Whiteboard Design Problems

General Approach

Step 1: Clarify Requirements (2-3 minutes)

Ask:

- Throughput (requests/second)?
- Latency requirements (p50, p95, p99)?
- Read vs write ratio?
- Stateful or stateless?
- Single machine or distributed?
- Consistency requirements?

Step 2: Identify Concurrency Model (2 minutes)

Choose based on:

- **I/O-bound** → Many goroutines (10× CPUs)
- **CPU-bound** → Worker pool (= NumCPU)
- **Mixed** → Pipeline (I/O stage + CPU stage)
- **Stateful** → Actor/mailbox pattern

Step 3: Sketch Architecture (5 minutes)

- Components (boxes)
- Channels (arrows)
- Worker pools (group of boxes)
- Synchronization points (locks, WaitGroups)

Step 4: Discuss Failure Modes (5 minutes)

- Timeouts
- Circuit breakers
- Graceful degradation
- Goroutine leaks

Step 5: Code Key Components (10-15 minutes)

Write skeleton code for critical paths.

Problem 1: URL Shortener

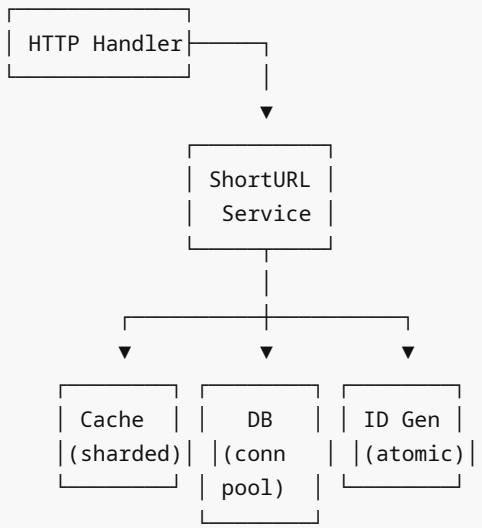
Requirements Clarification

Q: "Design a concurrent URL shortener."

Ask:

- Throughput: 1000 writes/sec, 10,000 reads/sec
- Latency: < 10ms p99
- Storage: Single machine initially (1M URLs)
- Uniqueness: Must be unique per URL

High-Level Design



Key Components

```
type ShortenerService struct {
    cache  *ShardedCache
    db     *sql.DB
    idGen  *IDGenerator

    // Rate limiting
    limiter *rate.Limiter
}

func NewShortenerService(shards int) *ShortenerService {
    return &ShortenerService{
        cache:   NewShardedCache(shards),  // 16 shards
        db:      initDB(),
        idGen:   NewIDGenerator(),
        limiter: rate.NewLimiter(1000, 2000),  // 1000/s, burst 2000
    }
}

func (s *ShortenerService) Shorten(ctx context.Context, longURL string) (string,
error) {
    // Rate limit
    if err := s.limiter.Wait(ctx); err != nil {
        return "", err
    }

    // Check cache first
    if shortURL, ok := s.cache.Get(longURL); ok {
        return shortURL, nil
    }
```

```
    // Generate short URL
    id := s.idGen.Next()
    shortURL := encode(id)

    // Store in DB (with timeout)
    ctx, cancel := context.WithTimeout(ctx, 100*time.Millisecond)
    defer cancel()

    if err := s.db.ExecContext(ctx,
        "INSERT INTO urls (short, long) VALUES (?, ?)",
        shortURL, longURL); err != nil {
        return "", err
    }

    // Cache it
    s.cache.Set(longURL, shortURL)

    return shortURL, nil
}

func (s *ShortenerService) Resolve(ctx context.Context, shortURL string) (string,
error) {
    // Check cache
    if longURL, ok := s.cache.Get(shortURL); ok {
        return longURL, nil
    }

    // Query DB
    ctx, cancel := context.WithTimeout(ctx, 50*time.Millisecond)
    defer cancel()

    var longURL string
    err := s.db.QueryRowContext(ctx,
        "SELECT long FROM urls WHERE short = ?",
        shortURL).Scan(&longURL)
    if err != nil {
        return "", err
    }

    // Cache it
    s.cache.Set(shortURL, longURL)

    return longURL, nil
}
```

ID Generator (Thread-Safe)

```
type IDGenerator struct {
    counter atomic.Uint64
}
```

```go
func (g *IDGenerator) Next() uint64 {
    return g.counter.Add(1)
}
```

Sharded Cache (Reduce Contention)

```go
type ShardedCache struct {
    shards []*Shard
}

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

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

    val, ok := shard.data[key]
    return val, ok
}

func (c *ShardedCache) getShard(key string) *Shard {
    hash := fnv32(key)
    return c.shards[hash%uint32(len(c.shards))]
}
```

Discussion Points

Concurrency Strategy:

- Read-heavy (10:1) → RWMutex in sharded cache
- Sharding reduces lock contention (16 shards = 1/16 contention)
- Atomic counter for ID generation (lock-free)

Failure Handling:

- Timeouts on all DB ops (prevent thread pool exhaustion)
- Rate limiting (prevent overload)
- Cache fallback if DB slow

Scalability:

- Horizontal: Add more app servers (stateless)
- Vertical: Increase shards, DB connection pool

Problem 2: Rate-Limited API Client

Requirements

Q: "Design a concurrent API client that respects rate limits."

