Choosing Concurrency Primitives

Decision Tree

```
Need concurrency?
│
├─ NO → Use sequential code
│
└─ YES → What are you doing?
    │
    ├─ Communicating between goroutines?
    │   │
    │   ├─ One-to-one message → Unbuffered channel
    │   ├─ Producer-consumer → Buffered channel
    │   ├─ Broadcast signal → Close channel or sync.Cond
    │   ├─ Select from multiple → select statement
    │   └─ Request-response → Channel pair or sync.WaitGroup
    │
    ├─ Protecting shared data?
    │   │
    │   ├─ Simple counter/flag → atomic.Int64/Bool
    │   ├─ Read-heavy (>80% reads) → sync.RWMutex
    │   ├─ Read-write balanced → sync.Mutex
    │   ├─ One-time initialization → sync.Once
    │   └─ Map operations → sync.Map or mutex+map
    │
    ├─ Waiting for events?
    │   │
    │   ├─ Wait for all → sync.WaitGroup
    │   ├─ Wait for any → select with channels
    │   ├─ Timeout/cancellation → context.Context
    │   └─ Periodic events → time.Ticker
    │
    └─ Controlling access?
        │
        ├─ Limit concurrent ops → Buffered channel (semaphore)
        ├─ Rate limiting → golang.org/x/time/rate
        ├─ One-time execution → sync.Once
        └─ Complex coordination → State machine with channels
```

Detailed Guidelines

1. Channels: For Communication

Use when:

- Passing data between goroutines
- Signaling events
- Coordinating work
- Building pipelines

Unbuffered Channel

When: Synchronization required (sender waits for receiver).

```
// Example: Handoff
ch := make(chan Task)

go func() {
    task := <-ch  // Blocks until sender
    process(task)
}()

ch <- createTask()  // Blocks until receiver ready
```

Properties:

- Synchronizes sender and receiver
- Zero capacity
- Guarantees delivery before sender continues

Buffered Channel

When: Decouple sender and receiver, handle bursts.

```
// Example: Work queue
ch := make(chan Task, 100)  // Buffer 100 tasks

// Producer doesn't block until buffer full
for i := 0; i < 1000; i++ {
    ch <- Task{ID: i}
}
```

Properties:

- Sender blocks only when full
- Receiver blocks only when empty
- Good for smoothing bursty work

Buffer size guidelines:

- Small (1-10): Occasional bursts
- Medium (10-100): Producer-consumer decoupling
- Large (100+): High-throughput pipelines
- Formula: `buffer = expected burst size × 2`

Channel as Semaphore

When: Limit concurrent operations.

```
// Example: Limit to 10 concurrent operations
sem := make(chan struct{}, 10)

for _, item := range items {
    sem <- struct{}{}  // Acquire
```

```
    go func(item Item) {
        defer func() { <-sem }()  // Release
        process(item)
    }(item)
}
```

2. Mutexes: For Protecting Data

Use when:

- Protecting in-memory data structures
- Short critical sections
- No communication needed

sync.Mutex

When: Balanced read-write or write-heavy.

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

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

func (c *Counter) Get() int {
    c.mu.Lock()
    defer c.mu.Unlock()
    return c.value
}
```

Performance: ~20-30ns uncontended, ~100ns contended

Use when:

- Reads and writes both frequent
- Critical section < 100ns
- Simple data protection

sync.RWMutex

When: Read-heavy workload (>80% reads).

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

func (c *Cache) Get(key string) string {
    c.mu.RLock()  // Multiple readers OK
```

```
    defer c.mu.RUnlock()
    return c.data[key]
}

func (c *Cache) Set(key, value string) {
    c.mu.Lock()  // Exclusive write
    defer c.mu.Unlock()
    c.data[key] = value
}
```

Performance: Read ~15ns, Write ~110ns (more expensive than Mutex)

Use when:

