Race Conditions

What is a Race Condition?

Race condition: When the correctness of a program depends on the timing or ordering of uncontrollable events (goroutine scheduling, CPU execution order, memory access timing).

Data race (specific type): Concurrent access to shared memory where at least one access is a write, without synchronization.

Critical distinction:

- **Data race** (memory safety): Undefined behavior at memory level
- **Race condition** (logic bug): Incorrect behavior at application level

You can have:

- Data race without race condition (rare)
- Race condition without data race (common: incorrect synchronization logic)
- Both (most common bug)

Example: Check-Then-Act Race

```go
var balance int
var mu sync.Mutex

func withdraw(amount int) bool {
    mu.Lock()
    canWithdraw := balance >= amount  // Check
    mu.Unlock()

    if canWithdraw {
        mu.Lock()
        balance -= amount  // Act
        mu.Unlock()
        return true
    }
    return false
}

// Goroutine 1: withdraw(100), balance=100
// Goroutine 2: withdraw(100), balance=100

// Both pass check (balance >= 100)
// Both perform withdrawal
// Final balance: -100 (incorrect!)
```

No data race (all accesses protected).
Race condition: Check and act are not atomic.

Fix: Make check-then-act atomic.

```
func withdraw(amount int) bool {
    mu.Lock()
    defer mu.Unlock()

    if balance >= amount {
        balance -= amount
        return true
    }
    return false
}
```

Example: Read-Modify-Write Race

```
var counter int

func increment() {
    counter++  // Read, modify, write
}

// Goroutine 1: counter++ (read 0, write 1)
// Goroutine 2: counter++ (read 0, write 1)
// Expected: counter = 2
// Actual: counter = 1 (lost update)
```

Disassembly of counter++:

```
MOV  R1, [counter]   ; Read
ADD  R1, 1           ; Modify
MOV  [counter], R1   ; Write
```

Interleaving:

```
G1: MOV R1, [counter]  ; R1 = 0
G2: MOV R2, [counter]  ; R2 = 0
G1: ADD R1, 1          ; R1 = 1
G2: ADD R2, 1          ; R2 = 1
G1: MOV [counter], R1  ; counter = 1
G2: MOV [counter], R2  ; counter = 1 (overwrites G1's update)
```

Fix: Use atomic operation.

```
var counter int64

func increment() {
    atomic.AddInt64(&counter, 1)
}
```

Example: Idempotent Operation Race

```
var initialized bool
var resource *Resource

func getResource() *Resource {
    if !initialized {  // (1) Check
        resource = &Resource{}  // (2) Initialize
        initialized = true      // (3) Mark
    }
    return resource
}

// Goroutine 1: getResource()
// Goroutine 2: getResource()

// Both see initialized=false
// Both create resource
// Second overwrites first (resource leak)
```

Fix: Use sync.Once.

```
var (
    resource *Resource
    once     sync.Once
)

func getResource() *Resource {
    once.Do(func() {
        resource = &Resource{}
    })
    return resource
}
```

Example: Map Race (Most Common Data Race in Go)

```
m := make(map[string]int)

// Goroutine 1
m["key"] = 1

// Goroutine 2
v := m["key"]

// Goroutine 3
delete(m, "key")
```

Data race: Concurrent map operations without synchronization.

Go runtime detects this:

```
fatal error: concurrent map writes
```

Fix 1: Use mutex.

```
var (
    m  = make(map[string]int)
    mu sync.RWMutex
)

// Write
mu.Lock()
m["key"] = 1
mu.Unlock()

// Read
mu.RLock()
v := m["key"]
mu.RUnlock()
```

Fix 2: Use sync.Map (for specific use cases).

```
var m sync.Map

m.Store("key", 1)
v, ok := m.Load("key")
m.Delete("key")
```

Example: Slice Append Race

```
var items []int

// Goroutine 1
items = append(items, 1)

// Goroutine 2
items = append(items, 2)
```

Why it races:

1. **Read:** Both read `len(items)` and `cap(items)`
2. **Decision:** Both decide if reallocation needed
3. **Write:** Both write to backing array or create new array
4. **Assign:** Both assign to `items` variable

Possible outcomes:

- Lost updates (one append lost)
- Corrupted slice header

- Panic (out of bounds)

Fix: Protect with mutex.

```go
var (
    items []int
    mu    sync.Mutex
)

func add(v int) {
    mu.Lock()
    items = append(items, v)
    mu.Unlock()
}
```

Example: Time-of-Check to Time-of-Use (TOCTOU)

```go
func processFile(filename string) error {
    // Check
    if _, err := os.Stat(filename); err != nil {
        return err  // File doesn't exist
    }

    // Use (later)
    time.Sleep(time.Millisecond)  // Simulated delay

    data, err := os.ReadFile(filename)  // File might be deleted now!
    if err != nil {
        return err
    }

    return process(data)
}
```

Race: File state changes between check and use.

