

## sync.WaitGroup

### Definition (Precise)

A **WaitGroup** is a synchronization primitive that waits for a collection of goroutines to finish. It's a counter-based barrier: increment when starting work, decrement when finishing, wait until counter reaches zero.

**Purpose:** Answer the question: "Have all my goroutines finished?"

## Syntax

```
import "sync"

var wg sync.WaitGroup

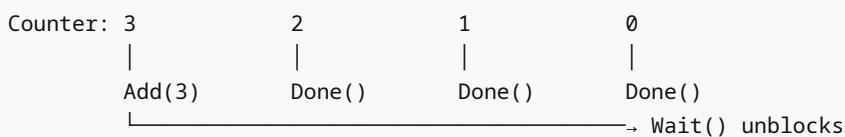
wg.Add(1)          // Increment counter (before starting goroutine)
go func() {
    defer wg.Done() // Decrement counter (Done() = Add(-1))
    // Work
}()

wg.Wait()          // Block until counter == 0
```

## Mental Model

Think of WaitGroup as a **countdown latch**:

- You tell it how many tasks to wait for ( Add )
- Each task signals completion ( Done )
- Wait blocks until all tasks complete (counter  $\rightarrow$  0)



## Typical Usage Pattern

```
func processItems(items []Item) {
    var wg sync.WaitGroup

    for _, item := range items {
        wg.Add(1) // Increment for each goroutine

        go func(it Item) {
            defer wg.Done() // Decrement when done
            process(it)
        }(item)
    }
}
```

```

    }

    wg.Wait() // Block until all goroutines complete
    fmt.Println("All items processed")
}

```

## Correct Usage Patterns

### Pattern 1: Add Before Goroutine Launch

```

// CORRECT
for i := 0; i < 10; i++ {
    wg.Add(1) // Add BEFORE go keyword
    go func() {
        defer wg.Done()
        doWork()
    }()
}
wg.Wait()

```

#### Why Add before go ?

- Ensures counter incremented before goroutine might call `Done()`
- Prevents race: `Wait()` unblocking before all goroutines started

### Pattern 2: Done with Defer (Safest)

```

wg.Add(1)
go func() {
    defer wg.Done() // Guaranteed even if panic
    // Work that might panic
}()

```

#### Why defer?

- Ensures `Done()` called even if panic
- Prevents counter stuck above 0 (deadlock)

### Pattern 3: Dynamic Work

```

func process(items []Item) {
    var wg sync.WaitGroup
    results := make(chan Result, len(items))

    for _, item := range items {
        wg.Add(1)
        go func(it Item) {
            defer wg.Done()
            results <- processItem(it)
        }(item)
    }
}

```

```

    }

    // Close results when all goroutines done
    go func() {
        wg.Wait()
        close(results)
    }()

    // Consume results
    for result := range results {
        handleResult(result)
    }
}

```

## Pattern 4: Worker Pool with WaitGroup

```

func workerPool(jobs <-chan Job, workers int) {
    var wg sync.WaitGroup

    for i := 0; i < workers; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            for job := range jobs { // Range exits when jobs closed
                process(job)
            }
        }()
    }

    wg.Wait() // All workers finished
}

```

## Common Bugs

### Bug 1: Add Inside Goroutine

```

// WRONG
for i := 0; i < 10; i++ {
    go func() {
        wg.Add(1) // RACE: Add() inside goroutine
        defer wg.Done()
        doWork()
    }()
}
wg.Wait() // Might return before all Add() calls

```

**Problem:** `Wait()` might execute before all `Add()` calls, seeing `counter==0` prematurely.

**Fix:** Always `Add()` before `go` .

## Bug 2: Forgetting Done()

```
// WRONG
wg.Add(1)
go func() {
    doWork()
    // Forgot wg.Done()
}()

wg.Wait() // Blocks forever (counter never reaches 0)
```

**Fix:** Always `defer wg.Done()` .

## Bug 3: Negative Counter (Add/Done Mismatch)

```
// WRONG
wg.Add(1)
wg.Done()
wg.Done() // PANIC: negative WaitGroup counter
```

**Error:**

```
panic: sync: negative WaitGroup counter
```

**Cause:** More `Done()` calls than `Add()` .

## Bug 4: Reusing WaitGroup Without Waiting

```
// WRONG
var wg sync.WaitGroup

func batch1() {
    wg.Add(1)
    go func() { defer wg.Done(); /*...*/ }()
    // Forgot to wait!
}

func batch2() {
    wg.Add(1) // Reusing wg without waiting for batch1
    go func() { defer wg.Done(); /*...*/ }()
    wg.Wait()
}
```

