Deadlocks

What is a Deadlock?

Deadlock: A situation where two or more goroutines are permanently blocked, waiting for each other to release resources.

Characteristics (Coffman Conditions):

1. **Mutual exclusion:** Resources cannot be shared (e.g., mutex)
2. **Hold and wait:** Goroutine holds resource while waiting for another
3. **No preemption:** Resources cannot be forcibly taken
4. **Circular wait:** Circular chain of goroutines, each waiting for next

All four must be true for deadlock to occur.

Example 1: Simple Circular Wait

```go
var mu1, mu2 sync.Mutex

// Goroutine 1
func transfer1() {
    mu1.Lock()  // Acquire mu1
    mu2.Lock()  // Wait for mu2
    // Transfer
    mu2.Unlock()
    mu1.Unlock()
}

// Goroutine 2
func transfer2() {
    mu2.Lock()  // Acquire mu2
    mu1.Lock()  // Wait for mu1
    // Transfer
    mu1.Unlock()
    mu2.Unlock()
}

// Deadlock scenario:
// G1 locks mu1, waits for mu2
// G2 locks mu2, waits for mu1
// Both wait forever
```

Solution 1: Lock ordering

```go
// Always acquire locks in same order
func transfer() {
    mu1.Lock()
    mu2.Lock()
    // Transfer
    mu2.Unlock()
    mu1.Unlock()
}
```

Solution 2: Try-lock (not available in Go's sync.Mutex)

Go doesn't support try-lock, but you can use channels with select:

```
type ChannelMutex struct {
    ch chan struct{}
}

func NewChannelMutex() *ChannelMutex {
    m := &ChannelMutex{ch: make(chan struct{}, 1)}
    m.ch <- struct{}{}
    return m
}

func (m *ChannelMutex) TryLock() bool {
    select {
    case <-m.ch:
        return true
    default:
        return false
    }
}

func (m *ChannelMutex) Unlock() {
    m.ch <- struct{}{}
}

// Usage:
func transfer() {
    if !mu1.TryLock() {
        return  // Failed to acquire, backoff
    }
    defer mu1.Unlock()

    if !mu2.TryLock() {
        return  // Failed to acquire, release mu1 and backoff
    }
    defer mu2.Unlock()

    // Perform transfer
}
```

Example 2: Channel Deadlock (Unbuffered)

```
ch := make(chan int)
```

```
// Single goroutine
ch <- 42  // Blocks forever (no receiver)
```

Why deadlock: Unbuffered channel send blocks until receiver ready. No other goroutine exists.

Go runtime detects:

```
fatal error: all goroutines are asleep - deadlock!
```

Solution 1: Buffered channel

```
ch := make(chan int, 1)
ch <- 42  // Doesn't block (buffer has space)
```

Solution 2: Separate goroutine

```
ch := make(chan int)
go func() {
    ch <- 42
}()
<-ch  // Receiver ready
```

Example 3: WaitGroup Deadlock

```
var wg sync.WaitGroup

// Forgot to call wg.Add()
go func() {
    wg.Done()  // Panics: negative counter
}()

wg.Wait()  // Returns immediately (counter=0)
```

Or:

```
var wg sync.WaitGroup

wg.Add(1)
go func() {
    // Forgot wg.Done()
}()

wg.Wait()  // Blocks forever
```

Solution: Always pair Add() with Done().

```
var wg sync.WaitGroup
```

```go
wg.Add(1)
go func() {
    defer wg.Done()  // Ensure it's called
    // Work
}()

wg.Wait()
```

Example 4: Mutex with Panic

```go
var mu sync.Mutex

func process() {
    mu.Lock()
    // Work that might panic
    doWork()  // Panics!
    mu.Unlock()  // Never called
}

// Other goroutines
mu.Lock()  // Deadlock: mu never released
```

Solution: Use defer.

```go
func process() {
    mu.Lock()
    defer mu.Unlock()

    doWork()  // If panics, mu still released
}
```

Example 5: Self-Deadlock (Recursive Lock)

```go
var mu sync.Mutex

func outer() {
    mu.Lock()
    defer mu.Unlock()
    inner()
}

func inner() {
    mu.Lock()  // Deadlock: same goroutine tries to lock again
    defer mu.Unlock()
    // Work
}
```

Go's sync.Mutex is not recursive (cannot be locked twice by same goroutine).

