak
func process() {
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    // Forgot: defer cancel()

    doWork(ctx)
}
```

Why: Timer never stopped, leaks resources.

Fix:

```go
ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
defer cancel()  // Always defer!
```

Atomic Operations

Trap 19: Atomic Not Composable

```go
// TRAP: Thinking this is atomic
var counter int64

// Check and increment - NOT ATOMIC!
if atomic.LoadInt64(&counter) < 100 {
    atomic.AddInt64(&counter, 1)  // Race! Value may have changed
}
```

Fix: Use mutex for compound operations:

```go
var mu sync.Mutex

mu.Lock()
if counter < 100 {
    counter++
}
mu.Unlock()
```

Trap 20: Atomic Struct Field

```go
// TRAP
type Stats struct {
    requests int64
    errors   int64
}

var stats Stats

func recordRequest() {
    atomic.AddInt64(&stats.requests, 1)  // OK
}

func updateBoth() {
    // TRAP: Two atomic ops != one atomic op
    atomic.AddInt64(&stats.requests, 1)
    atomic.AddInt64(&stats.errors, 1)
    // Not atomic together! Can observe inconsistent state
}
```

Tricky Interview Questions

Q1: What does this print?

```
func main() {
    ch := make(chan int, 1)
    ch <- 1
    close(ch)

    fmt.Println(<-ch)
    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
```

Answer:

```
1
0
0
```

Closed channel returns zero value forever (not panic on receive).

Q2: Does this leak goroutines?

```
func search(query string) Result {
    ch := make(chan Result)

    go func() {
        result := expensiveSearch(query)
        ch <- result  // Blocks if no one receives
    }()

    select {
    case result := <-ch:
        return result
    case <-time.After(100 * time.Millisecond):
        return Result{Error: "timeout"}  // Goroutine still blocked!
    }
}
```

Answer: YES! Goroutine leaks if timeout hit. Buffered channel fixes:

```
ch := make(chan Result, 1)  // Buffer size 1
```

Q3: Race condition or not?

```
var x int

func main() {
    go func() { x = 1 }()
    go func() { x = 2 }()
    time.Sleep(time.Second)
    fmt.Println(x)
}
```

Answer: YES, race! Two writes to same variable without synchronization. Use `-race` to detect.

Q4: What's the bug?

```go
type Cache struct {
    mu   sync.RWMutex
    data map[string]string
}

func (c *Cache) Get(key string) string {
    c.mu.RLock()
    val := c.data[key]
    c.mu.RUnlock()

    if val == "" {
        c.mu.Lock()
        c.data[key] = "default"  // Bug!
        val = "default"
        c.mu.Unlock()
    }

    return val
}
```

Answer: Race between check (`val == ""`) and set. Two goroutines can both see empty, both set.

Fix: Check again after acquiring write lock:

```go
c.mu.RLock()
val := c.data[key]
c.mu.RUnlock()

if val == "" {
    c.mu.Lock()
    // Check again!
    if c.data[key] == "" {
        c.data[key] = "default"
        val = "default"
    } else {
        val = c.data[key]
    }
    c.mu.Unlock()
}
```

Q5: How many goroutines after this?

```go
func main() {
    for i := 0; i < 10; i++ {
        go func() {
            time.Sleep(time.Hour)
        }()
    }
}
```

Answer: Zero! `main()` exits immediately, all goroutines killed. Program doesn't wait.

Fix:

```go
var wg sync.WaitGroup
for i := 0; i < 10; i++ {
    wg.Add(1)
    go func() {
        defer wg.Done()
        time.Sleep(time.Hour)
    }()
}
wg.Wait()
```

Anti-Patterns Checklist

Closures:

- [ ] Capturing loop variable by reference
- [ ] Range variable reuse in goroutine

Channels:

- [ ] Closing closed channel
- [ ] Sending to closed channel
- [ ] Not closing producer channel
- [ ] Send/receive on nil channel (unless intentional)

WaitGroup:

- [ ] Add() inside goroutine (race)
- [ ] Done() without Add()
- [ ] Not waiting for WaitGroup

Mutex:

- [ ] Copying mutex (use pointer receiver)
- [ ] Recursive locking
- [ ] Not unlocking (use defer)

Select:

- [ ] Forgetting loop (processes only one)

- [ ] Default causing busy loop
- [ ] Assuming deterministic order

Context:

- [ ] Storing in struct
- [ ] Not calling cancel() (leak)
- [ ] Passing nil context

Atomic:

- [ ] Assuming compound operations atomic
- [ ] Mixing atomic with non-atomic access

Interview Defense Strategies

1. **Always mention race detector:** "I'd run `-race` to confirm."

2. **Question assumptions:** "Is this buffered or unbuffered?"

3. **Consider timing:** "What if the loop finishes before goroutines start?"

4. **Check lifecycle:** "Who closes this channel? When?"

5. **Think about edge cases:** "What if context is cancelled?"

6. **Mention testing:** "I'd write a test with high concurrency to expose this."

7. **Ask clarifying questions:** "Should this block or fail fast?"

8. **Walk through execution:** "Goroutine 1 locks mu1, then Goroutine 2 locks mu2..."

Key Takeaways

1. **Closures capture by reference** (shadow variables)
2. **Only sender closes channels**
3. **Add() before spawning goroutine**
4. **Always defer cancel() for context**
5. **Atomic operations not composable**
6. **Select chooses randomly if multiple ready**
7. **Nil channel blocks forever** (useful for disabling)
8. **Buffered channel prevents send goroutine leak**
9. **RLock → unlock → Lock has race** (check twice)
10. **main() exit kills all goroutines**

Practice Problems

1. Identify race in given code without running.

2. Fix goroutine leak in timeout pattern.

3. Explain why closure captures wrong value.

4. Debug deadlock with stack traces.

5. Spot mutex copy bug.

Next: whiteboard-designs.md - Designing concurrent systems on whiteboard.