Failure Modes

Design for Failure

Murphy's Law: "Anything that can go wrong, will go wrong."

Principle: Failures are inevitable. Design systems to handle them gracefully.

Common Failure Modes

1. Dependency Failure

Scenario: External service (database, API, cache) unavailable.

Impact:

- Cascading failures
- Thread pool exhaustion
- System-wide outage

Example:

```go
// BAD: Blocks forever if DB down
func getUser(id string) (*User, error) {
    return db.Query(id)  // Hangs indefinitely
}

// GOOD: Timeout protection
func getUser(ctx context.Context, id string) (*User, error) {
    ctx, cancel := context.WithTimeout(ctx, 2*time.Second)
    defer cancel()

    result := make(chan *User, 1)
    errCh := make(chan error, 1)

    go func() {
        user, err := db.Query(id)
        if err != nil {
            errCh <- err
            return
        }
        result <- user
    }()

    select {
    case user := <-result:
        return user, nil
    case err := <-errCh:
        return nil, err
    case <-ctx.Done():
        return nil, ctx.Err()
    }
}
```

2. Slow Dependency

Scenario: Service responds, but very slowly.

Impact:

- Resource exhaustion (goroutines, connections)
- Degraded user experience
- Eventual failure

Solution: 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  // Normal operation
    StateOpen                  // Rejecting requests
    StateHalfOpen            // Testing recovery
)

func (cb *CircuitBreaker) Call(fn func() error) error {
    cb.mu.Lock()
    state := cb.state

    switch state {
    case StateOpen:
        // Check if should try again
        if time.Since(cb.lastAttempt) > cb.timeout {
            cb.state = StateHalfOpen
            cb.mu.Unlock()
            return cb.attempt(fn)
        }
        cb.mu.Unlock()
        return ErrCircuitOpen

    case StateHalfOpen, StateClosed:
        cb.mu.Unlock()
        return cb.attempt(fn)
    }
    return nil
}

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 - reset
    cb.failures = 0
    cb.state = StateClosed
    return nil
}

// Usage
var dbCircuit = &CircuitBreaker{maxFailures: 5, timeout: 10 * time.Second}

func queryWithCircuit(id string) (*User, error) {
    var user *User
    err := dbCircuit.Call(func() error {
        var err error
        user, err = db.Query(id)
        return err
    })
    return user, err
}
```

Benefits:

- Fail fast (don't wait for timeout)
- Prevent resource exhaustion
- Automatic recovery

3. Resource Exhaustion

Scenario: System runs out of resources (goroutines, connections, memory).

Solution: Bulkhead Pattern

Isolate critical components so failure in one doesn't affect others.

```go
type Bulkhead struct {
    critical    *WorkerPool
    nonCritical *WorkerPool
}

func NewBulkhead() *Bulkhead {
    return &Bulkhead{
        // Critical operations get guaranteed resources
        critical: NewWorkerPool(runtime.NumCPU() * 2),

        // Non-critical get less
        nonCritical: NewWorkerPool(runtime.NumCPU()),
    }
}

func (b *Bulkhead) ProcessCritical(task Task) Result {
    return b.critical.Submit(task)  // Always available
}

func (b *Bulkhead) ProcessNonCritical(task Task) Result {
    select {
    case result := <-b.nonCritical.TrySubmit(task):
        return result
    default:
        return Result{Error: ErrRejected}  // Shed load
    }
}
```

Examples:

- Separate connection pools (user traffic vs analytics)
- Dedicated goroutines (critical vs background)
- Resource quotas (CPU, memory limits)

4. Cascading Failure

Scenario: One failure triggers others.

Example:

1. Database slows down
2. API handlers wait on DB
3. Goroutines accumulate
4. HTTP server exhausts connections
5. Load balancer marks server unhealthy
6. Traffic shifts to other servers
7. They fail too (domino effect)

Prevention:

Timeout at Every Layer

