Semaphore Pattern

What is a Semaphore?

Semaphore: A concurrency primitive that limits the number of goroutines that can access a resource simultaneously.

Types:

- **Binary semaphore:** Limits to 1 (equivalent to mutex)
- **Counting semaphore:** Limits to N

Purpose:

- **Resource limiting:** Bound concurrent database connections, HTTP requests, file handles
- **Rate limiting:** Control throughput
- **Backpressure:** Prevent overwhelming downstream systems

Implementing Semaphore with Buffered Channel

```
type Semaphore chan struct{}

func NewSemaphore(maxConcurrency int) Semaphore {
    return make(chan struct{}, maxConcurrency)
}

func (s Semaphore) Acquire() {
    s <- struct{}{}  // Block if buffer full
}

func (s Semaphore) Release() {
    <-s  // Free slot
}

// Usage:
sem := NewSemaphore(10)  // Max 10 concurrent

for _, task := range tasks {
    sem.Acquire()
    go func(t Task) {
        defer sem.Release()
        process(t)
    }(task)
}

// Wait for all to complete
for i := 0; i < cap(sem); i++ {
    sem.Acquire()
}
```

How it works:

- Buffered channel with size N
- Send blocks when N goroutines hold semaphore
- Receive frees slot for next goroutine

Weighted Semaphore

```
type WeightedSemaphore struct {
    weights chan int
    capacity int
}

func NewWeightedSemaphore(capacity int) *WeightedSemaphore {
    return &WeightedSemaphore{
        weights:  make(chan int, capacity),
        capacity: capacity,
    }
}

func (s *WeightedSemaphore) Acquire(n int) {
    for i := 0; i < n; i++ {
        s.weights <- 1
    }
}

func (s *WeightedSemaphore) Release(n int) {
    for i := 0; i < n; i++ {
        <-s.weights
    }
}

// Usage: Large tasks acquire more slots
sem := NewWeightedSemaphore(100)

// Small task (requires 10 units)
sem.Acquire(10)
defer sem.Release(10)

// Large task (requires 50 units)
sem.Acquire(50)
defer sem.Release(50)
```

Standard Library: golang.org/x/sync/semaphore

```
import "golang.org/x/sync/semaphore"

sem := semaphore.NewWeighted(10)  // Max weight 10

// Acquire with context
ctx := context.Background()
```

```go
if err := sem.Acquire(ctx, 1); err != nil {
    // Context canceled
}
defer sem.Release(1)

// Try acquire (non-blocking)
if sem.TryAcquire(1) {
    defer sem.Release(1)
    // Got lock
} else {
    // Would block
}
```

Real-World Example: Database Connection Pool

```go
type ConnectionPool struct {
    sem        Semaphore
    createConn func() (*sql.DB, error)
    conns      chan *sql.DB
}

func NewConnectionPool(maxConns int, create func() (*sql.DB, error)) *ConnectionPool
{
    return &ConnectionPool{
        sem:        NewSemaphore(maxConns),
        createConn: create,
        conns:      make(chan *sql.DB, maxConns),
    }
}

func (p *ConnectionPool) Acquire(ctx context.Context) (*sql.DB, error) {
    // Try to get existing connection (non-blocking)
    select {
    case conn := <-p.conns:
        return conn, nil
    default:
    }

    // Acquire semaphore slot
    select {
    case p.sem.Acquire():
    case <-ctx.Done():
        return nil, ctx.Err()
    }

    // Create new connection
    conn, err := p.createConn()
    if err != nil {
        p.sem.Release()
        return nil, err
```

```
    }

    return conn, nil
}

func (p *ConnectionPool) Release(conn *sql.DB) {
    // Return connection to pool
    select {
    case p.conns <- conn:
        // Connection returned to pool
    default:
        // Pool full, close connection
        conn.Close()
        p.sem.Release()
    }
}

// Usage:
pool := NewConnectionPool(10, func() (*sql.DB, error) {
    return sql.Open("postgres", "...")
})

conn, err := pool.Acquire(ctx)
if err != nil {
    return err
}
defer pool.Release(conn)

// Use connection
rows, err := conn.Query("SELECT ...")
```

