Avoiding Goroutine Leaks

What is a Goroutine Leak?

Goroutine leak: Goroutine that runs forever or blocks indefinitely, never exits.

Impact:

- **Memory leak:** Each goroutine ~2KB minimum
- **Resource leak:** Open files, connections
- **Accumulation:** Leaks compound over time
- **OOM crash:** Eventually exhausts memory

Detection:

```go
import "runtime"

func main() {
    for {
        log.Printf("Goroutines: %d", runtime.NumGoroutine())
        time.Sleep(time.Second)
    }
}

// Output:
// Goroutines: 5
// Goroutines: 10    ← Leak!
// Goroutines: 15    ← Growing
// Goroutines: 23
```

Common Leak Pattern #1: Blocking Channel Send

```go
// LEAK: Goroutine blocks forever if channel not read
func leak() {
    ch := make(chan int)

    go func() {
        ch <- 42  // BLOCKS FOREVER (no receiver)
    }()

    // Returns without reading from ch
}

// Fix 1: Buffered channel (if one value)
func fixed1() {
    ch := make(chan int, 1)  // Buffer size 1

    go func() {
        ch <- 42  // Doesn't block
    }()
}

// Fix 2: Select with default
func fixed2() {
    ch := make(chan int)

    go func() {
        select {
        case ch <- 42:
        default:
            // Nobody listening, give up
        }
    }()
}

// Fix 3: Context cancellation
func fixed3(ctx context.Context) {
    ch := make(chan int)

    go func() {
        select {
        case ch <- 42:
        case <-ctx.Done():
            return
        }
    }()
}
```

Common Leak Pattern #2: Blocking Channel Receive

```go
// LEAK: Goroutine blocks waiting for channel that never sends
func leak() {
    ch := make(chan int)

    go func() {
        val := <-ch  // BLOCKS FOREVER (no sender)
        process(val)
    }()

    // Returns without sending to ch
}

// Fix: Always close channel or use context
func fixed(ctx context.Context) {
    ch := make(chan int)

    go func() {
        select {
        case val := <-ch:
            process(val)
        case <-ctx.Done():
            return
        }
    }()
}
```

Common Leak Pattern #3: Orphaned Goroutine

```go
// LEAK: Goroutine continues after caller returns
func processRequest(w http.ResponseWriter, r *http.Request) {
    results := make(chan Result)

    go func() {
        time.Sleep(10 * time.Second)
        results <- computeResult()  // LEAK if handler returns early
    }()

    // Client disconnects after 1 second
    select {
    case result := <-results:
        json.NewEncoder(w).Encode(result)
    case <-time.After(time.Second):
        http.Error(w, "Timeout", 408)
        return  // LEAK: Goroutine still running!
    }
}

// Fix: Use request context
func fixed(w http.ResponseWriter, r *http.Request) {
    ctx := r.Context()
    results := make(chan Result, 1)

    go func() {
        select {
        case <-ctx.Done():
            return  // Exit if request cancelled
        case <-time.After(10 * time.Second):
        }

        select {
        case results <- computeResult():
        case <-ctx.Done():
            return
        }
    }()

    select {
    case result := <-results:
        json.NewEncoder(w).Encode(result)
    case <-ctx.Done():
        http.Error(w, "Request cancelled", 499)
    }
}
```

Common Leak Pattern #4: Infinite Loop Without Exit

```go
// LEAK: Loop never exits
func leak() {
    go func() {
        for {
            doWork()
            time.Sleep(time.Second)
            // No exit condition!
        }
    }()
}

// Fix: Add cancellation
func fixed(ctx context.Context) {
    go func() {
        ticker := time.NewTicker(time.Second)
        defer ticker.Stop()

        for {
            select {
            case <-ticker.C:
                doWork()
            case <-ctx.Done():
                return
            }
        }
    }()
}
```

Common Leak Pattern #5: WaitGroup Never Done

```go
// LEAK: Goroutine waits forever
func leak() {
    var wg sync.WaitGroup

    wg.Add(1)
    go func() {
        defer wg.Done()
        doWork()
    }()

    wg.Add(1)  // Added but never Done()
    wg.Wait()  // BLOCKS FOREVER
}

// Fix: Match Add with Done
func fixed() {
    var wg sync.WaitGroup

    wg.Add(2)  // Exactly 2

    go func() {
        defer wg.Done()
        doWork()
    }()

    go func() {
        defer wg.Done()
        doWork()
    }()

    wg.Wait()
}
```