```go
// HTTP client
client := &http.Client{
    Timeout: 5 * time.Second,
}

// Database
ctx, cancel := context.WithTimeout(ctx, 2*time.Second)
defer cancel()
db.QueryContext(ctx, query)

// Cache
ctx, cancel := context.WithTimeout(ctx, 100*time.Millisecond)
defer cancel()
cache.Get(ctx, key)
```

Backpressure

```go
type Server struct {
    sem chan struct{}  // Limit concurrent requests
}

func (s *Server) HandleRequest(w http.ResponseWriter, r *http.Request) {
    // Reject if overloaded
    select {
    case s.sem <- struct{}{}:
        defer func() { <-s.sem }()
    default:
        http.Error(w, "Server overloaded", http.StatusServiceUnavailable)
        return
    }

    // Process request
    s.process(r)
}
```

Load Shedding

```go
func (s *Server) HandleRequest(w http.ResponseWriter, r *http.Request) {
    // Check system load
    if runtime.NumGoroutine() > 10000 {
        // Shed non-critical requests
        if !isCritical(r) {
            http.Error(w, "Try again later", http.StatusServiceUnavailable)
            return
        }
    }

    s.process(r)
}
```

5. Deadlock

Scenario: Goroutines wait for each other indefinitely.

Prevention:

Lock Ordering

```go
// BAD: Inconsistent order
func transfer(from, to *Account, amount int) {
    from.mu.Lock()
    to.mu.Lock()  // Potential deadlock
    // ...
}

// GOOD: Consistent order
func transfer(from, to *Account, amount int) {
    first, second := from, to
    if from.id > to.id {
        first, second = to, from  // Lower ID always first
    }

    first.mu.Lock()
    defer first.mu.Unlock()
    second.mu.Lock()
    defer second.mu.Unlock()

    from.balance -= amount
    to.balance += amount
}
```

Deadlock Detection

```go
// Detect with timeout
done := make(chan bool)
go func() {
    riskyOperation()
    done <- true
}()

select {
case <-done:
    // Success
case <-time.After(5 * time.Second):
    log.Error("Potential deadlock detected")
    dumpStackTrace()
}
```

6. Goroutine Leak

Scenario: Goroutines never exit, accumulate over time.

Detection:

```
func monitorGoroutines() {
    ticker := time.NewTicker(time.Minute)
    defer ticker.Stop()

    baseline := runtime.NumGoroutine()

    for range ticker.C {
        current := runtime.NumGoroutine()
        if current > baseline*2 {
            log.Errorf("Goroutine leak detected: %d (baseline: %d)", current,
baseline)
        }
    }
}
```

Prevention:

```
// GOOD: Always provide exit
func worker(ctx context.Context, tasks <-chan Task) {
    for {
        select {
        case task := <-tasks:
            process(task)
        case <-ctx.Done():
            return  // Exit when context cancelled
        }
    }
}
```

Graceful Degradation

Pattern 1: Fallback to Cache

```
func getUser(ctx context.Context, id string) (*User, error) {
    // Try primary source
    user, err := db.Query(ctx, id)
    if err == nil {
        return user, nil
    }

    // Fallback to cache (may be stale)
    log.Warnf("Database error, using cache: %v", err)
    return cache.Get(id)
}
```

Pattern 2: Degrade Features

```go
func getProfile(ctx context.Context, userID string) (*Profile, error) {
    profile := &Profile{}

    // Critical: User info (required)
    user, err := userService.Get(ctx, userID)
    if err != nil {
        return nil, err  // Fail hard
    }
    profile.User = user

    // Non-critical: Posts (best-effort)
    posts, err := postService.Get(ctx, userID)
    if err != nil {
        log.Warnf("Failed to fetch posts: %v", err)
        profile.Posts = nil  // Continue without
    } else {
        profile.Posts = posts
    }