**Problem:** Counter from batch1 leaks into batch2.

**Fix:** Always `Wait()` before reusing, or use new `WaitGroup`.

## Bug 5: Copying WaitGroup

```
// WRONG
func doWork(wg sync.WaitGroup) { // Passed by value (copied)
    defer wg.Done() // Decrements COPY, not original
    // Work
}

func main() {
    var wg sync.WaitGroup
    wg.Add(1)
    go doWork(wg) // Copies wg
    wg.Wait()    // Waits on original, never decremented
}
```

**Error:** `Wait()` blocks forever.

**Fix:** Always pass `*sync.WaitGroup` (pointer).

```
func doWork(wg *sync.WaitGroup) {
    defer wg.Done()
    // Work
}

func main() {
    var wg sync.WaitGroup
    wg.Add(1)
    go doWork(&wg) // Pass pointer
    wg.Wait()
}
```

**Detection:**

```
$ go vet
./main.go:10:9: doWork passes lock by value: sync.WaitGroup contains sync.noCopy
```

## WaitGroup vs Channels

Aspect	WaitGroup	Channel
<b>Purpose</b>	Wait for completion	Communication + synchronization
<b>Returns data</b>	No	Yes
<b>Partial waits</b>	No (all or nothing)	Yes (read first N)
<b>Cancellation</b>	No (must wait)	Yes (close channel)
<b>Errors</b>	No error mechanism	Can send errors via channel
<b>Best for</b>	"Wait for all workers"	"Process results as they arrive"

## Example: When to Use Each

```
// WaitGroup: Fire and forget, just wait for completion
func parallelWrite(items []Item) {
    var wg sync.WaitGroup
    for _, item := range items {
        wg.Add(1)
        go func(it Item) {
            defer wg.Done()
            writeToDatabase(it) // Fire and forget
        }(item)
    }
    wg.Wait() // All writes done
}

// Channel: Need results
func parallelFetch(urls []string) []Result {
    results := make(chan Result, len(urls))

    for _, url := range urls {
        go func(u string) {
            results <- fetch(u) // Send result
        }(url)
    }

    // Collect results
    collected := make([]Result, 0, len(urls))
    for i := 0; i < len(urls); i++ {
        collected = append(collected, <-results)
    }

    return collected
}

// Both: Wait for completion + close results channel
func parallelProcess(items []Item) <-chan Result {
    var wg sync.WaitGroup
    results := make(chan Result, len(items))

    for _, item := range items {
        wg.Add(1)
        go func(it Item) {
            defer wg.Done()
            results <- process(it)
        }(item)
    }

    go func() {
        wg.Wait() // Wait for all
        close(results) // Then close
    }()
}
```

```
    return results
}
```

## Performance Characteristics

Operation	Time	Notes
Add(1)	~10-20 ns	Atomic increment
Done()	~10-20 ns	Atomic decrement
Wait() (already 0)	~5-10 ns	Fast path
Wait() (blocking)	~500-1000 ns	Scheduler involvement

**Overhead:** Minimal. Use freely.

## Advanced Pattern: Bounded Goroutines with WaitGroup

```
func processBounded(items []Item, maxConcurrency int) {
    var wg sync.WaitGroup
    semaphore := make(chan struct{}, maxConcurrency)

    for _, item := range items {
        wg.Add(1)

        go func(it Item) {
            defer wg.Done()

            semaphore <- struct{}{} // Acquire slot
            defer func() { <-semaphore }() // Release slot

            process(it)
        }(item)
    }

    wg.Wait()
}
```

**Combines:**

- WaitGroup: Wait for all to complete
- Semaphore channel: Limit concurrency

## WaitGroup Internals (Conceptual)

```
type WaitGroup struct {
    state atomic.Uint64 // High 32 bits: counter, Low 32 bits: waiter count
    sema  uint32              // Semaphore for waiting goroutines
}
```

```

}

func (wg *WaitGroup) Add(delta int) {
    atomic.AddUint64(&wg.state, uint64(delta)<<32)
    // If counter == 0, wake waiters
}

func (wg *WaitGroup) Done() {
    wg.Add(-1)
}

func (wg *WaitGroup) Wait() {
    // Increment waiter count
    // If counter > 0, block on semaphore
    // Else return immediately
}

```

**Key insight:** WaitGroup uses atomic operations internally (lock-free).

## Real-World Failure: Goroutine Leak from Unclosed Channel

**Company:** E-commerce platform (2020)

### What happened:

Memory usage grew from 1GB to 20GB over 3 days. OOM crashes.

