Common Go Concurrency Interview Questions

Fundamentals (Entry Level)

Q1: What is a goroutine?

Answer:

"A goroutine is a lightweight thread managed by the Go runtime. Key characteristics:

1. **Lightweight:** ~2KB initial stack (vs 1-2MB OS thread)
2. **Cheap creation:** Can spawn thousands easily
3. **M:N scheduling:** Many goroutines on few OS threads
4. **Cooperative:** Yield at function calls, channel ops, blocking syscalls

Created with `go` keyword: `go functionName()`

Example: OS threads cost ~1-2MB each (can't create 10,000). Goroutines cost ~2KB (easily create millions)."

Follow-up: How does the Go scheduler work?

"Go uses M:N scheduler (M goroutines on N OS threads). Three components:

- **G (Goroutine):** Execution context
- **M (Machine):** OS thread
- **P (Processor):** Scheduling context (typically NumCPU)

Work-stealing: If P's runqueue empty, steals from other Ps. Global queue for fairness. Goroutine preemption at function calls (prevents CPU hogging)."

Q2: What's the difference between concurrency and parallelism?

Answer:

"Concurrency is *dealing with* many things at once. Parallelism is *doing* many things at once.

- **Concurrency:** Structure (multiple tasks making progress, may interleave)
- **Parallelism:** Execution (multiple tasks running simultaneously on different cores)

Example: Single-core CPU can be concurrent (task switching) but not parallel. Multi-core can be both.

Go provides concurrency primitives (goroutines, channels). Parallelism depends on GOMAXPROCS and available cores."

Q3: What is a channel? When to use buffered vs unbuffered?

Answer:

"Channel is a typed conduit for communication between goroutines.

Unbuffered (synchronous):

```
ch := make(chan int)
ch <- 42  // Blocks until receiver ready
```

Use when: Need synchronization, handoff pattern, guarantee delivery.

Buffered (asynchronous):

```
ch := make(chan int, 100)
ch <- 42  // Blocks only if buffer full
```

Use when: Decouple sender/receiver, handle bursts, producer-consumer queue.

Key difference: Unbuffered = rendezvous point. Buffered = mailbox."

Q4: What happens if you close a channel?

Answer:

"Closing a channel signals no more values will be sent.

Effects:

- **Send on closed channel:** Panic
- **Receive from closed channel:** Returns zero value, then (value, false)
- **Close already closed:** Panic
- **Receive on nil channel:** Blocks forever
- **Send on nil channel:** Blocks forever

Pattern:

```
for val := range ch {  // Exits when channel closed
    process(val)
}
```

Only sender should close. Receivers check: `val, ok := <-ch` (ok=false if closed)."

Q5: What is a race condition?

Answer:

"Race condition occurs when multiple goroutines access shared data concurrently, and at least one modifies it, without proper synchronization.

Example:

```
var counter int
for i := 0; i < 1000; i++ {
    go func() {
        counter++  // RACE! Read-modify-write not atomic
    }()
}
```

Detection: `go test -race`

Fixes:

1. Mutex: `mu.Lock(); counter++; mu.Unlock()`
2. Atomic: `atomic.AddInt64(&counter, 1)`

3. Channel: Pass messages instead of sharing memory"

Intermediate Questions

Q6: Explain the difference between sync.Mutex and sync.RWMutex.

Answer:

"sync.Mutex: Exclusive lock (one goroutine at a time)

```
mu.Lock()    // Exclusive
defer mu.Unlock()
```

sync.RWMutex: Allows multiple readers or single writer

```
mu.RLock()   // Multiple readers OK
defer mu.RUnlock()

mu.Lock()    // Exclusive write
defer mu.Unlock()
```

Use RWMutex when reads >> writes (>80% reads). Write is more expensive than Mutex.

Performance: Read ~15ns, Write ~110ns (vs Mutex ~20-30ns both)."

Q7: What is a deadlock? How to detect and prevent?

Answer:

"Deadlock: Goroutines blocked waiting for each other, unable to proceed.

