sync.RWMutex

Definition (Precise)

A **reader-writer mutex** (RWMutex) allows multiple concurrent readers OR one exclusive writer. It's an optimization for read-heavy workloads where contention on a regular mutex would waste parallelism.

Key insight: Reads don't conflict with each other—only reads vs writes and writes vs writes.

Syntax

```
import "sync"

var mu sync.RWMutex

// Reader lock (shared, multiple concurrent readers)
mu.RLock()
// Read operations
mu.RUnlock()

// Writer lock (exclusive, blocks all readers and writers)
mu.Lock()
// Write operations
mu.Unlock()
```

Mental Model

Think of it as **library access**:

- **Readers (RLock):** Multiple people can read books simultaneously
- **Writer (Lock):** When someone needs to reorganize shelves, everyone leaves

```
Time: ──────────────────────────────────────────>

R1:   [RLock]──────────[RUnlock]
R2:   [RLock]──────────[RUnlock]
R3:       [RLock]──────────[RUnlock]
          ↑ Multiple readers concurrent

W1:                         [Lock]──────[Unlock]
                            ↑ Writer waits for all readers
                            ↑ No new readers while writer waits

R4:                         [Blocked]──→[RLock]──
                            ↑ Reader waits for writer
```

When to Use RWMutex

✅ **Use RWMutex when:**

- **Read-heavy workload** (90%+ reads)
- Reads significantly outpace writes
- Critical section worth ~500ns+ (RWMutex overhead ~30ns more than mutex)

❌ **Don't use RWMutex when:**

- **Write-heavy** (>50% writes)
- Critical section extremely short (<100ns)
- Equal mix of reads/writes

Benchmark to confirm! RWMutex is not always faster.

Correct Usage Patterns

Pattern 1: Read-Heavy Cache

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

func (c *Cache) Get(key string) (string, bool) {
    c.mu.RLock()
    defer c.mu.RUnlock()
    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
}
```

Why RWMutex helps:

- Multiple Get() calls execute in parallel
- Set() blocks all reads, but rare (write-infrequent)

Pattern 2: Configuration (Read-Many, Write-Rarely)

```
type Config struct {
    mu       sync.RWMutex
    settings map[string]string
}

func (c *Config) Get(key string) string {
    c.mu.RLock()
    defer c.mu.RUnlock()
    return c.settings[key]
}
```

```
func (c *Config) Reload() error {
    newSettings := loadFromFile()

    c.mu.Lock()
    c.settings = newSettings
    c.mu.Unlock()

    return nil
}
```

Pattern: Read thousands of times/sec, write once/hour → perfect for RWMutex.

Pattern 3: Upgrade Read Lock to Write Lock (WRONG)

```
// WRONG: Cannot upgrade lock
func (c *Cache) GetOrCompute(key string) string {
    c.mu.RLock()
    if val, ok := c.items[key]; ok {
        c.mu.RUnlock()
        return val
    }

    // Cannot upgrade RLock to Lock!
    c.mu.Lock()     // DEADLOCK: RLock still held
    defer c.mu.Unlock()

    val := compute(key)
    c.items[key] = val
    return val
}
```

Fix 1: Unlock, then Lock

```
func (c *Cache) GetOrCompute(key string) string {
    c.mu.RLock()
    val, ok := c.items[key]
    c.mu.RUnlock()  // Release read lock

    if ok {
        return val
    }

    c.mu.Lock()     // Now acquire write lock
    defer c.mu.Unlock()

    // Double-check (race: another goroutine may have written)
    if val, ok := c.items[key]; ok {
        return val
    }
```

```
    val = compute(key)
    c.items[key] = val
    return val
}
```

Fix 2: Pessimistic Lock (if writes common)

```
func (c *Cache) GetOrCompute(key string) string {
    c.mu.Lock()  // Just use write lock from start
    defer c.mu.Unlock()

    if val, ok := c.items[key]; ok {
        return val
    }

    val := compute(key)
    c.items[key] = val
    return val
}
```

Common Bugs

Bug 1: RLock/Lock Mismatch

```
// WRONG
c.mu.RLock()
defer c.mu.Unlock()  // Unlocking write lock, but hold read lock!
```

Impact: Lock leak, deadlock.

Fix: Match lock/unlock pairs.

