Livelocks and Starvation

Part 1: Livelocks

What is a Livelock?

Livelock: Goroutines are not blocked, but continuously change state in response to each other without making progress.

Key difference from deadlock:

- **Deadlock:** Goroutines blocked, waiting forever
- **Livelock:** Goroutines active, but stuck in cycle without progress

Analogy: Two people trying to pass each other in hallway. Both step left, both step right, both step left again... Nobody passes (active but no progress).

Example 1: Polite Philosophers

```
type Fork struct {
    mu    sync.Mutex
    inUse bool
}

func (f *Fork) tryPickup() bool {
    f.mu.Lock()
    defer f.mu.Unlock()

    if f.inUse {
        return false
    }
    f.inUse = true
    return true
}

func (f *Fork) putDown() {
    f.mu.Lock()
    defer f.mu.Unlock()
    f.inUse = false
}

func philosopher(left, right *Fork) {
    for {
        // Try to pick up left fork
        if !left.tryPickup() {
            time.Sleep(time.Microsecond)  // Wait a bit
            continue  // Try again
        }

        // Got left, try right
        if !right.tryPickup() {
            left.putDown()  // Be polite: release left
```

```
            time.Sleep(time.Microsecond)  // Wait
            continue  // Try again
        }

        // Eat (both forks acquired)
        fmt.Println("Eating")
        left.putDown()
        right.putDown()
        return
    }
}

// Livelock scenario:
// All philosophers pick up left fork
// All try right fork (fails)
// All put down left fork
// All wait same duration
// All pick up left fork again (cycle repeats)
```

Why livelock: All philosophers respond to conflict identically (deterministic retry).

Solution: Random backoff

```
func philosopher(left, right *Fork) {
    for {
        if !left.tryPickup() {
            // Random backoff breaks synchronization
            time.Sleep(time.Duration(rand.Intn(100)) * time.Microsecond)
            continue
        }

        if !right.tryPickup() {
            left.putDown()
            time.Sleep(time.Duration(rand.Intn(100)) * time.Microsecond)
            continue
        }

        // Eat
        fmt.Println("Eating")
        left.putDown()
        right.putDown()
        return
    }
}
```

Example 2: Message Passing Livelock

```
type Message struct {
    From   int
    To     int
```

```
    Retry  int
}

func worker(id int, in <-chan Message, out chan<- Message) {
    for msg := range in {
        if msg.To == id {
            // Process message
            process(msg)
        } else {
            // Forward to next worker
            msg.Retry++
            if msg.Retry < 10 {
                out <- msg  // Forward
            }
        }
    }
}

// Livelock: Message bounces between workers if routing logic is broken
// Workers are active (forwarding messages) but no progress (never processed)
```

Solution: Add TTL or circuit breaker

```
func worker(id int, in <-chan Message, out chan<- Message, dead chan<- Message) {
    for msg := range in {
        if msg.To == id {
            process(msg)
        } else {
            msg.Retry++
            if msg.Retry >= 10 {
                dead <- msg  // Dead letter queue
            } else {
                out <- msg
            }
        }
    }
}
```

Example 3: HTTP Retry Livelock

```
func fetchWithRetry(url string, maxRetries int) error {
    for i := 0; i < maxRetries; i++ {
        resp, err := http.Get(url)
        if err == nil && resp.StatusCode == 200 {
            return nil
        }

        // Immediate retry without backoff
        // If server is overloaded, all clients retry simultaneously → livelock
    }
```

```
    return fmt.Errorf("failed after %d retries", maxRetries)
}

// Livelock: All clients retry at same time, server stays overloaded
```

Solution: Exponential backoff with jitter

```
func fetchWithRetry(url string, maxRetries int) error {
    backoff := time.Millisecond * 100

    for i := 0; i < maxRetries; i++ {
        resp, err := http.Get(url)
        if err == nil && resp.StatusCode == 200 {
            return nil
        }

        // Exponential backoff
        sleep := backoff * time.Duration(1<<i)

        // Add jitter (±25%)
        jitter := time.Duration(rand.Float64() * 0.5 * float64(sleep))
        time.Sleep(sleep + jitter)
    }
    return fmt.Errorf("failed after %d retries", maxRetries)
}
```

Real-World Failure: AWS S3 Outage (2017)

Date: February 28, 2017
Duration: 4 hours
Incident: S3 subsystem overloaded, retry storms

Simplified scenario:

```
// Client library
func uploadFile(s3 *S3Client, file File) error {
    for attempt := 0; attempt < 100; attempt++ {
        err := s3.PutObject(file)
        if err == nil {
            return nil
        }

        // Fast retry without backoff
        time.Sleep(10 * time.Millisecond)
    }
    return errors.New("upload failed")
}

// What happened:
// 1. S3 subsystem slows down (operator error during maintenance)
// 2. Clients see timeouts, start retrying immediately
```

```
// 3. Retry storm overwhelms S3 further
// 4. More timeouts → more retries → livelock
// 5. S3 unable to recover for 4 hours
```

Fix: Exponential backoff, circuit breaker, rate limiting.