Example:

```
mu1.Lock(); mu2.Lock()  // Goroutine 1
mu2.Lock(); mu1.Lock()  // Goroutine 2 → Deadlock!
```

Detection:

- Go runtime: 'fatal error: all goroutines are asleep - deadlock!'
- Tool: `go-deadlock` library with timeouts
- Stack traces: `kill -QUIT <pid>` or pprof

Prevention:

1. Lock ordering (always acquire in same order)
2. Timeout-based acquisition
3. Avoid nesting (don't call functions while holding lock)
4. Use channels (Go detects channel deadlocks)"

Q8: What happens when you send to a closed channel?

Answer:

"Panic: `panic: send on closed channel`

Why dangerous in producer-consumer:

```
// Producer
for _, item := range items {
    ch <- item         // Race: What if consumer closes ch?
}
close(ch)

// Consumer
for item := range ch {
    if done() {
        close(ch)      // Consumer shouldn't close!
        return
    }
}
```

Rule: Only sender should close channel.

Safe pattern:

```
// Producer owns channel lifecycle
done := make(chan struct{})
go producer(ch, done)

// Consumer signals done
done <- struct{}{}
```"

Q9: How do you prevent goroutine leaks?

Answer:

"Goroutine leak: Goroutine never exits, accumulates over time.

Common causes:
1. Blocking channel send (no receiver)
2. Blocking channel receive (no sender)
3. Infinite loop without exit
4. Context not checked

Prevention:
```go
// GOOD: Always provide exit path
func worker(ctx context.Context, tasks <-chan Task) {
    for {
        select {
        case task := <-tasks:
            process(task)
        case <-ctx.Done():
            return  // Exit on cancellation
        }
```

```
    }
}
```

Detection:

- `runtime.NumGoroutine()` (monitor over time)
- `goleak` library in tests
- pprof goroutine profile"

Q10: Explain context.Context and its use cases.

Answer:

"context.Context carries cancellation signals, deadlines, and request-scoped values.

Methods:

- `Done() <-chan struct{}` : Closed when context cancelled
- `Err() error` : Why cancelled (Canceled or DeadlineExceeded)
- `Deadline() (time.Time, bool)` : Expiration time
- `Value(key) interface{}` : Request-scoped data

Use cases:

1. HTTP request cancellation (client disconnects)
2. Database query timeouts
3. Graceful shutdown (cancel tree of goroutines)
4. Trace IDs, user IDs across call stack

Creation:

```
ctx, cancel := context.WithCancel(parent)
ctx, cancel := context.WithTimeout(parent, 5*time.Second)
ctx, cancel := context.WithDeadline(parent, time.Now().Add(5*time.Second))
```

Best practices:

- Pass as first parameter
- Always defer cancel()
- Never store in struct"

Advanced Questions

Q11: How does select work with multiple channels?

Answer:

"select chooses one ready case (non-deterministic if multiple ready).

```
select {
case msg := <-ch1:
    // Chosen if ch1 has data
case ch2 <- msg:
    // Chosen if ch2 has space
case <-time.After(time.Second):
```

```
    // Timeout
default:
    // Non-blocking (chosen if others not ready)
}
```

Key behaviors:

- Random selection if multiple ready (prevents starvation)
- Blocks if no case ready (unless default)
- `default` makes non-blocking
- All cases evaluated each iteration

Use cases:

- Timeout pattern
- Non-blocking send/receive
- Cancellation with context
- Fan-in (merge channels)"

Q12: What is the memory model? What is 'happens-before'?

Answer:

"Go memory model defines when one goroutine's write is guaranteed visible to another's read.

Happens-before relationship guarantees visibility:

Guaranteed:

- Channel send happens-before corresponding receive
- Channel close happens-before receive of zero value
- Mutex.Lock() happens-before subsequent Lock() calls
- WaitGroup.Done() happens-before Wait() returns

Example (safe):

```
var a string
var done = make(chan struct{})

// Goroutine 1
a = \"hello\"
close(done)

// Goroutine 2
<-done
print(a)  // Guaranteed to see \"hello\"
```

Not guaranteed (race):

```
var a, b int
go func() { a = 1; b = 2 }()
go func() { print(b, a) }()  // May print 2, 0 (b visible but not a)
```

Must use synchronization (channels, mutexes) for visibility."

Q13: Explain worker pool pattern and when to use it.

Answer:

"Worker pool: Fixed number of goroutines processing tasks from shared queue.