Solution 1: Don't call inner from outer (redesign)

Solution 2: Pass flag indicating lock held

```
func outer() {
    mu.Lock()
    defer mu.Unlock()
    innerLocked()
}

func inner() {
    mu.Lock()
    defer mu.Unlock()
    innerLocked()
}

func innerLocked() {
    // Assumes lock held
}
```

Solution 3: Use RWMutex if only reading (allows nested RLocks)

```
var mu sync.RWMutex

func outer() {
    mu.RLock()
    defer mu.RUnlock()
    inner()
}

func inner() {
    mu.RLock()  // OK: Multiple RLocks allowed
    defer mu.RUnlock()
    // Read-only work
}
```

Example 6: Dependency Cycle

```
// Service A depends on Service B
type ServiceA struct {
    B *ServiceB
}

// Service B depends on Service A
type ServiceB struct {
    A *ServiceA
}

// Initialization deadlock
a := &ServiceA{}
```

```
b := &ServiceB{}

a.B = b
b.A = a

// If both try to initialize each other in constructors → deadlock
```

Solution: Break circular dependency with interface or channel.

```
type ServiceA struct {
    B BInterface  // Interface, not concrete type
}

type BInterface interface {
    DoWork()
}

type ServiceB struct {
    // No dependency on A
}
```

Example 7: Dining Philosophers (Classic)

```
const numPhilosophers = 5

var forks [numPhilosophers]sync.Mutex

func philosopher(id int) {
    left := id
    right := (id + 1) % numPhilosophers

    for {
        // Think
        time.Sleep(time.Millisecond)

        // Pick up forks
        forks[left].Lock()   // Pick up left fork
        forks[right].Lock()  // Pick up right fork

        // Eat
        fmt.Printf("Philosopher %d eating\n", id)

        // Put down forks
        forks[right].Unlock()
        forks[left].Unlock()
    }
}

// Deadlock: All philosophers pick up left fork, then wait for right fork
```

Solution 1: Asymmetric lock order

```go
func philosopher(id int) {
    left := id
    right := (id + 1) % numPhilosophers

    // Order locks consistently
    first, second := left, right
    if first > second {
        first, second = second, first
    }

    for {
        time.Sleep(time.Millisecond)

        forks[first].Lock()
        forks[second].Lock()

        fmt.Printf("Philosopher %d eating\n", id)

        forks[second].Unlock()
        forks[first].Unlock()
    }
}
```

Solution 2: Limit number of diners (resource hierarchy)

```go
semaphore := make(chan struct{}, numPhilosophers-1)

func philosopher(id int) {
    for {
        time.Sleep(time.Millisecond)

        semaphore <- struct{}{}  // At most N-1 can try to eat

        forks[left].Lock()
        forks[right].Lock()

        fmt.Printf("Philosopher %d eating\n", id)

        forks[right].Unlock()
        forks[left].Unlock()

        <-semaphore
    }
}
```

Real-World Failure: MySQL Deadlock (2019, E-Commerce)

Date: November 2019
Incident: Transaction deadlocks caused order processing failures

Simplified scenario:

```
// Transaction 1: Update order, then inventory
func processOrder(orderID, productID int) error {
    tx, _ := db.Begin()
    defer tx.Rollback()

    // Lock order row
    tx.Exec("UPDATE orders SET status=? WHERE id=?", "processing", orderID)

    // Lock inventory row
    tx.Exec("UPDATE inventory SET quantity=quantity-1 WHERE product_id=?",
productID)

    return tx.Commit()
}

// Transaction 2: Update inventory, then order
func refundOrder(orderID, productID int) error {
    tx, _ := db.Begin()
    defer tx.Rollback()

    // Lock inventory row
    tx.Exec("UPDATE inventory SET quantity=quantity+1 WHERE product_id=?",
productID)

    // Lock order row
    tx.Exec("UPDATE orders SET status=? WHERE id=?", "refunded", orderID)

    return tx.Commit()
}

// Deadlock:
// T1 locks order 123, waits for inventory 456
// T2 locks inventory 456, waits for order 123
```