Livelock Detection

Characteristics:

1. CPU usage high (goroutines active)
2. No progress (no meaningful work completed)
3. Repetitive patterns in logs

Monitoring:

```
type Worker struct {
    processed   int64  // Completed work
    lastChecked int64
}

func (w *Worker) checkProgress() {
    current := atomic.LoadInt64(&w.processed)
    prev := atomic.LoadInt64(&w.lastChecked)

    if current == prev {
        // No progress in last interval → livelock?
        log.Warn("Possible livelock: no progress")
    }

    atomic.StoreInt64(&w.lastChecked, current)
}

// Run periodically
go func() {
    ticker := time.NewTicker(5 * time.Second)
    for range ticker.C {
        worker.checkProgress()
    }
}()
```

Part 2: Starvation

What is Starvation?

Starvation: A goroutine is perpetually denied resources it needs, unable to make progress.

Key difference:

- **Deadlock:** All goroutines blocked
- **Livelock:** Goroutines active but no progress
- **Starvation:** Some goroutines make progress, others don't

Example 1: Writer Starvation (RWMutex)

```
var (
    mu   sync.RWMutex
    data int
)

// Many readers
func reader() {
    for {
        mu.RLock()
        _ = data  // Read
        mu.RUnlock()

        // Short pause
        time.Sleep(time.Microsecond)
    }
}

// Single writer
func writer() {
    for {
        mu.Lock()  // Waits for all readers
        data++
        mu.Unlock()

        time.Sleep(time.Microsecond)
    }
}

// If readers are constant, writer may never acquire lock → starvation
```

Go's RWMutex implementation is writer-preferring (mitigates this), but heavy read load can still starve writers.

Solution: Bounded work

```
func reader() {
    for {
        mu.RLock()
        _ = data
        mu.RUnlock()

        // Longer pause to give writer a chance
        time.Sleep(time.Millisecond)
    }
}
```

Or use separate read-through cache.

Example 2: Goroutine Scheduling Starvation

```
// Low-priority goroutine
go func() {
    for {
        // CPU-intensive work
        compute()
    }
}()

// High-priority goroutine (lots of them)
for i := 0; i < 100; i++ {
    go func() {
        for {
            compute()
            runtime.Gosched()  // Yield, but immediately requeued
        }
    }()
}

// Low-priority goroutine may be starved of CPU time
```

Go scheduler is preemptive but fair. Starvation rare but possible under extreme load.

Solution: Cooperative yielding

```
func lowPriority() {
    for {
        compute()
        runtime.Gosched()  // Yield to others
    }
}
```

Example 3: Channel Starvation

```
ch := make(chan int)

// Producer (fast)
go func() {
    for i := 0; ; i++ {
        ch <- i
    }
}()

// Consumer 1 (always ready)
go func() {
    for v := range ch {
        process(v)  // Fast
    }
```

```
}()

// Consumer 2 (slow, needs data occasionally)
go func() {
    for {
        time.Sleep(time.Second)  // Slow
        select {
        case v := <-ch:
            process(v)
        default:
            // Channel empty → starved
        }
    }
}()

// Consumer 1 receives all messages, Consumer 2 starved
```

Solution: Fair distribution (fan-out)

```
ch := make(chan int)
ch1 := make(chan int)
ch2 := make(chan int)

// Fair distributor
go func() {
    for v := range ch {
        select {
        case ch1 <- v:
        case ch2 <- v:
        }
    }
}()

// Consumers
go func() { for v := range ch1 { process(v) } }()
go func() { for v := range ch2 { process(v) } }()
```

Example 4: Lock Starvation (Unfair Locks)

```
var mu sync.Mutex

// Many short holders
for i := 0; i < 100; i++ {
    go func() {
        for {
            mu.Lock()
            // Very short work
            mu.Unlock()
        }
    }()
```

```
}

// One long holder
go func() {
    for {
        mu.Lock()
        time.Sleep(time.Millisecond)  // Tries to hold longer
        mu.Unlock()
    }
}()

// Long holder may be starved (never gets lock)
```

Go's sync.Mutex is fair (FIFO order in contention), so this is rare. But high contention can delay specific goroutines.

