sync.Mutex

Definition (Precise)

A **mutex** (mutual exclusion lock) is a synchronization primitive that protects access to shared resources by ensuring only one goroutine can hold the lock at a time.

Purpose: Protect **critical sections**—code that accesses shared state.

Syntax

```go
import "sync"

var mu sync.Mutex

mu.Lock()       // Acquire lock (blocks if already held)
// Critical section
mu.Unlock()     // Release lock

// Common pattern with defer
mu.Lock()
defer mu.Unlock()
// Critical section
```

Mental Model

Think of a mutex as a **single key to a room**:

- Only one person (goroutine) can hold the key at a time
- Others wait in line until the key is returned
- The room is your **critical section** (shared state)

```
Goroutine A: [Waiting]──→[Lock]──→[Critical Section]──→[Unlock]
Goroutine B: [Waiting]────────────────[Waiting]─────→[Lock]──→...
Goroutine C: [Waiting]────────────────[Waiting]──────────────→...
```

When to Use Mutex

✅ **Use mutex for:**

- Protecting **shared state** (counters, maps, slices)
- **Short critical sections** (microseconds)
- **Caching** (read-heavy shared data)
- Coordinating **access, not transfer** of data

```go
// Good: Protecting shared map
type Cache struct {
    mu    sync.Mutex
    items map[string]string
}

func (c *Cache) Get(key string) (string, bool) {
    c.mu.Lock()
    defer c.mu.Unlock()
    val, ok := c.items[key]
    return val, ok
}

func (c *Cache) Set(key, val string) {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.items[key] = val
}
```

❌ **Don't use mutex for:**

- **Transferring ownership** of data (use channels)
- **Long operations** (holding lock during I/O, network, etc.)
- **Goroutine coordination** at program level (use channels/context)

Correct Usage Patterns

Pattern 1: Defer Unlock (Safest)

```go
func (c *Counter) Increment() {
    c.mu.Lock()
    defer c.mu.Unlock()  // Guaranteed to unlock even if panic
    c.count++
}
```

Why defer?

- Ensures unlock even if function panics
- Clear pairing of Lock/Unlock
- Less error-prone

Pattern 2: Struct with Embedded Mutex

```go
type SafeCounter struct {
    mu    sync.Mutex  // Unexported (private)
    count int
}

func (c *SafeCounter) Inc() {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.count++
}

func (c *SafeCounter) Value() int {
    c.mu.Lock()
    defer c.mu.Unlock()
    return c.count
}
```

Key points:

- Mutex is **unexported** (lowercase `mu`)
- Only methods lock/unlock (enforces encapsulation)
- Never expose the mutex or the protected field directly

Pattern 3: Returning Copies, Not References

```
// WRONG: Returns reference to protected state
func (c *Config) GetSettings() *Settings {
    c.mu.Lock()
    defer c.mu.Unlock()
    return &c.settings  // Leaks reference!
}
// Caller can modify c.settings without lock → race

// CORRECT: Return copy
func (c *Config) GetSettings() Settings {
    c.mu.Lock()
    defer c.mu.Unlock()
    return c.settings  // Copy (if value) or deep copy (if complex)
}
```

Common Bugs

Bug 1: Forgetting to Unlock

```
// WRONG
func (c *Counter) Inc() {
    c.mu.Lock()
    c.count++
    // Forgot mu.Unlock() → Lock held forever → Deadlock
}
```

Fix: Always use `defer c.mu.Unlock()` immediately after `Lock()` .

Bug 2: Copying Mutex

```
// WRONG
type Counter struct {
    mu    sync.Mutex
    count int
}
```

```go
func NewCounter() Counter {
    return Counter{}  // Returns copy → mutex copied
}

c1 := NewCounter()
c2 := c1  // Copies mutex → c1 and c2 have different locks!
```

Impact: `c1` and `c2` don't share a lock → no mutual exclusion.

Fix: Use pointers.

```go
func NewCounter() *Counter {
    return &Counter{}
}
```

Detection:

```
$ go vet
./main.go:10:9: assignment copies lock value to c2: main.Counter contains sync.Mutex
```

Bug 3: Nested Locking (Deadlock)

```go
// WRONG: Same goroutine tries to acquire same lock twice
func (c *Counter) Add(n int) {
    c.mu.Lock()
    defer c.mu.Unlock()

    for i := 0; i < n; i++ {
        c.Inc()  // Inc() tries to Lock() again → DEADLOCK
    }
}

func (c *Counter) Inc() {
    c.mu.Lock()  // Blocks waiting for itself
    defer c.mu.Unlock()
    c.count++
}
```

Fix: Split into locked and unlocked methods.

```go
func (c *Counter) Add(n int) {
    c.mu.Lock()
    defer c.mu.Unlock()
    c.count += n  // Direct access, already holding lock
}

func (c *Counter) Inc() {
    c.Add(1)  // Delegates to Add
}
```