Implementation:

```
type Pool struct {
    tasks   chan Task
    results chan Result
    workers int
    wg      sync.WaitGroup
}

func (p *Pool) worker() {
    defer p.wg.Done()
    for task := range p.tasks {
        p.results <- task.Execute()
    }
}

func NewPool(n int) *Pool {
    p := &Pool{
        tasks:   make(chan Task, n*2),
        results: make(chan Result, n*2),
    }
    for i := 0; i < n; i++ {
        p.wg.Add(1)
        go p.worker()
    }
    return p
}
```

When to use:

- Limit concurrency (prevent unbounded goroutines)
- CPU-bound work (workers = NumCPU)
- Resource-bound (database connections, API rate limits)

Worker count:

- CPU-bound: NumCPU
- I/O-bound: NumCPU × 10-100
- Benchmark to find optimal"

Q14: How do you implement a rate limiter in Go?

Answer:

"Common approaches:

1. Token Bucket (golang.org/x/time/rate):

```
limiter := rate.NewLimiter(10, 20)  // 10 req/s, burst 20
if limiter.Allow() {
    // Process request
} else {
    // Reject
}
```

2. Fixed Window:

```
type FixedWindow struct {
    mu      sync.Mutex
    count   int
    limit   int
    window  time.Time
}

func (fw *FixedWindow) Allow() bool {
    fw.mu.Lock()
    defer fw.mu.Unlock()

    now := time.Now()
    if now.Sub(fw.window) >= time.Second {
        fw.count = 0
        fw.window = now
    }

    if fw.count < fw.limit {
        fw.count++
        return true
    }
    return false
}
```

3. Semaphore (limit concurrent):

```
sem := make(chan struct{}, 10)
sem <- struct{}{}        // Acquire
defer func() { <-sem }() // Release
```

Choose based on needs: Token bucket (smooth), Fixed window (simple), Semaphore (concurrency limit)."

Q15: What is the difference between buffered channel and sync.Cond?

Answer:

"Both coordinate goroutines but different use cases:

Buffered channel:

- Send/receive values
- FIFO queue

- Decouples producer/consumer

```
ch := make(chan Task, 100)
ch <- task  // Non-blocking if space
```

sync.Cond:

- Signal without value
- Broadcast to multiple waiters
- Must hold lock while checking condition

```
cond.L.Lock()
for !condition {
    cond.Wait()  // Releases lock, waits, reacquires
}
cond.L.Unlock()
```

Prefer channels (more idiomatic). Use Cond only when:

- Need to broadcast to many waiters
- Complex condition under same lock
- Porting Java code (wait/notify)"

System Design Questions

Q16: Design a concurrent cache with expiration.

Answer:

"Requirements: Thread-safe, LRU eviction, TTL expiration, high read throughput.

Design:

```
type Cache struct {
    shards []*Shard  // Reduce contention
    ttl    time.Duration
}

type Shard struct {
    mu      sync.RWMutex
    data    map[string]*Entry
    lru     *list.List
}

type Entry struct {
    key     string
    value   interface{}
    expires time.Time
    element *list.Element
}

func (c *Cache) Get(key string) (interface{}, bool) {
```

```
    shard := c.getShard(key)
    shard.mu.RLock()
    defer shard.mu.RUnlock()

    entry, ok := shard.data[key]
    if !ok || time.Now().After(entry.expires) {
        return nil, false
    }

    // LRU: Move to front
    shard.lru.MoveToFront(entry.element)

    return entry.value, true
}
```

Key decisions:

- Sharded RWMutex (16 shards = 1/16 contention)
- Read-heavy → RWMutex
- Background cleanup goroutine
- LRU for size limit"

Q17: How would you implement graceful shutdown?

Answer:

"Graceful shutdown: Stop accepting new work, finish in-flight, clean up resources.

Pattern:

```
func (s *Server) Shutdown(ctx context.Context) error {
    // 1. Stop accepting new work
    close(s.quit)

    // 2. Wait for in-flight to complete
    done := make(chan struct{})
    go func() {
        s.wg.Wait()
        close(done)
    }()

    // 3. Timeout
    select {
    case <-done:
        return nil
    case <-ctx.Done():
        return ctx.Err()
    }
}

// HTTP server
func main() {
    server := &http.Server{}
```

```
    // Signal handling
    sigCh := make(chan os.Signal, 1)
    signal.Notify(sigCh, os.Interrupt, syscall.SIGTERM)

    go server.ListenAndServe()

    <-sigCh

    // Graceful shutdown with timeout
    ctx, cancel := context.WithTimeout(context.Background(), 30*time.Second)
    defer cancel()

    server.Shutdown(ctx)
}
```

Order:

1. HTTP server (stop accepting requests)
2. Worker pools (drain tasks)
3. Cache (flush if persistent)
4. Database connections (close pool)"

Q18: Design a concurrent web crawler.