Fix: Consistent lock ordering.

```
// Always update inventory before orders
func processOrder(orderID, productID int) error {
    tx, _ := db.Begin()
    defer tx.Rollback()

    tx.Exec("UPDATE inventory SET quantity=quantity-1 WHERE product_id=?",
productID)
    tx.Exec("UPDATE orders SET status=? WHERE id=?", "processing", orderID)

    return tx.Commit()
```

```go
}

func refundOrder(orderID, productID int) error {
    tx, _ := db.Begin()
    defer tx.Rollback()

    tx.Exec("UPDATE inventory SET quantity=quantity+1 WHERE product_id=?",
productID)
    tx.Exec("UPDATE orders SET status=? WHERE id=?", "refunded", orderID)

    return tx.Commit()
}
```

Real-World Failure: Kubernetes Deadlock (2018)

Date: 2018
Component: Controller manager

Simplified version: Multiple controllers with shared cache, cyclic dependencies.

```go
type Controller struct {
    cache *Cache
    mu    sync.Mutex
}

var (
    controllerA *Controller
    controllerB *Controller
)

// Controller A handler
func (c *Controller) handleEventA() {
    c.mu.Lock()
    defer c.mu.Unlock()

    // Needs data from Controller B
    controllerB.mu.Lock()  // Deadlock risk
    data := controllerB.getData()
    controllerB.mu.Unlock()

    // Process
}

// Controller B handler
func (c *Controller) handleEventB() {
    c.mu.Lock()
    defer c.mu.Unlock()

    // Needs data from Controller A
    controllerA.mu.Lock()  // Deadlock risk
    data := controllerA.getData()
    controllerA.mu.Unlock()

    // Process
}
```

Fix: Use channels for inter-controller communication (no lock nesting).

```go
type Controller struct {
    requestCh chan Request
    responseCh chan Response
}

func (c *Controller) handleEvent() {
    for req := range c.requestCh {
        // Process request
        c.responseCh <- response
    }
}

// Request data from another controller (non-blocking)
func (c *Controller) requestData(target *Controller) Response {
    target.requestCh <- Request{}
    return <-target.responseCh
}
```

Detecting Deadlocks

1. Runtime Detection

Go runtime detects global deadlock:

```
fatal error: all goroutines are asleep - deadlock!
```

Limitation: Only detects when ALL goroutines are blocked. Partial deadlock (some goroutines blocked, others running) not detected.

2. Timeout Pattern

```go
func processWithTimeout(timeout time.Duration) error {
    done := make(chan bool)

    go func() {
        mu.Lock()
        // Work
        mu.Unlock()
        done <- true
    }()

    select {
    case <-done:
        return nil
    case <-time.After(timeout):
        return fmt.Errorf("timeout: possible deadlock")
    }
}
```

3. Lock Contention Profiling

```
go test -mutexprofile=mutex.out
go tool pprof mutex.out
```

Identifies hot locks (high contention → deadlock risk).

4. Static Analysis

go vet doesn't detect deadlocks directly, but can find:

- Copying locks (mutex by value)
- Funcs that don't return

Third-party tools:

- `go-deadlock` (drop-in replacement for sync.Mutex with deadlock detection)

```
import "github.com/sasha-s/go-deadlock"

var mu deadlock.Mutex  // Detects deadlock at runtime
```

Deadlock Prevention Strategies

Strategy 1: Lock Ordering

Rule: Always acquire locks in same global order.

```
// Define order: mu1 < mu2 < mu3
func transfer() {
    mu1.Lock()
    mu2.Lock()
    mu3.Lock()
    // Work
    mu3.Unlock()
    mu2.Unlock()
    mu1.Unlock()
}
```

Challenge: Hard to enforce in large codebase.