Bug 4: Lock Held During Slow Operation

```go
// WRONG: Lock held during network I/O
func (c *Cache) FetchAndCache(key string) {
    c.mu.Lock()
    defer c.mu.Unlock()

    data := fetchFromNetwork(key)  // 100ms+ → all other operations blocked!
    c.items[key] = data
}
```

Fix: Minimize critical section.

```go
// CORRECT: Fetch outside lock
func (c *Cache) FetchAndCache(key string) {
    data := fetchFromNetwork(key)  // Outside lock

    c.mu.Lock()
    c.items[key] = data
    c.mu.Unlock()
}
```

Bug 5: Exposing Mutex or Protected Field

```go
// WRONG
type Counter struct {
    Mu    sync.Mutex  // Exported!
    Count int         // Exported!
}

// Callers can bypass protection
c := Counter{}
c.Count++  // Race! No lock held
```

Fix: Always keep mutex and protected fields unexported.

Deadlock Scenarios

Scenario 1: Lock Ordering Violation

```go
// Goroutine 1
mu1.Lock()
mu2.Lock()
// ...
mu2.Unlock()
mu1.Unlock()

// Goroutine 2
```

```go
mu2.Lock()  // Different order!
mu1.Lock()
// ...
mu1.Unlock()
mu2.Unlock()

// DEADLOCK: G1 holds mu1, waits for mu2; G2 holds mu2, waits for mu1
```

Fix: Always acquire locks in same order.

```go
// Both goroutines
mu1.Lock()
mu2.Lock()
// ...
mu2.Unlock()
mu1.Unlock()
```

Scenario 2: Forgotten Unlock in Error Path

```go
// WRONG
func (c *Counter) Process() error {
    c.mu.Lock()

    if err := validate(); err != nil {
        return err  // Forgot to unlock!
    }

    c.count++
    c.mu.Unlock()
    return nil
}
```

Fix: Use defer.

```go
func (c *Counter) Process() error {
    c.mu.Lock()
    defer c.mu.Unlock()

    if err := validate(); err != nil {
        return err  // Unlocks via defer
    }

    c.count++
    return nil
}
```

Performance Characteristics

Operation	Uncontended	Contended (10 goroutines)
Lock/Unlock	~20-30 ns	~500-1000 ns
Critical section	Duration of protected code	+ waiting time

Contention: When multiple goroutines compete for the same lock.

Cost of contention:

- Goroutines block (wasted parallelism)
- Context switches (scheduler overhead)
- Cache invalidation (coherency traffic)

Minimizing Lock Contention

Technique 1: Reduce Critical Section Size

```
// WORSE: Large critical section
func (c *Cache) Compute(key string) {
    c.mu.Lock()
    defer c.mu.Unlock()

    value := expensiveComputation()  // Slow, lock held entire time
    c.items[key] = value
}

// BETTER: Small critical section
func (c *Cache) Compute(key string) {
    value := expensiveComputation()  // Outside lock

    c.mu.Lock()
    c.items[key] = value
    c.mu.Unlock()
}
```

Technique 2: Use RWMutex for Read-Heavy Workloads

(See rwmutex.md)

Technique 3: Shard Locks

```
// WORSE: Single lock for entire map
type Cache struct {
    mu    sync.Mutex
    items map[string]string  // 1M items
}

// BETTER: Multiple locks (sharding)
type ShardedCache struct {
    shards [16]struct {
        mu    sync.Mutex
        items map[string]string
    }
}

func (c *ShardedCache) getShard(key string) *struct {
    mu    sync.Mutex
    items map[string]string
} {
    hash := fnv1a(key)
    return &c.shards[hash%16]
}

func (c *ShardedCache) Get(key string) (string, bool) {
    shard := c.getShard(key)
    shard.mu.Lock()
    defer shard.mu.Unlock()
    val, ok := shard.items[key]
    return val, ok
}
```

Benefit: 16x less contention (goroutines only block on same shard).

Mutex vs Channels Decision Tree

```
Question: Are you TRANSFERRING data between goroutines?
├─ YES → Use channels
│   └─ Examples: work distribution, results collection, pipelining
│
└─ NO → Are you PROTECTING shared state?
    ├─ YES → Use mutex
    │   └─ Examples: counter, cache, shared map
    │
    └─ UNCLEAR → Start with channels (safer, more Go-idiomatic)
        └─ Profile and optimize to mutex if needed
```

Real-World Failure: Lock Held During I/O

Company: Health tech startup (2017)

What happened:
API latency spiked from 50ms to 30+ seconds during peak load.

