Concurrency Boundaries

What are Concurrency Boundaries?

Concurrency boundary: Point where sequential code transitions to concurrent execution.

Key question: "Where should I spawn goroutines?"

Bad answer: "Everywhere!" (leads to chaos)
Good answer: "At natural system boundaries" (maintains clarity)

Identifying Natural Boundaries

Boundary 1: Network I/O

Example: HTTP Server

```
// Natural boundary: One goroutine per request
func (s *Server) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // This already runs in a goroutine (net/http does it)

    // Process request sequentially
    user := authenticate(r)
    data := fetchData(user)
    response := render(data)
    w.Write(response)
}
```

Why natural:

- Each request independent
- I/O-bound (waiting for network)
- Clear lifecycle (request → response)

Boundary 2: Independent Tasks

Example: Batch Processing

```
func processBatch(items []Item) []Result {
    // Boundary: Each item independent
    resultChan := make(chan Result, len(items))

    for _, item := range items {
        item := item
        go func() {
            resultChan <- process(item)  // Concurrent
        }()
    }

    // Collect results
    results := make([]Result, 0, len(items))
    for i := 0; i < len(items); i++ {
```

```
        results = append(results, <-resultChan)
    }

    return results
}
```

Why natural:

- No dependencies between items
- CPU-bound work benefits from parallelism
- Each task self-contained

Boundary 3: Long-Running Services

Example: Background Workers

```
type Application struct {
    httpServer  *http.Server
    taskQueue   *TaskQueue
    cacheSync   *CacheSync
    healthCheck *HealthCheck
}

func (a *Application) Start() {
    // Boundary: Each service independent
    go a.httpServer.ListenAndServe()
    go a.taskQueue.Run()
    go a.cacheSync.Run()
    go a.healthCheck.Run()
}
```

Why natural:

- Each service has its own lifecycle
- Run indefinitely
- Independent responsibilities

Boundary 4: Fan-Out Operations

Example: Multi-API Aggregation

```
func getProfile(userID string) (*Profile, error) {
    // Boundary: Fetch from multiple sources concurrently

    var userInfo UserInfo
    var posts []Post
    var friends []User
    var errInfo, errPosts, errFriends error

    var wg sync.WaitGroup
    wg.Add(3)

    go func() {
```

```
        defer wg.Done()
        userInfo, errInfo = fetchUserInfo(userID)
    }()

    go func() {
        defer wg.Done()
        posts, errPosts = fetchPosts(userID)
    }()

    go func() {
        defer wg.Done()
        friends, errFriends = fetchFriends(userID)
    }()

    wg.Wait()

    if errInfo != nil {
        return nil, errInfo
    }

    return &Profile{
        User:    userInfo,
        Posts:   posts,
        Friends: friends,
    }, nil
}
```

Why natural:

- Independent API calls
- I/O-bound (network requests)
- Results combined at end

Where NOT to Add Concurrency

Anti-Pattern 1: Fine-Grained Concurrency

```
// BAD: Goroutine for every tiny operation
func calculateTotal(items []int) int {
    results := make(chan int, len(items))

    for _, item := range items {
        go func(item int) {
            results <- item * 2  // Goroutine overhead > work!
        }(item)
    }

    total := 0
    for i := 0; i < len(items); i++ {
        total += <-results
    }
```

```
    return total
}

// GOOD: Sequential for small work
func calculateTotal(items []int) int {
    total := 0
    for _, item := range items {
        total += item * 2
    }
    return total
}
```

Rule: Goroutine overhead (~2KB stack + scheduling) should be < work time.

Anti-Pattern 2: Hidden Concurrency in Libraries

```
// BAD: Library spawns goroutines internally
type Cache struct{}

func (c *Cache) Get(key string) string {
    result := make(chan string)
    go func() {
        // Fetch from cache
        result <- fetch(key)
    }()
    return <-result  // Caller waits anyway!
}

// GOOD: Let caller decide concurrency
func (c *Cache) Get(key string) string {
    return fetch(key)  // Sequential
}
```

Rule: Don't hide goroutines in synchronous-looking APIs.

Anti-Pattern 3: Premature Optimization

```
// BAD: Adding concurrency "just in case"
func readConfig() Config {
    configChan := make(chan Config)
    errChan := make(chan error)

    go func() {
        // Read config (happens once at startup)
        data, err := ioutil.ReadFile("config.json")
        if err != nil {
            errChan <- err
            return
        }
        var cfg Config
```

```
        json.Unmarshal(data, &cfg)
        configChan <- cfg
    }()

    select {
    case cfg := <-configChan:
        return cfg
    case err := <-errChan:
        log.Fatal(err)
    }
}

// GOOD: Sequential is simpler
func readConfig() Config {
    data, err := ioutil.ReadFile("config.json")
    if err != nil {
        log.Fatal(err)
    }
    var cfg Config
    json.Unmarshal(data, &cfg)
    return cfg
}
```

Rule: Add concurrency when proven necessary, not speculatively.

