Select Statement

Definition (Precise)

The `select` statement lets a goroutine wait on multiple channel operations simultaneously. It blocks until one of its cases can proceed, then executes that case. If multiple cases are ready, it chooses one **at random**.

Think of it as: `switch` for channels with blocking behavior.

Syntax

```
select {
case val := <-ch1:
    // Received from ch1
case ch2 <- val:
    // Sent to ch2
case val, ok := <-ch3:
    // Received from ch3 with closed-check
default:
    // No case ready (non-blocking)
}
```

Basic Examples

Example 1: Wait on Multiple Channels

```
ch1 := make(chan string)
ch2 := make(chan string)

go func() {
    time.Sleep(100 * time.Millisecond)
    ch1 <- "from ch1"
}()

go func() {
    time.Sleep(200 * time.Millisecond)
    ch2 <- "from ch2"
}()

select {
case msg1 := <-ch1:
    fmt.Println(msg1)  // "from ch1" (arrives first)
case msg2 := <-ch2:
    fmt.Println(msg2)
}
```

Behavior: Blocks until **at least one** case is ready. Executes the first ready case.

Example 2: Non-Blocking Send

```
ch := make(chan int, 1)
ch <- 1  // Buffer full

select {
case ch <- 2:
    fmt.Println("Sent")
default:
    fmt.Println("Channel full, skipping")  // Executes
}
```

Behavior: `default` makes select non-blocking. If no case ready, `default` executes immediately.

Example 3: Non-Blocking Receive

```
ch := make(chan int)

select {
case val := <-ch:
    fmt.Println("Received:", val)
default:
    fmt.Println("No value available")  // Executes
}
```

Select Semantics

Rule 1: Random Selection on Multiple Ready

```
ch := make(chan int, 2)
ch <- 1
ch <- 2

for i := 0; i < 4; i++ {
    select {
    case val := <-ch:
        fmt.Println("Received:", val)
    case val := <-ch:
        fmt.Println("Also received:", val)
    }
}
// Output (random):
// Received: 1
// Also received: 2
// (or vice versa, or mixed)
```

Why random? Prevents starvation. If deterministic, one case could always win.

Rule 2: Nil Channels Are Ignored

```
var ch chan int  // nil channel

select {
case <-ch:
    // Never selected (nil channel blocks forever)
case <-time.After(time.Second):
    fmt.Println("Timeout")  // This executes
}
```

Use case: Dynamically enable/disable select cases.

```
var stopCh chan struct{}

switch mode {
case "interruptible":
    stopCh = make(chan struct{})
case "non-interruptible":
    stopCh = nil  // Disables this case
}

select {
case <-workCh:
    // Do work
case <-stopCh:  // Only active if stopCh != nil
    // Stop
}
```

Rule 3: Default Prevents Blocking

```
select {
case <-ch:
    // Receive if ready
default:
    // Execute immediately if ch not ready
}
```

Warning: Tight loop with default can consume CPU.

```
// BAD: Busy-wait loop
for {
    select {
    case msg := <-ch:
        process(msg)
    default:
        // Spins at 100% CPU
    }
}
```

```
// BETTER: Block when no work
for {
    select {
    case msg := <-ch:
        process(msg)
    // No default → blocks
    }
}
```

Common Patterns

Pattern 1: Timeout

```
select {
case result := <-resultCh:
    fmt.Println("Got result:", result)
case <-time.After(5 * time.Second):
    fmt.Println("Timeout")
}
```

Warning: `time.After` creates a timer that isn't stopped. For loops, use `time.NewTimer`:

```
// BAD: Leaks timers
for {
    select {
    case msg := <-ch:
        process(msg)
    case <-time.After(time.Second):  // Creates new timer every iteration!
        timeout()
    }
}

// GOOD: Reusable timer
timer := time.NewTimer(time.Second)
defer timer.Stop()

for {
    select {
    case msg := <-ch:
        process(msg)
        if !timer.Stop() {
            <-timer.C
        }
        timer.Reset(time.Second)
    case <-timer.C:
        timeout()
        timer.Reset(time.Second)
```

```
    }
}
```

Pattern 2: Context Cancellation

```
func worker(ctx context.Context, jobs <-chan Job) {
    for {
        select {
        case job := <-jobs:
            process(job)
        case <-ctx.Done():
            fmt.Println("Context cancelled, exiting")
            return
        }
    }
}
```

Critical: Always include `ctx.Done()` case in long-running goroutines.

Pattern 3: Done Channel

```
done := make(chan struct{})

go func() {
    // Long-running work
    for {
        select {
        case <-done:
            return
        default:
            doWork()
        }
    }
}()

// Later: signal completion
close(done)
```

Better alternative: Use context (more composable).