Common Leak Pattern #6: Select Without Default or Timeout

```go
// LEAK: Blocks forever if no cases ready
func leak(ch1, ch2 chan int) {
    go func() {
        select {
        case val := <-ch1:
            process(val)
        case val := <-ch2:
            process(val)
        }
        // Blocks forever if both empty
    }()
}

// Fix: Add timeout or context
func fixed(ctx context.Context, ch1, ch2 chan int) {
    go func() {
        select {
        case val := <-ch1:
            process(val)
        case val := <-ch2:
            process(val)
        case <-ctx.Done():
            return
        }
    }()
}
```

Detecting Leaks: runtime.NumGoroutine()

```go
func TestNoLeak(t *testing.T) {
    before := runtime.NumGoroutine()

    // Run code
    runCode()

    // Give goroutines time to exit
    time.Sleep(100 * time.Millisecond)

    after := runtime.NumGoroutine()

    if after > before {
        t.Errorf("Goroutine leak: before=%d, after=%d", before, after)
    }
}
```

Detecting Leaks: goleak Library

```go
import "go.uber.org/goleak"

func TestMain(m *testing.M) {
    goleak.VerifyTestMain(m)
}

func TestFunction(t *testing.T) {
    defer goleak.VerifyNone(t)

    // Test code
    runFunction()
}
```

Leak Pattern: HTTP Client Doesn't Close Body

```go
// LEAK: Response body not closed
func leak() error {
    resp, err := http.Get("https://example.com")
    if err != nil {
        return err
    }

    // Process response
    data, _ := io.ReadAll(resp.Body)
    // LEAK: Body not closed, connection not returned to pool

    return nil
}

// Fix: Always defer Close
func fixed() error {
    resp, err := http.Get("https://example.com")
    if err != nil {
        return err
    }
    defer resp.Body.Close()  // Must close!

    data, _ := io.ReadAll(resp.Body)
    return nil
}
```

Leak Pattern: Ticker Not Stopped

```go
// LEAK: Ticker goroutine never stops
func leak() {
    ticker := time.NewTicker(time.Second)

    go func() {
        for range ticker.C {
            doWork()
        }
    }()

    // LEAK: ticker still running
}

// Fix: Stop ticker
func fixed(ctx context.Context) {
    ticker := time.NewTicker(time.Second)
    defer ticker.Stop()  // Important!

    for {
        select {
        case <-ticker.C:
            doWork()
        case <-ctx.Done():
            return
        }
    }
}
```

Real Example: Fixing HTTP Handler Leak

```go
// BEFORE: Leaks goroutines on timeout
func handlerWithLeak(w http.ResponseWriter, r *http.Request) {
    result := make(chan string)

    go func() {
        // Expensive operation
        data := queryDatabase()
        result <- data  // LEAK if handler times out
    }()

    select {
    case data := <-result:
        w.Write([]byte(data))
    case <-time.After(2 * time.Second):
        http.Error(w, "Timeout", 408)
        // Goroutine still writing to result channel!
    }
}

// AFTER: No leak
func handlerFixed(w http.ResponseWriter, r *http.Request) {
    ctx, cancel := context.WithTimeout(r.Context(), 2*time.Second)
    defer cancel()

    result := make(chan string, 1)  // Buffered!

    go func() {
        select {
        case <-ctx.Done():
            return  // Exit on cancellation
        default:
        }

        data := queryDatabaseCtx(ctx)

        select {
        case result <- data:
        case <-ctx.Done():
            return  // Exit if nobody listening
        }
    }()

    select {
    case data := <-result:
        w.Write([]byte(data))
    case <-ctx.Done():
        http.Error(w, "Timeout", 408)
    }
}
```