```
c.mu.RLock()
defer c.mu.RUnlock()  // Correct
```

Bug 2: Assuming RLock Excludes Other RLock

```
// WRONG: Assuming exclusive read access
func (c *Counter) Get() int {
    c.mu.RLock()
    defer c.mu.RUnlock()
    // Does NOT prevent other concurrent reads
    return c.count
}
```

Not a bug per se, but misunderstanding: RLock allows **concurrent reads**. If you need exclusive access (even for reads), use `Lock()`.

Bug 3: Long Reads Starve Writers

```
// BAD: Read lock held during slow operation
func (c *Cache) Process(key string) {
    c.mu.RLock()
    defer c.mu.RUnlock()

    val := c.items[key]
    doSlowProcessing(val)  // 1 second; read lock held entire time
}

// Writer:
c.mu.Lock()  // Blocked for up to 1 second per concurrent reader
defer c.mu.Unlock()
c.items[key] = newVal
```

Fix: Release lock before slow operation.

```
func (c *Cache) Process(key string) {
    c.mu.RLock()
    val := c.items[key]
    c.mu.RUnlock()  // Release immediately

    doSlowProcessing(val)
}
```

Bug 4: Using RWMutex for Write-Heavy Workload

```
// BAD: 50% reads, 50% writes → RWMutex slower than mutex
func benchmark() {
    var mu sync.RWMutex

    // 50% reads
    mu.RLock()
    _ = data
    mu.RUnlock()

    // 50% writes
    mu.Lock()
    data = newVal
    mu.Unlock()
}
// RWMutex adds overhead without benefit
```

Guideline: Use RWMutex when reads are 90%+ of operations.

Performance Characteristics

Operation	Mutex (Uncontended)	RWMutex RLock (No Writers)	RWMutex Lock
Lock/Unlock	~20-30 ns	~30-50 ns	~25-35 ns
Contention (10 readers)	~500-1000 ns	~50-100 ns (parallel!)	~500-1000 ns

Key insight: RWMutex benefit comes from **parallel reads**, not faster operations.

Benchmark Example

```
// Setup
type Store struct {
    mu    sync.RWMutex  // or sync.Mutex
    items map[int]int
}

// 95% reads, 5% writes
func (s *Store) BenchmarkReads(b *testing.B) {
    for i := 0; i < b.N; i++ {
        if i%20 == 0 {
            // Write (5%)
            s.mu.Lock()
            s.items[i] = i
            s.mu.Unlock()
        } else {
            // Read (95%)
            s.mu.RLock()
            _ = s.items[i]
            s.mu.RUnlock()
        }
    }
}
```

Results (8 cores, heavy contention):

- `sync.Mutex` : 150 ns/op
- `sync.RWMutex` : 80 ns/op (1.9x faster)

Results (95% writes):

- `sync.Mutex` : 150 ns/op
- `sync.RWMutex` : 180 ns/op (1.2x **slower**)

Lesson: Benchmark your workload!

Writer Starvation Protection

Pre-Go 1.19: Writers could starve if continuous stream of readers.

Go 1.19+: RWMutex has **writer preference** after waiting goroutines:

- If a writer is waiting, new readers block
- Existing readers finish, then writer acquires lock

- Prevents indefinite starvation

```
// Continuous readers
for i := 0; i < 1000; i++ {
    go func() {
        mu.RLock()
        time.Sleep(time.Millisecond)
        mu.RUnlock()
    }()
}

// Writer
mu.Lock()  // Will eventually acquire (not starved indefinitely)
mu.Unlock()
```

RWMutex vs Atomic for Counters

```
// Option 1: RWMutex
type Counter struct {
    mu    sync.RWMutex
    count int64
}

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

func (c *Counter) Value() int64 {
    c.mu.RLock()
    defer c.mu.RUnlock()
    return c.count
}

// Option 2: atomic (better for counters)
type Counter struct {
    count int64  // atomic
}

func (c *Counter) Inc() {
    atomic.AddInt64(&c.count, 1)
}

func (c *Counter) Value() int64 {
    return atomic.LoadInt64(&c.count)
}
```

Performance:

- RWMutex: ~30-50 ns per operation
- Atomic: ~5-10 ns per operation

Lesson: For simple counters, use atomic (see atomic.md).

When Mutex Beats RWMutex

Scenario 1: High Write Ratio

```
// 40% reads, 60% writes
// Mutex: 150 ns/op
// RWMutex: 180 ns/op (slower due to overhead)
```

Scenario 2: Very Short Critical Sections

```
// Critical section: 10 ns
// Mutex overhead: 20 ns → Total: 30 ns
// RWMutex overhead: 35 ns → Total: 45 ns
// RWMutex is 50% slower!
```

Scenario 3: Single-Core Systems

```
// 1 core: no parallel reads anyway
// RWMutex adds complexity without benefit
```

Real-World Failure: Premature RWMutex Optimization

Company: Analytics platform (2019)

What happened:
Engineers switched from `sync.Mutex` to `sync.RWMutex` for a shared data structure, expecting performance improvement. Performance **decreased by 30%**.

Root cause:

```
// Before (150 ns/op)
type Stats struct {
    mu    sync.Mutex
    count int64
}

// After (195 ns/op - slower!)
type Stats struct {
    mu    sync.RWMutex
    count int64
}

func (s *Stats) Inc() {
    s.mu.Lock()  // 60% of operations
```

```
    s.count++
    s.mu.Unlock()
}

func (s *Stats) Get() int64 {
    s.mu.RLock()  // 40% of operations
    defer s.mu.RUnlock()
    return s.count
}
```

Problem:

- Workload was 60% writes, 40% reads
- RWMutex has higher overhead
- No benefit from parallel reads (writes too frequent)

Fix:

```
// Reverted to sync.Mutex (30% faster)
// OR: Use atomic for simple counter
type Stats struct {
    count int64  // atomic
}

func (s *Stats) Inc() {
    atomic.AddInt64(&s.count, 1)  // 10x faster than mutex
}
```

Lessons:

1. **Profile before optimizing**
2. RWMutex is not always faster
3. For counters, atomic is best
4. Understand your read/write ratio

Decision Tree: Mutex vs RWMutex

```
Is your workload read-heavy (90%+ reads)?
├─ NO → Use sync.Mutex
│   └─ Simpler, lower overhead
│
└─ YES → Continue
    │
    Is the critical section >500ns?
    ├─ NO → Use sync.Mutex
    │   └─ RWMutex overhead not worth it
    │
    └─ YES → Continue
        │
        Will you have concurrent readers?
        ├─ NO (single core or low concurrency) → Use sync.Mutex
        │
```

```
        └─ YES → Use sync.RWMutex
            └─ But benchmark to confirm!
```

Interview Traps

Trap 1: "RWMutex is always better for reads"

Wrong. Only better for read-heavy, high-contention workloads.

Correct answer:
"RWMutex is beneficial when reads significantly outnumber writes (90%+) and there's contention from multiple simultaneous readers. For write-heavy or low-contention workloads, the added complexity and overhead of RWMutex can make it slower than a regular mutex. Always benchmark."

Trap 2: "RLock provides exclusive access"

Wrong. RLock allows concurrent reads.

Correct answer:
"RLock allows multiple concurrent readers—it provides shared, not exclusive, access. Only Lock() provides exclusive access, blocking all readers and writers. Use RLock when you need consistent reads but don't need to prevent other concurrent reads."

Trap 3: "I can upgrade from RLock to Lock"

Wrong. No lock upgrade mechanism.

Correct answer:
"Go's RWMutex doesn't support upgrading a read lock to a write lock. You must RUnlock() then Lock(). This creates a window where another goroutine can acquire the lock, so you may need to double-check conditions after acquiring the write lock."

Trap 4: "RWMutex prevents writer starvation"

Partially correct (as of Go 1.19).

Correct answer:
"As of Go 1.19, RWMutex has starvation prevention: if a writer waits too long, new readers block, allowing the writer to eventually acquire the lock. However, this doesn't eliminate starvation entirely—pathological patterns can still delay writers significantly."

Key Takeaways

1. **RWMutex = multiple concurrent readers, one exclusive writer**
2. **Only beneficial for read-heavy workloads** (90%+ reads)
3. **RLock/RUnlock must pair** (don't mix with Lock/Unlock)
4. **No lock upgrade** (must RUnlock then Lock)
5. **RWMutex has overhead** (~10-20ns more than mutex)
6. **Benchmark before using** (not always faster)
7. **For simple counters, use atomic** (10x faster than RWMutex)
8. **Keep read locks short** (don't hold during slow operations)

What You Should Be Thinking Now

- "When should I use atomic operations instead of mutexes?"
- "How do atomics provide synchronization?"
- "What's the performance difference between mutex and atomic?"
- "What operations does Go's atomic package provide?"

Next: atomic.md - Lock-free synchronization with atomic operations.

Exercises (Do These Before Moving On)

1. Implement a cache with both `sync.Mutex` and `sync.RWMutex`. Benchmark with 95% reads, 5% writes. Compare.
2. Write code that deadlocks trying to upgrade RLock to Lock. Fix it properly.
3. Create a configuration store that reloads rarely (1/hour) but reads frequently (1000/sec). Measure RWMutex vs Mutex.
4. Write a case where RWMutex performs **worse** than Mutex (hint: write-heavy). Verify with benchmarks.
5. Implement writer starvation: continuous readers preventing writer from acquiring lock. Run on pre-1.19 vs post-1.19 Go versions.

Don't continue until you can explain: "Why does RWMutex have higher overhead than Mutex, and when is that overhead worth paying?"