Requirements:

- 100 requests/second limit
- Handle bursts up to 200
- Fail fast if limit exceeded
- Retry on failures

Design

```
type APIClient struct {
    client  *http.Client
    limiter *rate.Limiter

    // Circuit breaker
    breaker *CircuitBreaker

    // Retry config
    maxRetries int
}

func NewAPIClient() *APIClient {
    return &APIClient{
        client: &http.Client{
            Timeout: 10 * time.Second,
        },
        limiter:    rate.NewLimiter(100, 200),  // 100/s, burst 200
        breaker:    NewCircuitBreaker(5, 30*time.Second),
        maxRetries: 3,
    }
}

func (c *APIClient) Get(ctx context.Context, url string) (*Response, error) {
    // Rate limiting
    if err := c.limiter.Wait(ctx); err != nil {
        return nil, err
    }

    // Circuit breaker
    var resp *Response
    err := c.breaker.Call(func() error {
        var err error
        resp, err = c.doWithRetry(ctx, url)
        return err
    })

    return resp, err
}
```

```go
func (c *APIClient) doWithRetry(ctx context.Context, url string) (*Response, error)
{
    backoff := 100 * time.Millisecond

    for attempt := 0; attempt < c.maxRetries; attempt++ {
        req, err := http.NewRequestWithContext(ctx, "GET", url, nil)
        if err != nil {
            return nil, err
        }

        resp, err := c.client.Do(req)
        if err == nil && resp.StatusCode < 500 {
            return parseResponse(resp), nil
        }

        // Retry on 5xx or network error
        if attempt < c.maxRetries-1 {
            select {
            case <-time.After(backoff):
                backoff *= 2  // Exponential backoff
            case <-ctx.Done():
                return nil, ctx.Err()
            }
        }
    }

    return nil, fmt.Errorf("max retries exceeded")
}
```

Circuit Breaker

```go
type CircuitBreaker struct {
    maxFailures int
    timeout     time.Duration

    mu          sync.Mutex
    failures    int
    lastAttempt time.Time
    state       State
}

type State int

const (
    StateClosed State = iota
    StateOpen
    StateHalfOpen
)

func (cb *CircuitBreaker) Call(fn func() error) error {
    cb.mu.Lock()
```

```
    state := cb.state

    if state == StateOpen {
        if time.Since(cb.lastAttempt) > cb.timeout {
            cb.state = StateHalfOpen
            cb.mu.Unlock()
            return cb.attempt(fn)
        }
        cb.mu.Unlock()
        return ErrCircuitOpen
    }

    cb.mu.Unlock()
    return cb.attempt(fn)
}

func (cb *CircuitBreaker) attempt(fn func() error) error {
    err := fn()

    cb.mu.Lock()
    defer cb.mu.Unlock()

    if err != nil {
        cb.failures++
        cb.lastAttempt = time.Now()

        if cb.failures >= cb.maxFailures {
            cb.state = StateOpen
        }
        return err
    }

    // Success
    cb.failures = 0
    cb.state = StateClosed
    return nil
}
```

Problem 3: Parallel File Processor

Requirements

Q: "Process 1000 large files concurrently."

Constraints:

- Each file 100MB
- CPU-bound processing (compression)
- Limited memory (can't load all at once)

Design: Pipeline

```
Read Files → Process → Write Results
(10 workers) (NumCPU) (5 workers)
```

```go
type FileProcessor struct {
    readWorkers    int
    processWorkers int
    writeWorkers   int

    readCh    chan string      // File paths
    processCh chan FileData    // Raw data
    writeCh   chan ProcessedData
}

func NewFileProcessor() *FileProcessor {
    cpus := runtime.NumCPU()

    return &FileProcessor{
        readWorkers:    10,     // I/O-bound
        processWorkers: cpus,   // CPU-bound
        writeWorkers:   5,      // I/O-bound

        readCh:    make(chan string, 100),
        processCh: make(chan FileData, 100),
        writeCh:   make(chan ProcessedData, 100),
    }
}

func (fp *FileProcessor) Start(ctx context.Context) {
    // Stage 1: Read files
    for i := 0; i < fp.readWorkers; i++ {
        go fp.reader(ctx)
    }

    // Stage 2: Process (CPU-bound)
    for i := 0; i < fp.processWorkers; i++ {
        go fp.processor(ctx)
    }

    // Stage 3: Write results
    for i := 0; i < fp.writeWorkers; i++ {
        go fp.writer(ctx)
    }
}

func (fp *FileProcessor) reader(ctx context.Context) {
    for {
        select {
        case path := <-fp.readCh:
            data, err := ioutil.ReadFile(path)
            if err != nil {
                log.Error(err)
```

```
                continue
            }

            fp.processCh <- FileData{Path: path, Data: data}

        case <-ctx.Done():
            return
        }
    }
}

func (fp *FileProcessor) processor(ctx context.Context) {
    for {
        select {
        case fileData := <-fp.processCh:
            // CPU-intensive work
            processed := compress(fileData.Data)

            fp.writeCh <- ProcessedData{
                Path: fileData.Path,
                Data: processed,
            }

        case <-ctx.Done():
            return
        }
    }
}

func (fp *FileProcessor) writer(ctx context.Context) {
    for {
        select {
        case processed := <-fp.writeCh:
            outPath := processed.Path + ".gz"
            if err := ioutil.WriteFile(outPath, processed.Data, 0644); err != nil {
                log.Error(err)
            }

        case <-ctx.Done():
            return
        }
    }
}

func (fp *FileProcessor) Process(files []string) {
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()

    fp.Start(ctx)

    // Submit files
    for _, file := range files {
```

```
        fp.readCh <- file
    }

    close(fp.readCh)

    // Wait for completion (track with WaitGroup in real impl)
}
```