Goroutine Lifecycle Management

```go
type Worker struct {
    tasks  chan Task
    quit   chan struct{}
    wg     sync.WaitGroup
    cancel context.CancelFunc
}

func NewWorker() *Worker {
    ctx, cancel := context.WithCancel(context.Background())

    w := &Worker{
        tasks:  make(chan Task, 10),
        quit:   make(chan struct{}),
        cancel: cancel,
    }

    w.wg.Add(1)
    go w.run(ctx)

    return w
}

func (w *Worker) run(ctx context.Context) {
    defer w.wg.Done()

    for {
        select {
        case task := <-w.tasks:
            task.Execute()

        case <-ctx.Done():
            // Drain remaining tasks
            for {
                select {
                case task := <-w.tasks:
                    task.Execute()
                default:
                    return
                }
            }
        }
    }
}

func (w *Worker) Submit(task Task) error {
    select {
    case w.tasks <- task:
        return nil
    case <-w.quit:
        return errors.New("worker stopped")
    }
}

func (w *Worker) Stop() {
    close(w.quit)
    w.cancel()
    w.wg.Wait()
}
```

Preventing Leaks: Checklist

✅ **Every goroutine must have:**

1. **Exit path:** Way to exit (done channel, context)
2. **Ownership:** Someone responsible for stopping it
3. **Timeout:** Bounded execution time
4. **Error handling:** What if operation fails?

✅ **Channel operations:**

1. **Sends:** Use buffered channel or select with default/timeout
2. **Receives:** Always have done/context channel in select
3. **Close:** Owner closes channel, receivers check for close

✅ **Resources:**

1. **HTTP bodies:** Always defer resp.Body.Close()
2. **Files:** Always defer file.Close()
3. **Tickers:** Always defer ticker.Stop()
4. **Database rows:** Always defer rows.Close()

Debugging Leaks

pprof Goroutine Profile

```go
import _ "net/http/pprof"

func main() {
    // Enable pprof
    go func() {
        http.ListenAndServe(":6060", nil)
    }()

    // Rest of application
}

// View goroutines:
// http://localhost:6060/debug/pprof/goroutine
// curl http://localhost:6060/debug/pprof/goroutine?debug=2
```

Stack Trace Analysis

```
# Get goroutine dump
curl http://localhost:6060/debug/pprof/goroutine?debug=2 > goroutines.txt

# Look for:
# - Growing number of same goroutine
# - Goroutines blocked on channel operations
# - Goroutines waiting on select

# Example leak pattern:
# goroutine 123 [chan send]:
# main.sendData(...)
#     /app/main.go:45
# (repeated 1000+ times)
```

Interview Questions

Q: "What causes goroutine leaks?"

"Goroutine blocks forever and never exits. Common causes: 1) Blocking channel send/receive with no way out, 2) Infinite loop without exit condition, 3) Waiting on WaitGroup that never completes, 4) Select without timeout or context. Prevention: Every goroutine needs exit path (done channel, context, timeout)."

Q: "How do you detect goroutine leaks?"

"Monitor runtime.NumGoroutine() over time—increasing count indicates leak. Use goleak library in tests. Use pprof goroutine profile to see stack traces. Look for goroutines blocked on chan operations. Production: Alert on increasing goroutine count."

Q: "What's the pattern to prevent leaks in HTTP handlers?"

"Use request context. Pattern: 1) Get r.Context(), 2) Pass to goroutines, 3) Check ctx.Done() in select statements, 4) Use buffered channel if goroutine may write after handler returns. This ensures goroutines exit when client disconnects or times out."

Q: "Why must you close HTTP response bodies?"

"If body not read fully and closed, connection can't be returned to pool. Causes connection leak (exhausts connection pool), goroutine leak (goroutine managing connection), memory leak (buffers). Always defer resp.Body.Close() immediately after error check, even if not reading body."

Key Takeaways

1. **Every goroutine must have exit path**
2. **Use context for cancellation**
3. **Buffered channels prevent some send blocks**
4. **Select with context.Done() in loops**
5. **Stop tickers with ticker.Stop()**
6. **Close HTTP response bodies**
7. **Test with goleak**
8. **Monitor runtime.NumGoroutine()**

9. **Use pprof for debugging**
10. **Ownership: Someone must stop each goroutine**

Exercises

1. Find and fix the leak:

```go
func leak() {
    results := make(chan int)
    go compute(results)
    return  // Where's the leak?
}
```

2. Write test using goleak that detects goroutine leak.

3. Create worker pool with proper shutdown (no leaks).

4. Fix HTTP handler that leaks on client disconnect.

5. Use pprof to identify goroutine leak in running program.

Section 05 Complete! Next: ../06-testing-and-debugging/race-detector.md - Testing and debugging concurrent code.