Fix: Don't check; just use (EAFP: Easier to Ask Forgiveness than Permission).

```go
func processFile(filename string) error {
    data, err := os.ReadFile(filename)
    if err != nil {
        return err  // Handle error
    }
    return process(data)
}
```

Real-World Failure: Knight Capital ($440M Loss, 2012)

Date: August 1, 2012
Loss: $440 million in 45 minutes

Root cause: Race condition in deployment + stale code execution.

Simplified scenario:

```go
var orderCount int
var processedOrders map[int]bool  // Old code used this
var mu sync.Mutex

// New code (7 servers deployed)
func placeOrder(order Order) {
    mu.Lock()
    orderCount++
    id := orderCount
    mu.Unlock()

    submitOrder(order, id)
}

// Old code (1 server not deployed)
func placeOrder(order Order) {
    mu.Lock()
    orderCount++
    id := orderCount

    // Bug: Doesn't check if already processed
    if processedOrders[id] {
        mu.Unlock()
        return
    }
    processedOrders[id] = true
    mu.Unlock()

    // Sends order 8 times (reprocessing old IDs)
    for i := 0; i < 8; i++ {
        submitOrder(order, id)
    }
}
```

What happened:

1. Order sent to old server (1 of 8)
2. Old code doesn't see order in `processedOrders` (stale state)
3. Resubmits order 8 times
4. Race between new servers' state and old server's stale state
5. 4 million orders placed in 45 minutes

Impact: $440 million loss, company bankrupt.

Lessons:

- Deployment races (partial rollouts)
- State synchronization across servers
- Idempotency checks

Real-World Failure: Therac-25 (Deaths, 1985-1987)

Date: 1985-1987
Impact: 6 radiation overdoses, 3 deaths

Root cause: Race condition between operator input and machine state.

Simplified version:

```go
type Machine struct {
    mode      string  // "X-ray" or "Electron"
    intensity int
}

var machine Machine

// UI goroutine
func setMode(m string) {
    machine.mode = m
    // Race: Operator can change mode here
}

// Control goroutine
func setIntensity(i int) {
    machine.intensity = i
    // Race: Mode might change before fire()
}

// Execution goroutine
func fire() {
    // Reads mode and intensity
    if machine.mode == "X-ray" {
        applyIntensity(machine.intensity)  // Safe dose
    } else {
        // Mode was changed to "Electron" after intensity set
        applyIntensity(machine.intensity)  // 100x overdose!
    }
}
```

Actual bug: Race between setting mode and calibrating magnets. If operator changed mode quickly (within 8ms), magnet position didn't match mode → radiation beam unfiltered.

Impact: Patients received 100x intended dose.

Lessons:

- Race conditions can kill
- Timing-dependent bugs are deadly
- Must synchronize state transitions

Real-World Failure: Docker Hub (2018)

Date: April 2018
Incident: Index corruption, data loss

Root cause (simplified): Race condition in database index updates.

```go
type Index struct {
    entries map[string]*Entry
    mu      sync.RWMutex
}

func (idx *Index) Update(key string, value *Entry) {
    idx.mu.RLock()  // Read lock
    existing := idx.entries[key]
    idx.mu.RUnlock()

    // Merge logic (outside lock)
    merged := merge(existing, value)

    idx.mu.Lock()  // Write lock
    // Race: Another goroutine might have updated key
    idx.entries[key] = merged  // Overwrites concurrent update
    idx.mu.Unlock()
}
```

Problem: Read-compute-write is not atomic. Concurrent updates lost.

Fix: Hold write lock for entire operation.

```go
func (idx *Index) Update(key string, value *Entry) {
    idx.mu.Lock()
    defer idx.mu.Unlock()

    existing := idx.entries[key]
    merged := merge(existing, value)
    idx.entries[key] = merged
}
```

Impact: Index corrupted, some images lost.

