Debugging Deadlocks

What is a Deadlock?

Deadlock: Set of goroutines blocked waiting for each other, unable to proceed.

Classic definition: "Circular wait" - goroutine A waits for B, B waits for C, C waits for A.

Example:

```go
mu1, mu2 := &sync.Mutex{}, &sync.Mutex{}

go func() {
    mu1.Lock()
    time.Sleep(10 * time.Millisecond)
    mu2.Lock()  // Waits for mu2
}()

go func() {
    mu2.Lock()
    time.Sleep(10 * time.Millisecond)
    mu1.Lock()  // Waits for mu1 - DEADLOCK!
}()
```

Result: Both goroutines blocked forever.

Necessary Conditions for Deadlock

Coffman conditions (all must hold):

1. **Mutual exclusion:** Resources held exclusively (mutexes)
2. **Hold and wait:** Holding one, waiting for another
3. **No preemption:** Can't forcibly take resources
4. **Circular wait:** A→B→C→A dependency cycle

Prevention: Break any one condition.

Detecting Deadlocks

Method 1: Stack Dump (SIGQUIT)

Send `SIGQUIT` (Ctrl+\) to get all goroutine stacks.

```bash
# Run program
go run main.go

# In another terminal
kill -QUIT <pid>

# Or press Ctrl+\ in same terminal
```

Output:

```
goroutine 5 [semacquire]:
sync.runtime_SemacquireMutex(0xc000010000)
    runtime/sema.go:71 +0x3e
sync.(*Mutex).Lock(0xc000010000)
    sync/mutex.go:138 +0x105
main.goroutine1()
    main.go:12 +0x45

goroutine 6 [semacquire]:
sync.runtime_SemacquireMutex(0xc000010008)
    runtime/sema.go:71 +0x3e
sync.(*Mutex).Lock(0xc000010008)
    sync/mutex.go:138 +0x105
main.goroutine2()
    main.go:18 +0x45
```

Analysis: Both blocked on `Mutex.Lock` - circular wait.

Method 2: Programmatic Stack Dump

```go
import (
    "os"
    "runtime"
)

func dumpAllStacks() {
    buf := make([]byte, 1<<20)  // 1MB buffer
    stacklen := runtime.Stack(buf, true)  // true = all goroutines
    os.Stderr.Write(buf[:stacklen])
}

// Call when stuck
if time.Since(start) > 5*time.Second {
    dumpAllStacks()
}
```

Method 3: pprof Goroutine Profile

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

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

    // Your code
}
```

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

# Analyze
less goroutines.txt
```

Shows: All goroutines with full stacks, including blocking reasons.

Method 4: Timeout Detection

```go
done := make(chan bool)

go func() {
    // Operation that might deadlock
    riskyOperation()
    done <- true
}()

select {
case <-done:
    fmt.Println("Success")
case <-time.After(5 * time.Second):
    fmt.Println("Deadlock suspected!")
    dumpAllStacks()
    panic("deadlock")
}
```

Analyzing Stack Traces

Example 1: Mutex Deadlock

```
goroutine 1 [semacquire, 10 minutes]:
sync.(*Mutex).Lock(0xc000100080)
    sync/mutex.go:138
main.functionA()
    main.go:15

goroutine 2 [semacquire, 10 minutes]:
sync.(*Mutex).Lock(0xc000100088)
    sync/mutex.go:138
main.functionB()
    main.go:25
```

Key indicators:

- State: `semacquire` (waiting on semaphore/mutex)
- Duration: Long time (minutes)
- Multiple goroutines in same state

Diagnosis: Compare mutex addresses, check lock ordering in code.

Example 2: Channel Deadlock

```
goroutine 1 [chan send, 5 minutes]:
main.producer()
    main.go:20

goroutine 2 [chan receive, 5 minutes]:
main.consumer()
    main.go:30
```

Diagnosis:

- Producer blocked sending (channel full, no receiver)
- Consumer blocked receiving (channel empty, no sender)
- Check channel size, sender/receiver counts

Example 3: WaitGroup Deadlock

```
goroutine 1 [semacquire, 10 minutes]:
sync.(*WaitGroup).Wait()
    sync/waitgroup.go:130
main.main()
    main.go:10
```

Diagnosis: `Wait()` blocks forever - workers never called `Done()` .