Answer:

"Requirements: Crawl websites, respect politeness, avoid duplicates, bounded concurrency.

Architecture:

```
type Crawler struct {
    seen         sync.Map  // Visited URLs
    fetchWorkers int
    parseWorkers int

    urlQueue  chan string
    htmlQueue chan HTML
    results   chan Result
}

func NewCrawler(fetch, parse int) *Crawler {
    c := &Crawler{
        fetchWorkers: fetch,
        parseWorkers: parse,
        urlQueue:     make(chan string, 100),
        htmlQueue:    make(chan HTML, 100),
        results:      make(chan Result, 100),
    }

    // Start workers
    for i := 0; i < fetch; i++ {
        go c.fetcher()  // I/O-bound
```

```
    }
    for i := 0; i < parse; i++ {
        go c.parser()   // CPU-bound
    }

    return c
}

func (c *Crawler) fetcher() {
    for url := range c.urlQueue {
        if _, seen := c.seen.LoadOrStore(url, true); seen {
            continue
        }

        // Rate limiting per domain
        c.respectPoliteness(url)

        html := fetch(url)
        c.htmlQueue <- html
    }
}
```

Key components:

- sync.Map for visited check (concurrent-safe)
- Separate fetch (I/O) and parse (CPU) workers
- Rate limiter per domain (politeness)
- robots.txt respect
- Bounded concurrency (worker pools)"

Tricky Questions

Q19: What's wrong with this code?

```
var wg sync.WaitGroup
for i := 0; i < 10; i++ {
    wg.Add(1)
    go func() {
        defer wg.Done()
        fmt.Println(i)
    }()
}
wg.Wait()
```

Answer:

"Closure captures loop variable `i` by reference. All goroutines see final value (10).

Output: Prints 10 ten times (or random values if loop finishes before goroutines).

Fix 1: Pass as parameter

```
go func(i int) {
    defer wg.Done()
    fmt.Println(i)
}(i)
```

Fix 2: Shadow variable

```
i := i  // Create new variable
go func() {
    defer wg.Done()
    fmt.Println(i)
}()
```

Common interview trap - tests understanding of closures and goroutine scheduling."

Q20: Why does this deadlock?

```
ch := make(chan int)
ch <- 42
val := <-ch
```

Answer:

"Unbuffered channel blocks on send until receiver ready. Single goroutine deadlocks waiting for itself.

Fix 1: Buffered channel

```
ch := make(chan int, 1)  // Buffer size 1
ch <- 42                 // Non-blocking
val := <-ch
```

Fix 2: Goroutine

```
go func() { ch <- 42 }()
val := <-ch
```

Tip: Unbuffered channel requires goroutines on both sides."

Key Takeaways

1. **Goroutine: Lightweight thread (~2KB)**
2. **Channel: Communication, not just data structure**
3. **Buffered = async, Unbuffered = sync**
4. **Race detector is your friend: -race**
5. **Mutex protects data, channel communicates**
6. **RWMutex only if read-heavy (>80%)**
7. **Always provide goroutine exit path (context)**
8. **Deadlock = circular wait**
9. **Worker pool for bounded concurrency**

10. **Test concurrently: stress + race detector**

Interview Preparation Checklist

- [ ] Understand goroutines vs threads
- [ ] Know buffered vs unbuffered channels
- [ ] Explain select statement
- [ ] Understand race conditions
- [ ] Know mutex vs RWMutex trade-offs
- [ ] Explain deadlock causes and prevention
- [ ] Understand context.Context
- [ ] Implement worker pool pattern
- [ ] Design concurrent cache
- [ ] Implement graceful shutdown
- [ ] Know common gotchas (closure, channel close)
- [ ] Practice on whiteboard (no IDE)

Next: trick-questions.md - Gotchas and misleading questions.