Pattern 4: Fan-In (Multiplexing)

```
func fanIn(ch1, ch2 <-chan int) <-chan int {
    out := make(chan int)

    go func() {
        defer close(out)
        for {
            select {
```

```
            case val, ok := <-ch1:
                if !ok {
                    ch1 = nil  // Disable this case
                    continue
                }
                out <- val
            case val, ok := <-ch2:
                if !ok {
                    ch2 = nil  // Disable this case
                    continue
                }
                out <- val
            }

            if ch1 == nil && ch2 == nil {
                return  // Both closed
            }
        }
    }()

    return out
}
```

Technique: Set finished channels to `nil` to disable their cases.

Pattern 5: Priority Select (Workaround)

```
// Prefer high-priority channel over low-priority
for {
    select {
    case msg := <-highPriorityCh:
        process(msg)
    default:
        select {
        case msg := <-highPriorityCh:
            process(msg)
        case msg := <-lowPriorityCh:
            process(msg)
        }
    }
}
```

Why nested? Inner `default` makes outer select non-blocking, allowing another check of high-priority channel.

Warning: This is a workaround. Go's select is intentionally fair (random).

Pattern 6: Dropping Old Values (Latest-Only)

```
type Monitor struct {
    updates chan State
}

func (m *Monitor) UpdateState(state State) {
    select {
    case m.updates <- state:
        // Sent successfully
    default:
        // Channel full, drop old value
        <-m.updates            // Remove old
        m.updates <- state     // Send new (guaranteed to succeed)
    }
}
```

Use case: When only latest value matters (e.g., UI updates, monitoring).

Common Bugs

Bug 1: Select in Loop Without Exit

```
// WRONG: No way to exit
func worker(jobs <-chan Job) {
    for {
        select {
        case job := <-jobs:
            process(job)
        }
    }
    // Goroutine never exits → leak
}

// CORRECT: Add done channel
func worker(jobs <-chan Job, done <-chan struct{}) {
    for {
        select {
        case job := <-jobs:
            process(job)
        case <-done:
            return
        }
    }
}
```

Bug 2: time.After in Loop

```
// WRONG: Leaks timers
for {
    select {
```

```
    case <-ch:
        process()
    case <-time.After(time.Second):  // New timer every iteration!
        timeout()
    }
}

// After 1 hour: 3600 leaked timers
```

Impact: Memory leak + garbage collection pressure.

Fix: Use `time.NewTimer` (see Pattern 1).

Bug 3: Nil Channel Confusion

```
var ch chan int  // nil

select {
case ch <- 1:    // Never executes (send to nil blocks)
case <-ch:       // Never executes (receive from nil blocks)
default:
    fmt.Println("Always executes")
}
```

When this bites: Dynamic channel assignment.

```
var stopCh chan struct{}
if condition {
    stopCh = make(chan struct{})
}

// Later...
select {
case <-stopCh:  // Ignored if stopCh is nil
case <-workCh:
}
```

Bug 4: Forgetting Default in Non-Blocking Attempt

```
// WRONG: Blocks forever if ch not ready
val := <-ch  // Blocking receive

// CORRECT: Non-blocking attempt
select {
case val := <-ch:
    fmt.Println("Received:", val)
default:
    fmt.Println("Not ready")
}
```

Bug 5: Race on Shared State in Select

```
// WRONG
var counter int

select {
case <-ch1:
    counter++  // Race if other goroutines access counter
case <-ch2:
    counter++
}

// CORRECT: Protect with mutex or use atomic
select {
case <-ch1:
    atomic.AddInt64(&counter, 1)
case <-ch2:
    atomic.AddInt64(&counter, 1)
}
```

Performance Characteristics

Operation	Time	Notes
Select with 2 cases	~100-150 ns	Overhead vs direct channel op
Select with 10 cases	~200-300 ns	Linear in number of cases
Select with default	~50-100 ns	Fast path for non-blocking

Compared to:

- Direct channel op: ~50-100 ns
- Mutex lock/unlock: ~20-30 ns

Takeaway: Select adds ~2-3x overhead vs direct channel op. Not free, but not expensive.