Boundary Design Patterns

Pattern 1: API Layer Boundary

Concurrency starts at API handlers.

```
// HTTP handler already concurrent (net/http)
func (h *Handler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // This runs in its own goroutine

    // Business logic sequential
    result := h.service.Process(r.Context(), extractData(r))

    writeJSON(w, result)
}
```

Pattern 2: Repository Boundary

Data access points where I/O happens.

```
type UserRepository struct {
    db *sql.DB
}

// Sequential interface
func (r *UserRepository) GetByID(ctx context.Context, id string) (*User, error) {
```

```
    // Let caller decide if concurrent
    return queryUser(ctx, r.db, id)
}

// Caller controls concurrency
func getMultipleUsers(repo *UserRepository, ids []string) []*User {
    results := make(chan *User, len(ids))

    for _, id := range ids {
        id := id
        go func() {
            user, _ := repo.GetByID(context.Background(), id)
            results <- user
        }()
    }

    users := make([]*User, 0, len(ids))
    for i := 0; i < len(ids); i++ {
        users = append(users, <-results)
    }
    return users
}
```

Pattern 3: Service Layer Boundary

Business logic orchestrates concurrent operations.

```
type OrderService struct {
    inventory  InventoryService
    payment    PaymentService
    shipping   ShippingService
    notifier   NotificationService
}

func (s *OrderService) PlaceOrder(ctx context.Context, order Order) error {
    // Sequential critical path
    if err := s.inventory.Reserve(ctx, order.Items); err != nil {
        return err
    }

    if err := s.payment.Charge(ctx, order.Payment); err != nil {
        s.inventory.Release(ctx, order.Items)
        return err
    }

    // Concurrent non-critical operations
    var wg sync.WaitGroup

    wg.Add(2)
    go func() {
        defer wg.Done()
```

```
        s.shipping.Schedule(ctx, order)
    }()

    go func() {
        defer wg.Done()
        s.notifier.SendConfirmation(ctx, order.UserID)
    }()

    wg.Wait()

    return nil
}
```

Pattern 4: Worker Pool Boundary

Controlled concurrency for CPU-bound work.

```
type ImageProcessor struct {
    workers int
    tasks   chan Task
}

func NewImageProcessor(workers int) *ImageProcessor {
    p := &ImageProcessor{
        workers: workers,
        tasks:   make(chan Task, workers*2),
    }

    // Boundary: Fixed number of workers
    for i := 0; i < workers; i++ {
        go p.worker()
    }

    return p
}

func (p *ImageProcessor) Process(img Image) Result {
    // Caller submits, worker pool handles concurrency
    result := make(chan Result)
    p.tasks <- Task{Image: img, Result: result}
    return <-result
}

func (p *ImageProcessor) worker() {
    for task := range p.tasks {
        task.Result <- processImage(task.Image)
    }
}
```

Minimizing Shared State

Technique 1: Immutable Configuration

```
// Shared read-only config (no locks needed)
type Config struct {
    Timeout  time.Duration
    MaxRetry int
}

var cfg atomic.Value  // Holds *Config

func init() {
    cfg.Store(&Config{Timeout: time.Second, MaxRetry: 3})
}

func getConfig() *Config {
    return cfg.Load().(*Config)
}

// Safe to use across goroutines
func handler() {
    config := getConfig()
    // Use config.Timeout, config.MaxRetry
}
```

Technique 2: Message Passing

```
// Each goroutine owns its state
type Account struct {
    balance int
    ops     chan Operation
}

type Operation struct {
    Type   OpType
    Amount int
    Result chan int
}

func (a *Account) Run() {
    for op := range a.ops {
        switch op.Type {
        case Deposit:
            a.balance += op.Amount
            op.Result <- a.balance
        case Withdraw:
            a.balance -= op.Amount
            op.Result <- a.balance
        }
    }
}
```

```
// No shared state, no locks
func (a *Account) Deposit(amount int) int {
    result := make(chan int)
    a.ops <- Operation{Type: Deposit, Amount: amount, Result: result}
    return <-result
}
```

Technique 3: Partitioning

```
// Shard data to reduce contention
type ShardedCache struct {
    shards []*CacheShard
}

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

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

func (sc *ShardedCache) Get(key string) string {
    shard := sc.getShard(key)
    shard.mu.RLock()
    defer shard.mu.RUnlock()
    return shard.data[key]
}

// Each shard independent, reduces lock contention
```