Real Production Deadlock Stories

Story 1: Docker Hub Registry Outage (2014)

Cause: Race condition in database connection pool.

```go
type Pool struct {
    mu    sync.Mutex
    conns []*Conn
}

func (p *Pool) Get() *Conn {
    p.mu.Lock()
    defer p.mu.Unlock()

    if len(p.conns) == 0 {
        return p.create()  // BUG: create() locks p.mu too!
    }

    conn := p.conns[0]
    p.conns = p.conns[1:]
    return conn
}

func (p *Pool) create() *Conn {
    p.mu.Lock()  // DEADLOCK: Already locked in Get()
    defer p.mu.Unlock()

    conn := newConnection()
    return conn
}
```

Impact: Registry unavailable for hours.
Fix: Don't lock in `create()`, or unlock before calling it.

Story 2: Kubernetes API Server Deadlock (2017)

Cause: Two mutexes acquired in different orders.

```go
type Cache struct {
    mu1 sync.Mutex
    mu2 sync.Mutex
}

// Path A
func (c *Cache) Update() {
    c.mu1.Lock()
    defer c.mu1.Unlock()
    c.mu2.Lock()
    defer c.mu2.Unlock()
    // update
}

// Path B
func (c *Cache) Invalidate() {
    c.mu2.Lock()  // Opposite order!
    defer c.mu2.Unlock()
    c.mu1.Lock()
    defer c.mu1.Unlock()
    // invalidate
}
```

Impact: API server frozen, cluster inoperable.
Fix: Consistent lock ordering.

Story 3: Etcd Deadlock (2016)

Cause: Nested context cancellation.

```go
ctx, cancel := context.WithCancel(parentCtx)

go func() {
    <-parentCtx.Done()
    cancel()  // Cancels child
}()
```

```go
go func() {
    <-ctx.Done()
    // Tries to cancel parent - circular!
}()
```

Impact: Etcd cluster leader election stuck.
Fix: Clear parent-child context hierarchy.

Preventing Deadlocks

Strategy 1: Lock Ordering

Rule: Always acquire locks in the same order.

```go
// BAD: Inconsistent order
func transfer(from, to *Account, amount int) {
    from.mu.Lock()
    to.mu.Lock()  // Order depends on call
    // ...
}

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

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

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

Strategy 2: Lock Timeout

Idea: Don't wait forever, fail if can't acquire.

```go
func tryLock(mu *sync.Mutex, timeout time.Duration) bool {
    ch := make(chan struct{})
    go func() {
        mu.Lock()
        close(ch)
    }()

    select {
    case <-ch:
        return true
    case <-time.After(timeout):
        return false  // Couldn't acquire
    }
}

// Use it
if !tryLock(&mu1, time.Second) {
    return fmt.Errorf("lock timeout")
}
defer mu1.Unlock()
```

Note: Standard `sync.Mutex` doesn't support timeouts. Consider `github.com/sasha-s/go-deadlock` for this.

Strategy 3: Single Lock

Idea: Protect related data with one lock, not multiple.

```
// BAD: Two locks = potential deadlock
type BankAccount struct {
    balanceMu sync.Mutex
    balance   int

    historyMu sync.Mutex
    history   []Transaction
}

// GOOD: One lock
type BankAccount struct {
    mu      sync.Mutex
    balance int
    history []Transaction
}
```

Strategy 4: Avoid Nesting

Rule: Don't call functions while holding locks.

```
// BAD: Calls function while locked
func (s *Server) Handle(req Request) {
    s.mu.Lock()
    defer s.mu.Unlock()

    s.process(req)  // What if process() locks s.mu?
}

// GOOD: Copy data, unlock, then process
func (s *Server) Handle(req Request) {
    s.mu.Lock()
    data := s.getData()
    s.mu.Unlock()
    s.process(data)  // No lock held
}
```

Strategy 5: Use Channels

Idea: Channels can't deadlock (they block, but Go detects).

