sync.Cond

Definition (Precise)

A **condition variable** (Cond) is a synchronization primitive that allows goroutines to wait for or announce changes to a shared condition. It's a rendezvous point for goroutines waiting for an event.

Purpose: Efficiently wait for a condition to become true without busy-waiting (spinning).

Warning: Cond is the **most misunderstood** and **least used** sync primitive in Go. Most use cases are better solved with channels.

Syntax

```
import "sync"

var mu sync.Mutex
cond := sync.NewCond(&mu)

// Waiter: Wait for condition
mu.Lock()
for !condition {
    cond.Wait()  // Releases mu, blocks, reacquires mu when signaled
}
// Use shared state
mu.Unlock()

// Signaler: Signal condition changed
mu.Lock()
// Change shared state
condition = true
cond.Signal()    // Wake one waiter
// or cond.Broadcast()  // Wake all waiters
mu.Unlock()
```

Mental Model

Think of Cond as a **waiting room**:

- Goroutines enter the room (`Wait()`) and sleep
- Someone announces a change (`Signal()` or `Broadcast()`)
- Goroutines wake up and check if condition is met

```
Waiters: [G1]———[Sleeping]——→[Woken]—→[Check condition]
         [G2]———[Sleeping]——→[Woken]—→[Check condition]
         [G3]———[Sleeping]——→[Woken]—→[Check condition]
                      ↑
                      └—— Signal() or Broadcast()
```

Critical Rules

Rule 1: Always Wait in a Loop

```
// WRONG
mu.Lock()
if !condition {
    cond.Wait()  // Single check
}
mu.Unlock()

// CORRECT
mu.Lock()
for !condition {  // Loop!
    cond.Wait()
}
mu.Unlock()
```

Why loop?

1. **Spurious wakeups:** Waiter can wake without Signal/Broadcast
2. **Stolen wake:** Another goroutine might steal the condition
3. **Multiple conditions:** Signal might be for different predicate

Rule 2: Hold Lock When Calling Wait

```
// WRONG
cond.Wait()  // No lock held → PANIC

// CORRECT
mu.Lock()
for !condition {
    cond.Wait()
}
mu.Unlock()
```

Why? `Wait()` must release the lock atomically to avoid lost wakeups.

Rule 3: Hold Lock When Calling Signal/Broadcast

```
// WRONG
condition = true
cond.Signal()  // No lock held

// CORRECT
mu.Lock()
condition = true
cond.Signal()
mu.Unlock()
```

Why? Prevents race between signaler and waiter.

When to Use sync.Cond

✅ **Use Cond when:**

- Multiple goroutines wait for **same condition**
- Need to wake **one specific waiter** (Signal) vs **all waiters** (Broadcast)
- Condition is **complex** (can't express with channel)

❌ **Don't use Cond when:**

- Simple signaling (use channel + close)
- Producer-consumer (use buffered channel)
- One-time event (use `close(done)`)
- You're not sure (use channels—simpler)

Reality: 99% of Go code never uses Cond. Channels suffice.

Correct Usage Example

Example 1: Bounded Queue with Multiple Consumers

```
type Queue struct {
    mu       sync.Mutex
    items    []Item
    notEmpty *sync.Cond  // Signals "queue not empty"
    notFull  *sync.Cond  // Signals "queue not full"
    capacity int
}

func NewQueue(capacity int) *Queue {
    q := &Queue{
        items:    make([]Item, 0, capacity),
        capacity: capacity,
    }
    q.notEmpty = sync.NewCond(&q.mu)
    q.notFull = sync.NewCond(&q.mu)
    return q
}

func (q *Queue) Enqueue(item Item) {
    q.my.Lock()
    defer q.mu.Unlock()

    // Wait while queue full
    for len(q.items) == q.capacity {
        q.notFull.Wait()
    }

    q.items = append(q.items, item)
    q.notEmpty.Signal()  // Wake one consumer
```

```
}

func (q *Queue) Dequeue() Item {
    q.mu.Lock()
    defer q.mu.Unlock()

    // Wait while queue empty
    for len(q.items) == 0 {
        q.notEmpty.Wait()
    }

    item := q.items[0]
    q.items = q.items[1:]
    q.notFull.Signal()  // Wake one producer
    return item
}
```

Why Cond here?

- Multiple producers and consumers
- Need to wake specific waiter (one, not all)
- Two conditions (empty/full) share same lock

Example 2: Barrier (All Goroutines Reach Point)

```
type Barrier struct {
    mu       sync.Mutex
    cond     *sync.Cond
    count    int
    waiting  int
    released bool
}

func NewBarrier(count int) *Barrier {
    b := &Barrier{count: count}
    b.cond = sync.NewCond(&b.mu)
    return b
}

func (b *Barrier) Wait() {
    b.mu.Lock()
    defer b.mu.Unlock()

    b.waiting++

    if b.waiting == b.count {
        // Last goroutine arrives, wake everyone
        b.released = true
        b.cond.Broadcast()
        return
    }
```

```
    // Wait for all to arrive
    for !b.released {
        b.cond.Wait()
    }
}
```

Common Bugs

Bug 1: Forgetting Loop Around Wait

```
// WRONG
mu.Lock()
if !ready {
    cond.Wait()
}
mu.Unlock()
```

Problem: Spurious wakeup or stolen wake → condition still false.