- Reads >> Writes (at least 5:1 ratio)
- Benchmark to confirm benefit

3. Atomics: For Simple Values

Use when:

- Single value (int, bool, pointer)
- No need for mutex
- High-frequency updates

```
type Stats struct {
    requests atomic.Int64
    errors   atomic.Int64
}

func (s *Stats) RecordRequest() {
    s.requests.Add(1)  // Lock-free
}

func (s *Stats) GetRequests() int64 {
    return s.requests.Load()  // Lock-free
}
```

Performance: ~5-10ns (fastest)

Use when:

- Counters, flags, pointers
- No compound operations (can't atomically increment two values)

Don't use when:

- Need to update multiple values together
- Complex logic in critical section

4. sync.WaitGroup: Wait for Completion

When: Wait for multiple goroutines to finish.

```go
var wg sync.WaitGroup

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

wg.Wait()  // Blocks until all Done()
```

Use when:

- Need to wait for all goroutines
- No results to collect (use channels if results needed)
- No early termination (use context for cancellation)

5. sync.Once: One-Time Initialization

When: Lazy initialization, called multiple times but execute once.

```go
var (
    instance *Singleton
    once     sync.Once
)

func GetInstance() *Singleton {
    once.Do(func() {
        instance = &Singleton{}  // Only runs once
    })
    return instance
}
```

Use when:

- Singleton pattern
- Expensive initialization
- Thread-safe lazy loading

6. sync.Cond: Complex Waiting

When: Multiple goroutines wait for condition, broadcast signal.

```go
type Queue struct {
    mu    sync.Mutex
    cond  *sync.Cond
    items []Item
}

func NewQueue() *Queue {
    q := &Queue{}
```

```
    q.cond = sync.NewCond(&q.mu)
    return q
}

func (q *Queue) Push(item Item) {
    q.mu.Lock()
    q.items = append(q.items, item)
    q.cond.Signal()  // Wake one waiter
    q.mu.Unlock()
}

func (q *Queue) Pop() Item {
    q.mu.Lock()
    defer q.mu.Unlock()

    for len(q.items) == 0 {
        q.cond.Wait()  // Release lock, wait, reacquire
    }

    item := q.items[0]
    q.items = q.items[1:]
    return item
}
```

Use when:

- Complex condition checking
- Broadcast to multiple waiters
- Need to check condition under same lock

Prefer channels when possible (simpler, more idiomatic).

7. context.Context: Cancellation and Timeouts

When: Need to cancel operations, propagate deadlines.

```
func fetchWithTimeout(ctx context.Context, url string) ([]byte, error) {
    ctx, cancel := context.WithTimeout(ctx, 5*time.Second)
    defer cancel()

    req, _ := http.NewRequestWithContext(ctx, "GET", url, nil)
    resp, err := http.DefaultClient.Do(req)
    if err != nil {
        return nil, err
    }
    defer resp.Body.Close()

    return ioutil.ReadAll(resp.Body)
}
```

Use when:

- HTTP requests

- Database queries
- Any long-running operation
- Need to cancel tree of goroutines

8. sync.Map: Concurrent Map

When: Map with concurrent reads/writes, keys not known upfront.

```
var cache sync.Map

func get(key string) (string, bool) {
    val, ok := cache.Load(key)
    if !ok {
        return "", false
    }
    return val.(string), true
}

func set(key, value string) {
    cache.Store(key, value)
}
```

Use when:

- Map used by multiple goroutines
- Keys are write-once, read-many
- Keys not known initially (can't shard)

Don't use when:

- Map changes frequently
- Can shard by key (better: multiple mutexguarded maps)

Comparison Table

Primitive	Use Case	Performance	Complexity
Atomic	Simple counters, flags	~5-10ns	Low
Mutex	Protect data structure	~20-30ns	Low
RWMutex	Read-heavy workload	Read: ~15ns, Write: ~110ns	Low
Unbuffered channel	Synchronization, handoff	~50-100ns	Medium
Buffered channel	Producer-consumer, queue	~50-100ns	Medium
WaitGroup	Wait for goroutines	N/A (blocking)	Low
Once	One-time init	Same as Mutex	Low
Cond	Complex waiting	Same as Mutex	High
Context	Cancellation	N/A (signaling)	Medium

sync.Map	Concurrent map	Varies	Medium

Real-World Examples

Example 1: HTTP Rate Limiter

Requirement: Limit requests per IP.