### Root cause:

```

func processOrders(orders []Order) {
    var wg sync.WaitGroup
    results := make(chan Result) // Unbuffered!

    for _, order := range orders {
        wg.Add(1)
        go func(o Order) {
            defer wg.Done()
            results <- processOrder(o) // Blocks if no receiver
        }(order)
    }

    wg.Wait() // Waits, but results not consumed
    // results channel never closed
    // Goroutines block forever on results <-
}

```

### Problem:

- Goroutines block sending to `results`
- `wg.Wait()` completes (Done called), but goroutines still blocked
- Goroutine leak: memory grows unbounded

### Fix 1: Buffer results channel

```
results := make(chan Result, len(orders)) // Buffer
```

## Fix 2: Consume results

```
go func() {  
    wg.Wait()  
    close(results)  
}()  
  
for result := range results {  
    handleResult(result)  
}
```

### Lessons:

1. WaitGroup only tracks completion, not goroutine lifecycle
2. Goroutines can Complete but still be blocked (channel send)
3. Always ensure goroutines can exit (consume channels, close channels)

## Interview Traps

### Trap 1: "WaitGroup waits for goroutines to exit"

**Imprecise.** WaitGroup waits for `Done()` calls, not goroutine exit.

#### Correct answer:

"WaitGroup waits until its counter reaches zero, which happens when `Done()` is called for each `Add()`. This typically corresponds to goroutine completion if `Done()` is deferred, but a goroutine could call `Done()` and continue running, or block after calling `Done()`. WaitGroup tracks logical completion, not goroutine lifecycle."

### Trap 2: "I need WaitGroup to return results from goroutines"

**Wrong tool.** WaitGroup doesn't transfer data.

#### Correct answer:

"WaitGroup only provides a completion barrier—it doesn't collect results. To return data from goroutines, use channels. You can combine both: WaitGroup to know when all goroutines finish, and channels to collect results."

### Trap 3: "I can call Wait() multiple times"

**Technically yes, but usually wrong.**

#### Correct answer:

"You can call `Wait()` multiple times, and all will block until counter reaches zero. However, reusing a WaitGroup across multiple batches of work without waiting in between is a common bug. Typically, you create a new WaitGroup for each batch or ensure complete waiting before reuse."

### Trap 4: "This code is safe—I use WaitGroup"

```

var counter int
var wg sync.WaitGroup

for i := 0; i < 100; i++ {
    wg.Add(1)
    go func() {
        defer wg.Done()
        counter++ // DATA RACE
    }()
}
wg.Wait()

```

**Wrong.** WaitGroup prevents Wait from returning early but doesn't protect shared state.

**Correct answer:**

"WaitGroup provides synchronization for completion signaling but doesn't protect shared memory. The `counter++` operations still race. You need a mutex or atomic operations to protect `counter`, and WaitGroup separately to wait for completion."

## Key Takeaways

1. **WaitGroup = completion barrier** (wait for counter → 0)
2. **Add before goroutine launch** (prevents race)
3. **Always defer Done()** (ensures call even on panic)
4. *\*Pass WaitGroup* (pointer), never by value
5. **Don't reuse without waiting** (counter bleed-through)
6. **Doesn't protect shared state** (use mutex/atomic separately)
7. **Doesn't transfer data** (use channels for results)
8. **Can't cancel Wait()** (must wait for all, or use context+channel)

## What You Should Be Thinking Now

- "How do I coordinate goroutines waiting for a specific condition?"
- "What if I need to wake one goroutine vs all goroutines?"
- "When should I use sync.Cond?"
- "How do condition variables work in Go?"

**Next:** [cond.md](#) - Condition variables for complex signaling patterns.

## Exercises (Do These Before Moving On)

1. Process 1000 items concurrently using WaitGroup. Verify all complete.
2. Intentionally call `Done()` more times than `Add()`. Observe panic.
3. Pass WaitGroup by value instead of pointer. Run `go vet`. Fix it.
4. Create a goroutine leak: use WaitGroup but have goroutines block on unbuffered channel after `Done()`. Detect with `runtime.NumGoroutine()`.
5. Implement bounded concurrency: process 10,000 items with max 100 concurrent goroutines, using WaitGroup + semaphore channel.

Don't continue until you can explain: "Why must `Add ( )` be called before the `go` keyword, not inside the goroutine?"