Problem 4: Concurrent Cache with Expiration

Requirements

Q: "Design thread-safe cache with TTL."

Features:

- Get/Set operations
- LRU eviction (max 10,000 entries)
- Expiration (TTL per entry)
- High read throughput

Design

```
type Cache struct {
    shards []*Shard
    ttl    time.Duration
}

type Shard struct {
    mu      sync.RWMutex
    data    map[string]*Entry
    lru     *list.List
    maxSize int
}

type Entry struct {
    key     string
    value   interface{}
    expires time.Time
    element *list.Element
}

func NewCache(shards int, ttl time.Duration) *Cache {
    c := &Cache{
        shards: make([]*Shard, shards),
        ttl:    ttl,
    }

    for i := 0; i < shards; i++ {
        c.shards[i] = &Shard{
            data:    make(map[string]*Entry),
```

```
            lru:     list.New(),
            maxSize: 10000 / shards,
        }
    }

    // Background cleanup
    go c.cleanup()

    return c
}

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

    entry, ok := shard.data[key]
    if !ok {
        shard.mu.RUnlock()
        return nil, false
    }

    // Check expiration
    if time.Now().After(entry.expires) {
        shard.mu.RUnlock()

        // Upgrade to write lock to delete
        shard.mu.Lock()
        delete(shard.data, key)
        shard.lru.Remove(entry.element)
        shard.mu.Unlock()

        return nil, false
    }

    // LRU: Move to front (need write lock)
    shard.mu.RUnlock()
    shard.mu.Lock()
    shard.lru.MoveToFront(entry.element)
    shard.mu.Unlock()

    return entry.value, true
}

func (c *Cache) Set(key string, value interface{}) {
    shard := c.getShard(key)
    shard.mu.Lock()
    defer shard.mu.Unlock()

    // Check if exists
    if entry, ok := shard.data[key]; ok {
        entry.value = value
        entry.expires = time.Now().Add(c.ttl)
```

```
        shard.lru.MoveToFront(entry.element)
        return
    }

    // Evict if full
    if len(shard.data) >= shard.maxSize {
        oldest := shard.lru.Back()
        if oldest != nil {
            entry := oldest.Value.(*Entry)
            delete(shard.data, entry.key)
            shard.lru.Remove(oldest)
        }
    }

    // Add new entry
    entry := &Entry{
        key:     key,
        value:   value,
        expires: time.Now().Add(c.ttl),
    }
    entry.element = shard.lru.PushFront(entry)
    shard.data[key] = entry
}

func (c *Cache) cleanup() {
    ticker := time.NewTicker(time.Minute)
    defer ticker.Stop()

    for range ticker.C {
        for _, shard := range c.shards {
            shard.mu.Lock()

            now := time.Now()
            for key, entry := range shard.data {
                if now.After(entry.expires) {
                    delete(shard.data, key)
                    shard.lru.Remove(entry.element)
                }
            }

            shard.mu.Unlock()
        }
    }
}
```

Whiteboard Checklist

- [ ] Clarify requirements (throughput, latency, consistency)
- [ ] Choose concurrency model (workers, pipeline, actors)
- [ ] Draw architecture diagram (boxes + arrows)

- [ ] Identify critical sections (mutexes)
- [ ] Discuss failure modes (timeouts, circuit breakers)
- [ ] Explain scalability (horizontal/vertical)
- [ ] Consider edge cases (empty input, timeout, cancellation)
- [ ] Mention testing strategy (race detector, stress tests)

Key Takeaways

1. **Always clarify requirements first**
2. **Choose workers based on I/O vs CPU**
3. **Shard to reduce contention**
4. **Add timeouts to all I/O**
5. **Use circuit breakers for dependencies**
6. **Pipeline for mixed I/O + CPU**
7. **Context for cancellation**
8. **Graceful degradation over failure**

Next: how-to-explain-concurrency.md - Teaching techniques and communication.