Choice: `sync.Map` (per-IP state) + `golang.org/x/time/rate`

```
type RateLimiter struct {
    limiters sync.Map  // IP → *rate.Limiter
}

func (rl *RateLimiter) Allow(ip string) bool {
    limiter, _ := rl.limiters.LoadOrStore(ip, rate.NewLimiter(10, 20))
    return limiter.(*rate.Limiter).Allow()
}
```

Why: Keys (IPs) not known upfront, concurrent access, write-once-read-many.

Example 2: Worker Pool

Requirement: Fixed number of workers processing tasks.

Choice: Buffered channel (task queue) + WaitGroup (track completion)

```
type Pool struct {
    tasks   chan Task
    results chan Result
    wg      sync.WaitGroup
}

func NewPool(workers int) *Pool {
    p := &Pool{
        tasks:   make(chan Task, workers*2),
        results: make(chan Result, workers*2),
    }

    for i := 0; i < workers; i++ {
        p.wg.Add(1)
        go p.worker()
    }

    return p
}

func (p *Pool) worker() {
    defer p.wg.Done()
    for task := range p.tasks {
        p.results <- task.Execute()
```

```
    }
}
```

Why: Channels for communication, WaitGroup to wait for all workers.

Example 3: Cache with Expiration

Requirement: In-memory cache with TTL.

Choice: Sharded mutexes (reduce contention) + time.Ticker (cleanup)

```
type Cache struct {
    shards []*Shard
    ticker *time.Ticker
}

type Shard struct {
    mu   sync.RWMutex
    data map[string]*Entry
}

type Entry struct {
    value   string
    expires time.Time
}

func (c *Cache) Get(key string) (string, bool) {
    shard := c.getShard(key)
    shard.mu.RLock()
    defer shard.mu.RUnlock()

    entry, ok := shard.data[key]
    if !ok || time.Now().After(entry.expires) {
        return "", false
    }
    return entry.value, true
}

func (c *Cache) cleanup() {
    for range c.ticker.C {
        for _, shard := range c.shards {
            shard.mu.Lock()
            for key, entry := range shard.data {
                if time.Now().After(entry.expires) {
                    delete(shard.data, key)
                }
            }
            shard.mu.Unlock()
        }
    }
}
```

Why: Sharded RWMutex (read-heavy), Ticker (periodic cleanup).

Example 4: Connection Pool

Requirement: Reusable database connections.

Choice: Buffered channel (available connections) + Mutex (stats)

```
type Pool struct {
    conns chan *Conn
    stats struct {
        mu      sync.Mutex
        active  int
        created int
    }
}

func (p *Pool) Get() *Conn {
    select {
    case conn := <-p.conns:
        return conn
    default:
        // No available, create new
        p.stats.mu.Lock()
        p.stats.created++
        p.stats.active++
        p.stats.mu.Unlock()

        return newConn()
    }
}

func (p *Pool) Put(conn *Conn) {
    select {
    case p.conns <- conn:
        // Returned to pool
    default:
        // Pool full, close connection
        conn.Close()

        p.stats.mu.Lock()
        p.stats.active--
        p.stats.mu.Unlock()
    }
}
```

Why: Channel (connection pool), Mutex (protect stats).