```go
// Using mutexes (can deadlock)
var mu1, mu2 sync.Mutex

// Using channels (Go detects deadlock)
ch1, ch2 := make(chan int), make(chan int)

// If no goroutine can proceed, Go panics:
// "fatal error: all goroutines are asleep - deadlock!"
```

Tools for Deadlock Detection

Tool 1: go-deadlock

Replacement for `sync.Mutex` with deadlock detection.

```
go get github.com/sasha-s/go-deadlock
```

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

var mu deadlock.Mutex  // Drop-in replacement

func main() {
    deadlock.Opts.DeadlockTimeout = 10 * time.Second

    mu.Lock()
    // If lock held > 10s, prints stack trace
}
```

Tool 2: go-tools/deadcode

Static analysis to find potential deadlocks.

```
go install honnef.co/go/tools/cmd/staticcheck@latest
staticcheck ./...
```

Tool 3: goleak

Detects goroutine leaks (often cause of deadlocks).

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

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

    // Your test
}
```

Debugging Live Deadlock

Step 1: Identify Stuck Goroutines

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

# Find long-blocked goroutines
grep -A 10 "semacquire" stuck.txt
```

Step 2: Extract Lock Addresses

```
goroutine 42 [semacquire]:
sync.(*Mutex).Lock(0xc000100080)  ← Mutex address
```

Step 3: Find All Waiters

Search for same mutex address across all stacks.

```
grep "0xc000100080" stuck.txt
```

If multiple goroutines waiting: Classic deadlock.

Step 4: Identify Lock Holders

Look for goroutines in running state or holding locks.

```
goroutine 23 [running]:
main.longOperation()
    main.go:45  ← Check if this holds the mutex
```

Step 5: Trace Lock Acquisition

Build dependency graph:

- Goroutine A holds mutex1, waits for mutex2
- Goroutine B holds mutex2, waits for mutex3
- Goroutine C holds mutex3, waits for mutex1

Cycle detected: Deadlock confirmed.

Example: Full Deadlock Debugging

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

type Resource struct {
    mu sync.Mutex
    id int
}

func transfer(r1, r2 *Resource) {
    r1.mu.Lock()
    fmt.Printf("Locked r%d\n", r1.id)
    time.Sleep(100 * time.Millisecond)

    r2.mu.Lock()  // Potential deadlock
    fmt.Printf("Locked r%d\n", r2.id)

    // Transfer logic

    r2.mu.Unlock()
    r1.mu.Unlock()
}

func main() {
    res1 := &Resource{id: 1}
    res2 := &Resource{id: 2}

    done := make(chan bool)

    go func() {
        transfer(res1, res2)
        done <- true
    }()

    go func() {
        transfer(res2, res1)  // Opposite order - DEADLOCK!
        done <- true
    }()

    select {
    case <-done:
        fmt.Println("Success")
    case <-time.After(time.Second):
        fmt.Println("DEADLOCK DETECTED!")

        // Dump stacks
        buf := make([]byte, 1<<16)
        stacklen := runtime.Stack(buf, true)
        fmt.Printf("%s\n", buf[:stacklen])
    }
}
```

Output:

```
Locked r1
Locked r2
DEADLOCK DETECTED!

goroutine 6 [semacquire]:
sync.(*Mutex).Lock(0xc000100080)  ← Waiting for r2
main.transfer(0xc000100000, 0xc000100080)
    main.go:18

goroutine 7 [semacquire]:
sync.(*Mutex).Lock(0xc000100000)  ← Waiting for r1
main.transfer(0xc000100080, 0xc000100000)
    main.go:18
```

Fix:

```
func transfer(r1, r2 *Resource) {
    // Always lock in consistent order
    first, second := r1, r2
    if r1.id > r2.id {
        first, second = r2, r1
    }

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

    // Transfer logic
}
```

Deadlock vs Livelock

Deadlock: Goroutines blocked, not progressing
Livelock: Goroutines active but not progressing