Real-World Example: Rate-Limited HTTP Client

```
type RateLimitedClient struct {
    client    *http.Client
    semaphore Semaphore
    rps       int  // Requests per second
    ticker    *time.Ticker
}

func NewRateLimitedClient(maxConcurrency, rps int) *RateLimitedClient {
    rlc := &RateLimitedClient{
        client:    &http.Client{},
        semaphore: NewSemaphore(maxConcurrency),
        rps:       rps,
        ticker:    time.NewTicker(time.Second / time.Duration(rps)),
    }
    return rlc
}
```

```
func (rlc *RateLimitedClient) Get(ctx context.Context, url string) (*http.Response,
error) {
    // Wait for rate limit token
    select {
    case <-rlc.ticker.C:
    case <-ctx.Done():
        return nil, ctx.Err()
    }

    // Acquire semaphore (limit concurrent requests)
    select {
    case rlc.semaphore.Acquire():
        defer rlc.semaphore.Release()
    case <-ctx.Done():
        return nil, ctx.Err()
    }

    // Make request
    req, _ := http.NewRequestWithContext(ctx, "GET", url, nil)
    return rlc.client.Do(req)
}

// Usage:
client := NewRateLimitedClient(10, 100)  // Max 10 concurrent, 100 req/sec

for _, url := range urls {
    go func(u string) {
        resp, err := client.Get(context.Background(), u)
        if err != nil {
            log.Printf("Error: %v", err)
            return
        }
        defer resp.Body.Close()
        // Process response
    }(url)
}
```

Semaphore with Timeout

```
func (s Semaphore) AcquireWithTimeout(timeout time.Duration) bool {
    select {
    case s <- struct{}{}:
        return true
    case <-time.After(timeout):
        return false  // Timeout
    }
}

// Usage:
if !sem.AcquireWithTimeout(5 * time.Second) {
```

```
    log.Println("Timeout acquiring semaphore")
    return
}
defer sem.Release()
```

Semaphore with Context

```
func (s Semaphore) AcquireContext(ctx context.Context) error {
    select {
    case s <- struct{}{}:
        return nil
    case <-ctx.Done():
        return ctx.Err()
    }
}

// Usage:
ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
defer cancel()

if err := sem.AcquireContext(ctx); err != nil {
    return err
}
defer sem.Release()
```

Bounded Parallelism Pattern

```
func processTasks(tasks []Task, maxConcurrency int) []Result {
    sem := NewSemaphore(maxConcurrency)
    results := make([]Result, len(tasks))
    var wg sync.WaitGroup

    for i, task := range tasks {
        wg.Add(1)
        sem.Acquire()

        go func(idx int, t Task) {
            defer wg.Done()
            defer sem.Release()

            results[idx] = process(t)
        }(i, task)
    }

    wg.Wait()
    return results
}
```

Dynamic Semaphore (Adaptive Concurrency)

```
type DynamicSemaphore struct {
    current   chan struct{}
    max       int
    min       int
    mu        sync.Mutex
    latencies []time.Duration
}

func NewDynamicSemaphore(min, max int) *DynamicSemaphore {
    ds := &DynamicSemaphore{
        current: make(chan struct{}, min),
        max:     max,
        min:     min,
    }

    go ds.adjustConcurrency()
    return ds
}

func (ds *DynamicSemaphore) adjustConcurrency() {
    ticker := time.NewTicker(5 * time.Second)
    defer ticker.Stop()

    for range ticker.C {
        ds.mu.Lock()

        avgLatency := ds.averageLatency()
        currentCap := cap(ds.current)

        if avgLatency < 100*time.Millisecond && currentCap < ds.max {
            // Increase concurrency
            ds.scaleUp()
        } else if avgLatency > 500*time.Millisecond && currentCap > ds.min {
            // Decrease concurrency
            ds.scaleDown()
        }

        ds.latencies = nil  // Reset
        ds.mu.Unlock()
    }
}

func (ds *DynamicSemaphore) Acquire(ctx context.Context) (func(), error) {
    start := time.Now()

    select {
    case ds.current <- struct{}{}:
    case <-ctx.Done():
```

```
        return nil, ctx.Err()
    }

    // Release function that records latency
    return func() {
        latency := time.Since(start)
        ds.mu.Lock()
        ds.latencies = append(ds.latencies, latency)
        ds.mu.Unlock()

        <-ds.current
    }, nil
}
```