Real-World Failure: Video Streaming Starvation (2021)

Service: Video transcoding service
Issue: High-priority transcodes starved low-priority

Scenario:

```
type Job struct {
    Priority int
    Data     []byte
}

var jobQueue = make(chan Job, 1000)

// Workers always pick highest priority
func worker() {
    for {
        var job Job

        // Non-blocking scan for highest priority
        select {
        case j := <-jobQueue:
            job = j
            // Check if higher priority job arrived
            for {
                select {
                case j2 := <-jobQueue:
                    if j2.Priority > job.Priority {
                        jobQueue <- job  // Requeue lower priority
                        job = j2
                    } else {
                        jobQueue <- j2
                    }
                default:
                    goto process
                }
```

```
            }
        }

    process:
        process(job)
    }
}

// Low-priority jobs never processed (starved) if high-priority jobs keep arriving
```

Impact: Low-priority customers waited hours for transcoding.

Fix: Priority aging or separate queues with guaranteed time slices.

```
type Job struct {
    Priority  int
    Timestamp time.Time
}

// Age priority over time
func (j *Job) effectivePriority() int {
    age := time.Since(j.Timestamp).Seconds()
    return j.Priority + int(age/60)  // +1 priority per minute
}
```

Preventing Starvation

Strategy 1: Fair Scheduling

Use FIFO queues (Go's sync.Mutex is FIFO under contention).

Strategy 2: Priority Aging

```
type Task struct {
    Priority int
    Age      int
}

// Increase priority over time
func age(t *Task) {
    t.Priority += t.Age / 10
}
```

Strategy 3: Work Quotas

```
// Each goroutine gets N operations before yielding
const quota = 100

func worker() {
    count := 0
    for {
```

```
        work()
        count++
        if count >= quota {
            runtime.Gosched()
            count = 0
        }
    }
}
```

Strategy 4: Separate Queues

```
highPriority := make(chan Task, 100)
lowPriority := make(chan Task, 100)

// Fair scheduling: alternate between queues
for {
    select {
    case t := <-highPriority:
        process(t)
    default:
        select {
        case t := <-lowPriority:
            process(t)
        case t := <-highPriority:
            process(t)
        }
    }
}
```

Interview Traps

Trap 1: "Livelock and deadlock are the same"

Wrong.
Correct: "Deadlock: goroutines blocked, no CPU usage. Livelock: goroutines active, high CPU, no progress. Livelock often result of deadlock avoidance (retry logic)."

Trap 2: "RWMutex prevents starvation"

Wrong. Heavy read load can starve writers.
Correct: "Go's RWMutex is writer-preferring, but continuous readers can still delay writer acquisition. Need to bound read duration or use separate caching layer."

Trap 3: "Random backoff always prevents livelock"

Incomplete. Random backoff reduces livelock probability but doesn't eliminate it.
Correct: "Random backoff with exponential increase reduces collision probability. Should also add max retry limit and circuit breaker for robustness."

Trap 4: "Starvation means deadlock"

Wrong. Starvation means SOME goroutines blocked, not all.
Correct: "Starvation: specific goroutines can't acquire resources while others make progress. Deadlock: all goroutines blocked. Starvation is partial, deadlock is global."

Key Takeaways

Livelocks

1. **Livelock = active but no progress** (not blocked)
2. **Caused by deterministic retry logic** (all goroutines respond identically)
3. **Random backoff breaks synchronization** (exponential + jitter)
4. **Circuit breaker prevents retry storms**
5. **Monitor progress, not just activity** (detect livelock)

Starvation

1. **Starvation = perpetual resource denial** (specific goroutines)
2. **RWMutex: readers can starve writers**
3. **Priority inversion: low-priority never run**
4. **Fair scheduling uses FIFO or round-robin**
5. **Priority aging prevents indefinite starvation**
6. **Separate queues with quotas ensure fairness**

Exercises

1. Write a livelock with 2 goroutines and 2 channels. Both try to send and receive simultaneously.
2. Fix the polite philosophers using random backoff. Verify no livelock.
3. Create writer starvation with RWMutex. Monitor how long writer waits.
4. Implement priority aging: low-priority tasks become high-priority over time.
5. Design a fair scheduler for high/low priority tasks with guaranteed minimum throughput for low-priority.

Next: false-sharing.md - Understanding cache-line bouncing and performance bugs.