Real Example: Web Crawler

```
type Crawler struct {
    // Boundary 1: Fetcher goroutines
    fetchWorkers int

    // Boundary 2: Parser goroutines
    parseWorkers int

    // Internal state (not shared)
    urlQueue  chan string
    htmlQueue chan HTML
    results   chan Result

    // Shared state (protected)
```

```
    visited sync.Map
}

func NewCrawler(fetchWorkers, parseWorkers int) *Crawler {
    c := &Crawler{
        fetchWorkers: fetchWorkers,
        parseWorkers: parseWorkers,
        urlQueue:     make(chan string, 100),
        htmlQueue:    make(chan HTML, 100),
        results:      make(chan Result, 100),
    }

    // Boundary: Start workers
    for i := 0; i < fetchWorkers; i++ {
        go c.fetcher()
    }

    for i := 0; i < parseWorkers; i++ {
        go c.parser()
    }

    return c
}

func (c *Crawler) fetcher() {
    for url := range c.urlQueue {
        // Check if visited (shared state, but sync.Map handles concurrency)
        if _, seen := c.visited.LoadOrStore(url, true); seen {
            continue
        }

        // Fetch (I/O-bound, benefits from concurrency)
        html := fetch(url)
        c.htmlQueue <- html
    }
}

func (c *Crawler) parser() {
    for html := range c.htmlQueue {
        // Parse (CPU-bound, benefits from parallelism)
        result := parse(html)
        c.results <- result

        // Extract links
        for _, link := range result.Links {
            c.urlQueue <- link
        }
    }
}

func (c *Crawler) Crawl(startURL string) <-chan Result {
    c.urlQueue <- startURL
```

```
    return c.results
}
```

Boundaries:

1. **API boundary:** `Crawl()` spawns system
2. **I/O boundary:** Fetcher goroutines (network)
3. **CPU boundary:** Parser goroutines (processing)

State management:

- `urlQueue`, `htmlQueue`, `results`: Private channels (no locks)
- `visited`: Shared `sync.Map` (internally synchronized)

Testing Concurrency Boundaries

```
func TestCrawler(t *testing.T) {
    // Mock HTTP responses
    server := httptest.NewServer(http.HandlerFunc(func(w http.ResponseWriter, r
*http.Request) {
        w.Write([]byte("<a href='/page2'>Link</a>"))
    }))
    defer server.Close()

    crawler := NewCrawler(2, 2)
    results := crawler.Crawl(server.URL)

    // Collect results
    timeout := time.After(time.Second)
    found := 0

    for {
        select {
        case <-results:
            found++
        case <-timeout:
            if found < 2 {
                t.Errorf("Expected at least 2 results, got %d", found)
            }
            return
        }
    }
}
```

Boundary Design Checklist

- [ ] Concurrency at natural I/O or CPU boundaries
- [ ] Each boundary has clear ownership
- [ ] Shared state minimized or eliminated
- [ ] Goroutine count controlled (not unbounded)

- [ ] Each goroutine has exit condition
- [ ] Boundaries documented in code
- [ ] Testing covers concurrent execution
- [ ] Errors propagated across boundaries
- [ ] Graceful shutdown at each boundary
- [ ] Monitoring/observability at boundaries

Interview Questions

Q: "Where should concurrency start in a web service?"

"Primary boundary: HTTP handler (net/http already spawns goroutine per request). Secondary boundaries: 1) Fan-out for parallel API calls (fetch user, posts, friends concurrently). 2) Worker pools for CPU-intensive tasks (image processing, data transformation). 3) Background services (cache sync, health checks). Don't add concurrency in: Business logic (keep sequential), data access layer (caller controls), utility functions. Rule: Concurrency at I/O and CPU boundaries, not everywhere."

Q: "How do you minimize shared state?"

"1) Ownership: Each goroutine owns its data. 2) Message passing: Communicate via channels, not shared memory. 3) Immutable data: Read-only config safe to share. 4) Atomic operations: Simple counters/flags. 5) Partitioning: Shard data to reduce contention (16 shards = 1/16 lock contention). Example: Instead of shared cache with mutex, use actor pattern where cache goroutine owns state, others send requests via channel."

Q: "When is concurrency premature?"

"When: 1) Operation runs once (startup config loading). 2) Work smaller than goroutine overhead (~1µs). 3) No I/O or CPU parallelism (sequential dependencies). 4) Not measured as bottleneck. Example: Don't spawn goroutine to read 10-line config file. Do spawn for: HTTP endpoints, parallel API calls, batch processing, long-running services. Measure first, optimize second."

Q: "How do you control goroutine proliferation?"

"1) Worker pools: Fixed number of goroutines, tasks queued. 2) Semaphores: Limit concurrent operations (buffered channel of size N). 3) Rate limiters: Control creation rate (golang.org/x/time/rate). 4) Context: Cancel goroutines when work unnecessary. 5) WaitGroups: Track and wait for completion. Example: Instead of `go fetch(url)` for 10000 URLs (10000 goroutines), use worker pool with runtime.NumCPU()*10 workers."

Key Takeaways

1. **Place boundaries at natural I/O or CPU points**
2. **Don't hide goroutines in synchronous APIs**
3. **Minimize shared state across boundaries**
4. **Control goroutine count (worker pools, semaphores)**
5. **Each boundary needs clear ownership**
6. **Caller should control concurrency, not library**
7. **Add concurrency for I/O latency or CPU parallelism**
8. **Sequential is simpler, use when work is fast**
9. **Document boundaries in code**
10. **Test with realistic concurrency levels**

Exercises

1. Design boundaries for e-commerce checkout flow.
2. Identify unnecessary concurrency in codebas and remove it.
3. Convert unbounded goroutine creation to worker pool.
4. Build system with 3 clear concurrency boundaries, document each.
5. Refactor shared state to message-passing architecture.

Next: choosing-primitives.md - Decision tree for selecting concurrency primitives.