Select vs Other Patterns

Select vs Mutex

Use select when:

- Coordinating goroutines via channels
- Need timeout or cancellation
- Multiplexing multiple sources

Use mutex when:

- Protecting shared memory
- Critical section is short
- No need for timeout/cancellation

```
// Select: Coordination
select {
case msg := <-ch:
    process(msg)
case <-ctx.Done():
    return
}

// Mutex: Shared state
mu.Lock()
counter++
mu.Unlock()
```

Select vs WaitGroup

Select: Wait for **one of many** operations.
WaitGroup: Wait for **all** operations.

```
// Select: Process whichever arrives first
select {
case result1 := <-ch1:
    return result1
case result2 := <-ch2:
    return result2
}

// WaitGroup: Wait for both
var wg sync.WaitGroup
wg.Add(2)
go func() { defer wg.Done(); fetch1() }()
go func() { defer wg.Done(); fetch2() }()
wg.Wait()
```

Real-World Failure: Timeout Leak

Company: Video streaming service (2018)

What happened:
Memory usage grew 10GB/hour, causing OOM crashes every 6 hours.

Root cause:

```
func processStream(stream Stream) {
    for frame := range stream.Frames() {
        select {
        case resultCh <- processFrame(frame):
        case <-time.After(100 * time.Millisecond):  // LEAK
            log.Println("Timeout processing frame")
        }
```

```
    }
}
```

At 30 FPS for 6 hours:

- 648,000 timers created
- Each timer: ~250 bytes
- Total leaked: ~162MB per stream
- 100 concurrent streams: ~16GB leaked

Fix:

```
func processStream(stream Stream) {
    timer := time.NewTimer(100 * time.Millisecond)
    defer timer.Stop()

    for frame := range stream.Frames() {
        timer.Reset(100 * time.Millisecond)

        select {
        case resultCh <- processFrame(frame):
        case <-timer.C:
            log.Println("Timeout processing frame")
        }
    }
}
```

Lessons:

1. `time.After` in loops is almost always wrong
2. Use `time.NewTimer` with `Reset` for repeated timeouts
3. Monitor goroutine and timer count in production

Interview Traps

Trap 1: "Select is like switch"

Superficial similarity. Key differences:

Correct answer:
"Select has switch-like syntax but fundamentally different semantics: 1) It blocks until a case is ready (unless default), 2) It chooses randomly among multiple ready cases, 3) Cases are channel operations only, 4) Nil channels disable cases. Switch evaluates top-to-bottom, doesn't block, and works with any comparable type."

Trap 2: "This select always executes in order"

```
select {
case val := <-ch1:
    // Case 1
case val := <-ch2:
```

```
    // Case 2
}
```

Wrong. If both ready, Go picks randomly.

Correct answer:
"Select chooses randomly among ready cases to prevent starvation. If both channels have data, either case could execute. This is intentional design—order would cause fairness issues."

Trap 3: "Default makes select asynchronous"

Confused terminology.

Correct answer:
"Default makes select non-blocking, not asynchronous. If no case is ready, default executes immediately instead of blocking. The operations themselves (send/receive) are still synchronous channel operations."

Trap 4: "This timeout will always fire after 1 second"

```
for {
    select {
    case <-ch:
        // Work (takes 2 seconds)
        time.Sleep(2 * time.Second)
    case <-time.After(1 * time.Second):
        // Timeout
    }
}
```

Wrong. Timeout starts **each iteration**, not overall.

Correct answer:
"The timeout is per select statement, not per case or overall. If the work takes 2 seconds, a new 1-second timeout starts for the next iteration. To implement a global timeout, use context.WithTimeout or a timer outside the loop."

Key Takeaways

1. **Select blocks until one case ready** (unless default)
2. **Random selection among ready cases** (fairness)
3. **Nil channels are ignored** (dynamic case enabling/disabling)
4. **Default prevents blocking** (non-blocking operations)
5. **time.After in loops leaks** (use time.NewTimer)
6. **Always include ctx.Done()** in long-running selects
7. **Select adds ~2-3x overhead** vs direct channel operation
8. **No priority or ordering** (intentional design)

What You Should Be Thinking Now

- "How do I protect shared state accessed by multiple goroutines?"
- "When should I use channels vs mutexes?"
- "What's the difference between sync.Mutex and sync.RWMutex?"

- "How do I safely update counters without channels?"

Next: mutex.md - Protecting shared memory with mutexes.

Exercises (Do These Before Moving On)

1. Write a program that uses select to wait on 3 channels. Send to them at different times and observe random selection when multiple are ready.
2. Create a goroutine leak with `time.After` in a loop. Fix it with `time.NewTimer`.
3. Implement a worker that can be cancelled via context using select.
4. Write a fan-in function that merges 5 channels, handling individual channel closure correctly.
5. Implement a non-blocking send and a non-blocking receive using select with default.

Don't continue until you can explain: "Why does Go's select choose randomly instead of first-ready or round-robin?"