Strategy 2: Timeout

```
func lockWithTimeout(mu *sync.Mutex, timeout time.Duration) bool {
    locked := make(chan struct{})

    go func() {
        mu.Lock()
        close(locked)
    }()

    select {
    case <-locked:
        return true
    case <-time.After(timeout):
        return false  // Can't force unlock → goroutine leaks!
    }
}
```

Problem: Timeout doesn't release lock. This approach leaks goroutines in Go.

Better: Use context for cancellation, not locks.

Strategy 3: Avoid Lock Nesting

Rule: Never hold one lock while acquiring another.

```
// Wrong: Nested locks
func transfer() {
    mu1.Lock()
    mu2.Lock()  // Nesting
    // Work
    mu2.Unlock()
    mu1.Unlock()
}

// Right: Flat locks (acquire both at start)
func transfer() {
    lockBoth(&mu1, &mu2)  // Atomic helper
    defer unlockBoth(&mu1, &mu2)
    // Work
}
```

Strategy 4: Use Higher-Level Abstractions

Channels instead of locks:

```
type Account struct {
    balance int
    ops     chan operation
}

func (a *Account) run() {
    for op := range a.ops {
```

```go
        op(a)  // Serialized access
    }
}

func (a *Account) withdraw(amount int, done chan bool) {
    a.ops <- func(a *Account) {
        if a.balance >= amount {
            a.balance -= amount
        }
        done <- true
    }
}
```

Strategy 5: Deadlock Detection at Runtime

```go
// Use go-deadlock library
import "github.com/sasha-s/go-deadlock"

var mu1 deadlock.Mutex
var mu2 deadlock.Mutex

func transfer() {
    mu1.Lock()
    mu2.Lock()  // If timeout, panics with stack trace
    // Work
    mu2.Unlock()
    mu1.Unlock()
}
```

Interview Traps

Trap 1: "Just use defer, problem solved"

Incomplete: "defer mu.Unlock() prevents deadlock."
Correct: "defer ensures unlock on panic, but doesn't prevent circular wait deadlock. Must also use lock ordering or avoid nested locks."

Trap 2: "Buffered channels can't deadlock"

Wrong. Buffered channels deadlock when buffer is full and no receiver.

Correct: "Buffered channels reduce contention but can still deadlock if buffer fills up with no receiver, or if sender/receiver logic has bugs."

Trap 3: "RWMutex allows multiple readers, so no deadlock"

Wrong. RWMutex can deadlock if Lock() called while RLocks held.

Correct: "RWMutex allows concurrent readers, but writer (Lock) waits for all readers to release. If reader tries to acquire another RLock while writer is waiting, can deadlock (deadlock between reader and writer)."

Trap 4: "Go runtime always detects deadlocks"

Wrong. Runtime detects only global deadlock (all goroutines blocked).

Correct: "Go runtime detects global deadlock when all goroutines are sleeping. Partial deadlock (some blocked, some running) is not detected automatically. Need timeouts or monitoring."

Key Takeaways

1. **Deadlock = circular wait + hold + mutual exclusion + no preemption**
2. **Lock ordering prevents deadlock** (consistent acquisition order)
3. **Avoid nested locks** (acquire all at once or use channels)
4. **Always use defer for unlock** (prevents deadlock on panic)
5. **Go detects global deadlock only** (all goroutines blocked)
6. **Unbuffered channels deadlock without receiver**
7. **sync.Mutex is not recursive** (same goroutine can't lock twice)
8. **Database transactions can deadlock** (consistent update order)

Exercises

1. Write a deadlock with two mutexes. Run it. Observe "fatal error".

2. Fix the dining philosophers problem using asymmetric lock order.

3. Create a partial deadlock (some goroutines blocked, others running). Verify Go runtime doesn't detect it.

4. Implement timeout-based lock acquisition using select and channels.

5. Design API that makes deadlocks impossible (hint: use channels, not locks).

Next: livelocks.md - Understanding livelocks and starvation patterns.