Comparing Semaphore vs. Worker Pool

Aspect	Semaphore	Worker Pool
Goroutines	Created per task	Pre-created workers
Memory	Higher (goroutine per task)	Lower (fixed workers)
Startup	Instant (buffered channel)	Upfront (create workers)
Shutdown	Automatic (goroutines end)	Manual (close channels)
Best for	Short-lived tasks	Long-running processing

Common Mistakes

Mistake 1: Forgetting to Release

```
// WRONG: Semaphore slot never released
sem.Acquire()
if err := process(); err != nil {
    return err  // Leaks semaphore slot!
}
sem.Release()
```

Fix: Always defer

```
sem.Acquire()
defer sem.Release()

if err := process(); err != nil {
    return err
}
```

Mistake 2: Acquiring Multiple Times

```
// WRONG: Deadlock on same goroutine
sem.Acquire()
// ... do work ...
sem.Acquire()  // Deadlock if sem full!
```

Fix: Acquire once

```
sem.Acquire()
defer sem.Release()

// All work using single slot
```

Mistake 3: Channel Size Mismatch

```
// WRONG: Waiting for completion
sem := NewSemaphore(10)

for _, task := range tasks {
    sem.Acquire()
    go func(t Task) {
        defer sem.Release()
        process(t)
    }(task)
}

// This doesn't wait for completion!
// Only ensures max 10 concurrent
```

Fix: Use WaitGroup

```
sem := NewSemaphore(10)
var wg sync.WaitGroup

for _, task := range tasks {
    wg.Add(1)
    sem.Acquire()
    go func(t Task) {
        defer wg.Done()
        defer sem.Release()
        process(t)
    }(task)
}

wg.Wait()  // Wait for all tasks
```

Performance Trade-offs

Semaphore advantages:

- Simple to understand and implement
- Low overhead (buffered channel)
- Flexible (create goroutines on demand)

Semaphore disadvantages:

- Goroutine creation overhead per task
- Memory usage grows with task count
- No task queue (tasks created immediately)

When to use semaphore:

- Limit concurrent access to resource
- Short-lived tasks
- Variable task arrival rate
- Simple bounded parallelism

When to use worker pool:

- Long-running task processing
- Fixed resource budget
- Task queue with backpressure
- Predictable memory usage

Interview Questions

Q: "How does semaphore differ from mutex?"

"Mutex is binary semaphore (count=1) ensuring exclusive access—only one goroutine in critical section. Semaphore allows N goroutines (count=N) for resource limiting. Mutex protects shared state; semaphore limits concurrency. Implementation: mutex uses atomic ops + OS facilities; semaphore uses buffered channel."

Q: "What's the difference between semaphore and channel?"

"Semaphore is for counting/limiting (how many can proceed). Channel is for communication (passing data). Semaphore implemented using buffered channel, but semantics differ: semaphore acquire/release pattern, channel send/receive. Use semaphore for resource limiting, channels for data flow."

Q: "How do you prevent semaphore leaks?"

"Always pair Acquire with Release using defer. Use context for cancellation—ensure Release called even on error/timeout. Pattern: `sem.Acquire(); defer sem.Release()`. Never return without releasing unless intentional (long-held lock)."

Q: "When would you use weighted semaphore?"

"When resources have different costs. Example: large tasks require more memory/CPU. Small task acquires weight 1, large task acquires weight 10. Prevents oversubscription—total weight can't exceed capacity. Also useful for fairness—prevent one large task from starving small tasks."

Key Takeaways

1. **Semaphore = limit concurrent access to N**
2. **Implement with buffered channel (size N)**
3. **Always defer Release after Acquire**
4. **Use context for cancellation**

5. **Weighted semaphore for variable-cost resources**
6. **Combine with WaitGroup for completion tracking**
7. **Semaphore limits concurrency, not total goroutines**
8. **Use for short-lived tasks; worker pool for long-lived**
9. **TryAcquire for non-blocking attempt**
10. **Adaptive semaphore adjusts to load**

Exercises

1. Implement semaphore using mutex and condition variable instead of channel.

2. Build rate-limited web scraper: max 10 concurrent, 50 req/sec.

3. Create weighted semaphore: small tasks (weight 1), large tasks (weight 5), max weight 20.

4. Add timeout to semaphore: return error if can't acquire in 5 seconds.

5. Implement adaptive semaphore that increases concurrency when latency low, decreases when high.

Next: rate-limiter.md - Implementing token bucket and leaky bucket algorithms.