```
// Livelock example
go func() {
    for {
        if tryLock(&mu1) {
            if tryLock(&mu2) {
                // work
                mu2.Unlock()
                mu1.Unlock()
                return
            } else {
                mu1.Unlock()  // Politely release
                time.Sleep(time.Millisecond)  // Try again
            }
        }
    }
}()
```

Both goroutines keep acquiring/releasing, never progressing.

Go's Deadlock Detector

Go runtime detects **some** deadlocks:

```
func main() {
    ch := make(chan int)
    <-ch  // No sender - deadlock!
}

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

Limitation: Only detects global deadlock (all goroutines stuck).

```
func main() {
    ch := make(chan int)

    go func() {
        time.Sleep(time.Hour)  // This goroutine busy
    }()

    <-ch  // Stuck, but Go doesn't detect (other goroutine alive)
}
```

Interview Questions

Q: "How do you detect a deadlock in production?"

"1) Monitor metrics: goroutine count increasing, request latency spiking. 2) Enable pprof, fetch goroutine profile with debug=2: `curl localhost:6060/debug/pprof/goroutine?debug=2` . 3) Look for goroutines in 'semacquire' state for long duration. 4) Identify mutex addresses, find circular wait. 5) Use tools like go-deadlock library for timeout-based detection. Once detected, dump stacks, trace lock acquisition order, fix by ensuring consistent ordering."

Q: "What's the difference between deadlock and livelock?"

"Deadlock: goroutines blocked, not consuming CPU, waiting for each other, no progress. Example: circular mutex wait. Livelock: goroutines active, consuming CPU, continually changing state in response to each other but no progress. Example: two goroutines repeatedly acquiring and releasing locks to be 'polite'. Deadlock = stuck waiting, livelock = stuck trying."

Q: "How do you prevent deadlocks?"

"1) Lock ordering: Always acquire locks in same order (sort by resource ID). 2) Single lock: Use one lock instead of multiple for related data. 3) Avoid nesting: Don't call functions while holding locks. 4) Use channels: Go detects channel deadlocks. 5) Timeouts: Don't wait forever, fail fast. 6) Static analysis: Use go-deadlock library or staticcheck."

Q: "What does Go's 'fatal error: all goroutines are asleep' mean?"

"Go's runtime detected global deadlock - all goroutines blocked with no way to proceed. Happens when: 1) All goroutines waiting on channels with no sender/receiver. 2) All waiting on mutexes held by each other. Limitation: Only detects when ALL goroutines stuck. If one goroutine busy (e.g., time.Sleep), Go won't detect even if others deadlocked."

Deadlock Checklist

Before deploying:

- [ ] All locks acquired in consistent order
- [ ] No function calls while holding locks
- [ ] All `Lock()` calls have matching `Unlock()`
- [ ] All WaitGroups have matching `Done()` calls
- [ ] Channel senders/receivers properly coordinated
- [ ] Timeouts on blocking operations
- [ ] Enabled deadlock detection (go-deadlock in dev/test)
- [ ] Added goroutine count monitoring
- [ ] pprof endpoints exposed for debugging
- [ ] Tested under high concurrency (stress tests)

Key Takeaways

1. **Deadlock = circular wait with no escape**
2. **Detect with stack dumps (SIGQUIT or pprof)**
3. **Look for 'semacquire' state in goroutines**
4. **Always acquire locks in same order**
5. **Avoid calling functions while holding locks**
6. **Use timeouts to fail fast**
7. **Go detects some deadlocks (all goroutines stuck)**
8. **Tools: go-deadlock, staticcheck, goleak**
9. **Monitor goroutine count in production**
10. **Test with high concurrency to expose issues**

Exercises

1. Create intentional deadlock, debug with pprof.

2. Build lock dependency graph from stack traces.

3. Fix deadlock by reordering lock acquisition.

4. Implement timeout-based lock acquisition.
5. Use go-deadlock to find potential deadlocks in codebase.

Next: ../07-lld-hld/designing-concurrent-components.md - Design principles for concurrent systems.