    // Non-critical: Friends (best-effort)
    friends, err := friendService.Get(ctx, userID)
    if err != nil {
        log.Warnf("Failed to fetch friends: %v", err)
        profile.Friends = nil
    } else {
        profile.Friends = friends
    }

    return profile, nil
}
```

Pattern 3: Serve Stale Data

```go
type CachedResult struct {
    data      interface{}
    timestamp time.Time
}

func getWithStale(ctx context.Context, key string, maxStale time.Duration)
(interface{}, error) {
    // Try fresh data
    data, err := fetchFresh(ctx, key)
    if err == nil {
        cache.Set(key, CachedResult{data: data, timestamp: time.Now()})
        return data, nil
    }

    // Fetch failed, check cache
    cached, ok := cache.Get(key)
    if !ok {
        return nil, err  // No cached data
    }

    // Return stale data if recent enough
    age := time.Since(cached.timestamp)
    if age < maxStale {
        log.Warnf("Serving stale data (age: %v): %v", age, err)
        return cached.data, nil
    }

    return nil, err  // Too stale
}
```

Retry Strategies

Exponential Backoff

```go
func retryWithBackoff(ctx context.Context, fn func() error, maxRetries int) error {
    var err error
    backoff := 100 * time.Millisecond

    for i := 0; i < maxRetries; i++ {
        err = fn()
        if err == nil {
            return nil
        }

        // Don't retry on certain errors
        if isNonRetryable(err) {
            return err
        }

        log.Warnf("Retry %d/%d after %v: %v", i+1, maxRetries, backoff, err)

        select {
        case <-time.After(backoff):
            backoff *= 2  // Double each time
            if backoff > 10*time.Second {
                backoff = 10 * time.Second  // Cap at 10s
            }
        case <-ctx.Done():
            return ctx.Err()
        }
    }

    return err
}
```

Jitter

Prevent thundering herd (all clients retry at same time).

```go
func backoffWithJitter(attempt int) time.Duration {
    base := time.Duration(attempt) * time.Second

    // Add random jitter (±25%)
    jitter := time.Duration(rand.Float64() * 0.5 - 0.25)
    return base + base*jitter
}
```

Health Checks

```go
type HealthCheck struct {
    db    *sql.DB
    cache *Cache
    api   *APIClient
}

func (hc *HealthCheck) Check(ctx context.Context) error {
    ctx, cancel := context.WithTimeout(ctx, 5*time.Second)
    defer cancel()

    // Check database
    if err := hc.db.PingContext(ctx); err != nil {
        return fmt.Errorf("database unhealthy: %w", err)
    }

    // Check cache
    if _, err := hc.cache.Get(ctx, "health_check_key"); err != nil {
        log.Warnf("Cache unhealthy: %v", err)
        // Non-critical, don't fail
    }

    // Check API
    if err := hc.api.Ping(ctx); err != nil {
        return fmt.Errorf("API unhealthy: %w", err)
    }

    return nil
}

// HTTP endpoint
func (s *Server) healthHandler(w http.ResponseWriter, r *http.Request) {
    ctx, cancel := context.WithTimeout(r.Context(), 5*time.Second)
    defer cancel()

    if err := s.health.Check(ctx); err != nil {
        log.Errorf("Health check failed: %v", err)
        http.Error(w, err.Error(), http.StatusServiceUnavailable)
        return
    }

    w.WriteHeader(http.StatusOK)
    w.Write([]byte("OK"))
}
```

Real Example: Resilient HTTP Client

```go
type ResilientClient struct {
    client          *http.Client
    circuitBreaker  *CircuitBreaker
    rateLimiter     *rate.Limiter
    maxRetries      int
}

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

    // Circuit breaker
    var resp *http.Response
    err := rc.circuitBreaker.Call(func() error {
        return rc.retryableGet(ctx, url, &resp)
    })

    return resp, err
}

func (rc *ResilientClient) retryableGet(ctx context.Context, url string, resp
**http.Response) error {
    backoff := 100 * time.Millisecond