Root cause:

```
type UserCache struct {
    mu    sync.Mutex
    users map[int64]*User
}

func (c *UserCache) GetOrFetch(id int64) (*User, error) {
    c.mu.Lock()
    defer c.mu.Unlock()

    if user, ok := c.users[id]; ok {
        return user, nil
    }

    // BUG: Lock held during DB query (100ms+)
    user, err := database.FetchUser(id)
    if err != nil {
        return nil, err
    }

    c.users[id] = user
    return user, nil
}
```

During peak:

- User cache miss rate: 30%
- Database query time: 100ms
- Lock held for 100ms per miss
- All other GetOrFetch calls blocked
- Cascading latency increase

Fix:

```go
func (c *UserCache) GetOrFetch(id int64) (*User, error) {
    // Check cache with lock
    c.mu.Lock()
    user, ok := c.users[id]
    c.mu.Unlock()

    if ok {
        return user, nil
    }

    // Fetch from DB without lock
    user, err := database.FetchUser(id)
    if err != nil {
        return nil, err
    }

    // Update cache with lock
    c.mu.Lock()
    c.users[id] = user
    c.mu.Unlock()

    return user, nil
}
```

Better fix (handles duplicate fetches):

```go
type UserCache struct {
    mu          sync.Mutex
    users       map[int64]*User
    fetching    map[int64]chan *User  // Track in-progress fetches
}

func (c *UserCache) GetOrFetch(id int64) (*User, error) {
    c.mu.Lock()

    if user, ok := c.users[id]; ok {
        c.mu.Unlock()
        return user, nil
    }

    // Check if already fetching
    if ch, ok := c.fetching[id]; ok {
        c.mu.Unlock()
        user := <-ch  // Wait for fetch to complete
        return user, nil
    }

    // Start fetch
    ch := make(chan *User, 1)
    c.fetching[id] = ch
    c.mu.Unlock()

    // Fetch without lock
    user, err := database.FetchUser(id)
    if err != nil {
        c.mu.Lock()
        delete(c.fetching, id)
        c.mu.Unlock()
        close(ch)
        return nil, err
    }

    // Store result
    c.mu.Lock()
    c.users[id] = user
    delete(c.fetching, id)
    c.mu.Unlock()

    ch <- user
    close(ch)

    return user, nil
}
```

Lessons:

1. Never hold locks during I/O operations

2. Critical sections should be **microseconds, not milliseconds**
3. Prevent duplicate work (fetch deduplication pattern)

Interview Traps

Trap 1: "Mutexes prevent data races"

Incomplete. Mutexes only prevent races **if used correctly**.

Correct answer:
"Mutexes provide the mechanism to prevent data races, but don't automatically prevent them. Code must consistently use the same mutex to protect all accesses to shared state. Missing just one Lock/Unlock creates a race."

Trap 2: "This code is safe—it has a mutex"

```go
type Counter struct {
    mu    sync.Mutex
    count int
}

c := Counter{}
c.mu.Lock()
c.count++
c.mu.Unlock()

// Later, different code:
c.count++  // NO LOCK → RACE
```

Wrong. Protection requires consistent use.

Correct answer:
"Having a mutex doesn't make code safe. Every access to `c.count` must acquire the lock. Missing even one creates a data race. Proper design encapsulates the mutex and protected field, exposing only safe methods."

Trap 3: "defer has no performance cost"

Misleading. Defer has small overhead (~50-100ns).

Correct answer:
"Defer has minimal but non-zero cost (~50-100ns). For typical critical sections (100ns+), this is negligible. For tight, hot loops where critical work is sub-50ns, explicit Unlock may be marginally faster, but defer is usually worth it for correctness (guarantees unlock even on panic)."

Trap 4: "Mutexes are fair (FIFO)"

Wrong. Go's mutexes use a **starvation-prevention** mode, not strict FIFO.

Correct answer:
"Go's mutexes are not FIFO. They use a hybrid approach: allow some barging (new goroutines can acquire lock without waiting), but switch to FIFO mode if a goroutine waits too long (>1ms) to prevent starvation. This balances throughput and fairness."

Key Takeaways

1. **Mutex = mutual exclusion** (one goroutine in critical section)
2. **Always use defer mu.Unlock()** (prevents leaks on panic)
3. **Never copy mutexes** (use pointers)
4. **Keep critical sections small** (microseconds, not milliseconds)
5. **Never hold lock during I/O** (network, disk, sleep)
6. **Acquire locks in consistent order** (prevents deadlock)
7. **Encapsulate mutex + protected state** (don't export)
8. **Use channels for transfer, mutexes for shared state**

What You Should Be Thinking Now

- "When should I use RWMutex instead of Mutex?"
- "How do I handle read-heavy workloads efficiently?"
- "What's the difference between a reader lock and a writer lock?"
- "How do I minimize lock contention in read-dominated scenarios?"

Next: rwmutex.md - Optimizing for read-heavy workloads with reader-writer locks.

Exercises (Do These Before Moving On)

1. Write a thread-safe counter using mutex. Increment it from 100 goroutines. Verify correctness with race detector.

2. Intentionally create a deadlock with two mutexes. Fix it by enforcing lock ordering.

3. Implement a cache that makes the mistake of holding a lock during I/O. Benchmark it. Fix it. Benchmark again. Observe improvement.

4. Copy a mutex (trigger `go vet`). Understand why this breaks.

5. Write code that forgets to unlock in an error path. Fix it with defer.

Don't continue until you can explain: "Why should critical sections be as small as possible?"