Common Mistakes

Mistake 1: Channel When Mutex Better

```go
// BAD: Channel for protecting shared state
type Counter struct {
    ops chan func()
}

func (c *Counter) Inc() {
    c.ops <- func() { c.value++ }
}

// GOOD: Mutex for simple protection
type Counter struct {
    mu    sync.Mutex
    value int
}

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

Mistake 2: Mutex When Atomic Better

```go
// BAD: Mutex for simple counter
type Stats struct {
    mu       sync.Mutex
    requests int64
}

func (s *Stats) Inc() {
    s.mu.Lock()
    s.requests++
    s.mu.Unlock()
}

// GOOD: Atomic for simple value
type Stats struct {
    requests atomic.Int64
}

func (s *Stats) Inc() {
    s.requests.Add(1)
}
```

Mistake 3: RWMutex When Not Read-Heavy

```go
// BAD: RWMutex for balanced workload (50% reads, 50% writes)
// RWMutex has overhead, slower than Mutex here
```

```
// GOOD: Benchmark and use Mutex if not read-heavy
```

Decision Flowchart Summary

1. **Simple value (counter, flag)** → Atomic
2. **Communicate/coordinate** → Channel
3. **Protect data structure** → Mutex (or RWMutex if read-heavy)
4. **Wait for goroutines** → WaitGroup
5. **One-time init** → sync.Once
6. **Cancellation/timeout** → context.Context
7. **Limit concurrency** → Buffered channel (semaphore)
8. **Concurrent map** → sync.Map (or sharded mutex+map)

Interview Questions

Q: "Channel vs mutex: when to use each?"

"Channels for communication (passing data, signaling), mutexes for protection (shared data structure). Rule: 'Share memory by communicating' → channels. 'Protect shared memory' → mutexes. Example: Worker pool uses channels (communication), cache uses mutex (protection). Benchmark if unclear: atomic < mutex < channel in performance."

Q: "When to use RWMutex instead of Mutex?"

"When reads >> writes (at least 5:1 ratio). RWMutex allows multiple concurrent readers but write is more expensive than Mutex. Benchmark to confirm: if workload is 90% reads, RWMutex ~3x faster. If 50/50, Mutex faster. Example: Configuration cache (read-heavy) uses RWMutex, user session store (balanced) uses Mutex."

Q: "Buffered vs unbuffered channel?"

"Unbuffered: Synchronize sender and receiver (handoff pattern, guarantee delivery). Buffered: Decouple sender/receiver, handle bursts (work queue, smooth bursty traffic). Size buffer = expected burst × 2. Example: Task queue buffered (100 tasks), synchronization signal unbuffered (immediate handoff)."

Q: "When to use atomic instead of mutex?"

"Atomic for single simple values (int64, bool, pointer) with no compound operations. Faster (~5ns vs ~20ns), lock-free. Use mutex when: 1) Multiple values updated together, 2) Complex logic in critical section, 3) Struct field updates. Example: Request counter uses atomic, cache (map) uses mutex."

Key Takeaways

1. **Atomic < Mutex < RWMutex < Channel (performance)**
2. **Channels for communication, mutexes for protection**
3. **Use atomic for simple counters/flags**
4. **RWMutex only if read-heavy (>80% reads)**
5. **Buffered channel for queues, unbuffered for sync**
6. **WaitGroup to wait, context to cancel**
7. **sync.Once for one-time initialization**
8. **Benchmark when unsure**
9. **Simple is better than complex**
10. **"Simplicity is prerequisite for reliability"**

Exercises

1. Implement cache using: a) Mutex, b) RWMutex, c) sync.Map. Benchmark with different read/write ratios.
2. Build rate limiter using three approaches: a) Atomic counter, b) Channel semaphore, c) time/rate. Compare.
3. Create connection pool using channels, then refactor to use sync.Pool. Measure performance.
4. Implement stats collector with atomic operations, then with mutex. Benchmark contention.
5. Design concurrent datastructure requiring 3+ primitives, justify each choice.

Next: scaling-strategies.md - Horizontal and vertical scaling patterns.