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

        r, err := rc.client.Do(req)
        if err == nil {
            if r.StatusCode < 500 {
                *resp = r
                return nil  // Success or client error (don't retry)
            }
            r.Body.Close()
        }
        // Server error or network error - retry
        if attempt < rc.maxRetries-1 {
            jitter := time.Duration(rand.Int63n(int64(backoff) / 2))
            sleep := backoff + jitter

            select {
            case <-time.After(sleep):
                backoff *= 2
            case <-ctx.Done():
                return ctx.Err()
            }
        }
    }

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

Monitoring for Failures

```go
type Metrics struct {
    requestsTotal     atomic.Int64
    requestsFailed    atomic.Int64
    circuitOpenCount  atomic.Int64
    retriesTotal      atomic.Int64
    goroutines        atomic.Int64
}

func (m *Metrics) Report() map[string]interface{} {
    total := m.requestsTotal.Load()
    failed := m.requestsFailed.Load()

    var errorRate float64
    if total > 0 {
        errorRate = float64(failed) / float64(total)
    }

    return map[string]interface{}{
        "requests_total":      total,
        "requests_failed":     failed,
        "error_rate":          errorRate,
        "circuit_open_count":  m.circuitOpenCount.Load(),
        "retries_total":       m.retriesTotal.Load(),
        "goroutines":          runtime.NumGoroutine(),
    }
}
```

Interview Questions

Q: "What is a circuit breaker and when to use it?"

"Circuit breaker prevents cascading failures by stopping calls to failing service. Three states: Closed (normal), Open (rejecting requests), Half-Open (testing recovery). Opens after N consecutive failures, stays open for timeout, then tries again. Use when: Calling external service, want to fail fast, prevent resource exhaustion. Example: If database timing out, circuit opens, immediately returns error instead of waiting for timeout each request."

Q: "How do you prevent cascading failures?"

"1) Timeouts: Every I/O operation has deadline. 2) Circuit breakers: Fail fast when dependency down. 3) Bulkheads: Isolate critical resources. 4) Backpressure: Reject requests when overloaded. 5) Load shedding: Drop non-critical work. 6) Retry with backoff: Don't overwhelm recovering service. Example: Database slows → timeout saves goroutines → circuit breaker fails fast → load shedding rejects non-critical → system stays up."

Q: "What is graceful degradation?"

"System continues with reduced functionality when components fail. Strategies: 1) Fallback to cache (stale data better than no data). 2) Degrade features (core works, extras fail). 3) Serve stale data (with staleness indicator). 4) Retry with fallback. Example: Profile page: If posts service down, still show user info and friends. Critical parts succeed, non-critical fail gracefully."

Q: "Explain bulkhead pattern."

"Isolate resources so failure in one area doesn't affect others (like ship compartments). Implementation: Separate connection pools, dedicated goroutines, resource quotas. Example: Separate pools for user traffic (100 connections) vs analytics (10 connections). If analytics queries slow, doesn't exhaust user traffic pool. Critical operations always have resources."

Key Takeaways

1. **Design for failure from the start**
2. **Timeout every I/O operation**
3. **Circuit breakers prevent cascading failures**
4. **Bulkheads isolate critical components**
5. **Retry with exponential backoff + jitter**
6. **Graceful degradation > complete failure**
7. **Monitor goroutine count (detect leaks)**
8. **Health checks for dependencies**
9. **Fail fast > wait forever**
10. **Test failure scenarios (chaos engineering)**

Exercises

1. Implement circuit breaker with Open/HalfOpen/Closed states.
2. Build HTTP client with retries, backoff, and circuit breaker.
3. Create bulkhead pattern with critical and non-critical pools.
4. Simulate cascading failure, then prevent with techniques learned.
5. Design system with graceful degradation for 3 failure scenarios.

Next: ../08-interview-prep/common-interview-questions.md - Top concurrency interview questions.