Detecting Race Conditions

1. The Race Detector

```
go run -race main.go
go test -race ./...
go build -race
```

What it detects:

- Unsynchronized accesses to shared memory
- At least one access is write
- No happens-before relationship

Example output:

```
==================
WARNING: DATA RACE
Write at 0x00c000018090 by goroutine 7:
  main.increment()
      /path/race.go:10 +0x44

Previous write at 0x00c000018090 by goroutine 6:
  main.increment()
      /path/race.go:10 +0x44
==================
```

Performance overhead: 5-10x slowdown, 5-10x memory usage.

CI/CD: Always run tests with `-race` in CI.

2. Static Analysis (go vet)

```
go vet ./...
```

Detects:

- Copying locks (mutex by value)
- Printf argument mismatches
- Unreachable code

3. Manual Code Review Checklist

For every shared variable, ask:

1. **Is it accessed from multiple goroutines?**
2. **Are all accesses protected by the same lock?**
3. **Are read and write operations atomic at business-logic level?**
4. **Is there a happens-before relationship?**

Common Race Patterns and Fixes

Pattern 1: Counter Race

Wrong:

```
var counter int
counter++
```

Right:

```
var counter int64
```

```go
atomic.AddInt64(&counter, 1)
```

Pattern 2: Map Race

Wrong:

```go
m := make(map[string]int)
m["key"]++
```

Right:

```go
var (
    m  = make(map[string]int)
    mu sync.Mutex
)
mu.Lock()
m["key"]++
mu.Unlock()
```

Pattern 3: Lazy Initialization Race

Wrong:

```go
var resource *Resource
if resource == nil {
    resource = &Resource{}
}
```

Right:

```go
var (
    resource *Resource
    once     sync.Once
)
once.Do(func() {
    resource = &Resource{}
})
```

Pattern 4: Double-Checked Locking Race

Wrong:

```go
if config == nil {  // Check
    mu.Lock()
    if config == nil {
        config = &Config{}  // Initialize
    }
    mu.Unlock()
}
```

Right:

```go
var (
    config *Config
    once   sync.Once
)
once.Do(func() {
    config = &Config{}
})
```

Pattern 5: Publish-Subscribe Race

Wrong:

```go
var subscribers []chan Event
subscribers = append(subscribers, ch)  // Race
```

Right:

```go
var (
    subscribers []chan Event
    mu          sync.Mutex
)
mu.Lock()
subscribers = append(subscribers, ch)
mu.Unlock()
```

Interview Traps

Trap 1: "This is just a counter, no need for synchronization"

Wrong: "It's just an int, reads/writes are atomic."
Correct: "Even if reads/writes don't tear, counter++ is read-modify-write, which is not atomic. Must use atomic.AddInt64 or mutex."

Trap 2: "I lock here and here, so it's safe"

Interviewer shows code with some accesses protected, others not.

Wrong: "Most accesses are protected, should be fine."
Correct: "ALL accesses must be protected by the same lock, or use atomic operations. Even one unprotected access creates a data race."

Trap 3: "Race detector didn't report anything"

Wrong: "Race detector proves it's correct."
Correct: "Race detector is dynamic analysis—only detects races in executed code paths. Must also verify happens-before relationships and ensure test coverage."

Trap 4: "It's read-only after initialization"

Wrong: "Once initialized, multiple readers are safe."
Correct: "Publication must be synchronized. Without happens-before, readers might see partial initialization or stale pointers. Use sync.Once or atomic.Pointer."

Key Takeaways

1. **Data race ≠ race condition** (memory behavior vs. logical behavior)
2. **All shared memory accesses need synchronization** (no exceptions)
3. **Check-then-act must be atomic** (hold lock for entire operation)
4. **Read-modify-write must be atomic** (use atomic ops or mutex)
5. **Map operations are not atomic** (protect with mutex or use sync.Map)
6. **Slice append is not atomic** (protect with mutex)
7. **Race detector is necessary, not sufficient** (dynamic analysis)
8. **Always run go test -race in CI** (catch races early)

Exercises

1. Find 3 races in your own code using race detector.

2. Write programs with these races, run with `-race`:

   - Counter race
   - Map race
   - Slice append race

3. Fix this code:

```go
var cache map[string]string
func get(key string) string {
    if cache == nil {
        cache = make(map[string]string)
    }
    return cache[key]
}
```

4. Explain why this races:

```go
var config *Config
go func() { config = &Config{} }()
go func() { if config != nil { use(config) } }()
```

5. Design a code review checklist for catching races.

Next: deadlocks.md - Understanding and preventing deadlocks.