Fix: `for !ready { cond.Wait() }`

Bug 2: Not Holding Lock

```
// WRONG
ready = true
cond.Signal()  // RACE: no lock held
```

Fix:

```
mu.Lock()
ready = true
cond.Signal()
mu.Unlock()
```

Bug 3: Using Wrong Lock

```
mu1 := sync.Mutex{}
mu2 := sync.Mutex{}
cond := sync.NewCond(&mu1)

mu2.Lock()  // Wrong lock!
cond.Wait() // PANIC: Unlock of unlocked mutex
```

Error:

```
fatal error: sync: unlock of unlocked mutex
```

Bug 4: Signal/Broadcast Outside Lock

```
// WRONG
ready = true  // Assignment races with waiter checking ready
cond.Broadcast()

// Waiter
for !ready {
    cond.Wait()
}
```

Race: Waiter checks `ready`, sees false, signaler sets true + broadcasts, waiter calls `Wait()` → missed wakeup.

Fix: Hold lock for both condition change and Signal/Broadcast.

Bug 5: Copying Cond

```
// WRONG
func process(c sync.Cond) {  // Passed by value
    c.Wait()  // Waits on copy, not original
}

// CORRECT
func process(c *sync.Cond) {
    c.Wait()
}
```

Cond vs Channel

Use Case	sync.Cond	Channel
One-time signal	Possible but awkward	`close(done)` (idiomatic)
Repeated signaling	Native (Signal/Broadcast)	Requires manual coordination
Wake one waiter	`Signal()`	Only with work-stealing
Wake all waiters	`Broadcast()`	`close(ch)` (once)
Carry data	No (condition on external state)	Yes (send value)
Readability	Low (unfamiliar to most)	High (Go idiom)

Rule of thumb: Use channels unless you specifically need `Signal()` vs `Broadcast()` distinction.

Channel Alternative to Cond

```
// Cond version
var (
```

```
    mu    sync.Mutex
    ready bool
    cond  = sync.NewCond(&mu)
)

// Waiter
mu.Lock()
for !ready {
    cond.Wait()
}
mu.Unlock()

// Signaler
mu.Lock()
ready = true
cond.Broadcast()
mu.Unlock()

// ---

// Channel version (simpler)
var ready = make(chan struct{})

// Waiter
<-ready

// Signaler
close(ready)  // Wakes all waiters
```

Channel wins: Simpler, no lock needed, same broadcast semantics.

When Cond wins: Need repeated signal/broadcast cycles without recreating channels.

Performance Characteristics

Operation	Time	Notes
Wait()	~500-1000 ns	Scheduler involvement
Signal()	~100-200 ns	Wake one waiter
Broadcast()	~100-500 ns	Wake all waiters (scales with waiter count)
Channel close	~50-100 ns	Wake all readers

Takeaway: Channel close is faster for one-time broadcast. Cond is for repeated cycles.

Real-World Example: You Probably Don't Need Cond

Most "Cond use cases" are better solved with channels:

```
// Perceived need for Cond: "Multiple goroutines wait for event"

// ❌ DON'T USE COND
var (
    mu    sync.Mutex
    ready bool
    cond  = sync.NewCond(&mu)
)

func worker() {
    mu.Lock()
    for !ready {
        cond.Wait()
    }
    mu.Unlock()
    doWork()
}

// ✅ USE CHANNEL
var ready = make(chan struct{})

func worker() {
    <-ready  // Simpler!
    doWork()
}

// Main:
// Cond: mu.Lock(); ready = true; cond.Broadcast(); mu.Unlock()
// Channel: close(ready)  // Much simpler
```

Cond is rarely needed in idiomatic Go.

Interview Traps

Trap 1: "Cond.Wait() returns when condition becomes true"

Wrong. Wait() returns when **signaled**; you must check condition.

Correct answer:
" `Wait()` returns when another goroutine calls `Signal()` or `Broadcast()`, or on spurious wakeup. It does NOT check the condition—you must check in a loop. Waiting goroutines can wake for reasons unrelated to your condition becoming true."

Trap 2: "Signal() wakes all waiters"

Wrong. `Signal()` wakes **one**; `Broadcast()` wakes **all**.

Correct answer:
" `Signal()` wakes at most one waiting goroutine (unspecified which). `Broadcast()` wakes all waiting goroutines. Use `Signal()` for work distribution (one goroutine handles the condition), and `Broadcast()` when all waiters should re-check the condition."

Trap 3: "I need Cond for my use case"

Probably wrong. 99% of cases: channels are better.

Correct answer:
"In Go, channels are the idiomatic way to coordinate goroutines. `sync.Cond` is rarely needed—it's useful for cases requiring repeated signal/broadcast cycles with multiple waiters on complex conditions. For one-time events, use `close(ch)`. For ongoing signaling, use buffered channels or worker pools."

Trap 4: "Cond is like a channel"

Superficial similarity.

Correct answer:
"Cond and channels both coordinate goroutines but serve different purposes. Channels transfer data and ownership with built-in synchronization. Cond is a low-level primitive for waiting on external conditions protected by a mutex. Channels are higher-level and more idiomatic in Go."

Key Takeaways

1. **Cond = waiting room for condition changes**
2. **Always wait in a loop** (spurious wakeups, stolen wakes)
3. **Hold lock when calling Wait/Signal/Broadcast**
4. **Signal() wakes one, Broadcast() wakes all**
5. **Rarely needed in Go** (channels suffice 99% of the time)
6. **Use channels unless you specifically need Signal vs Broadcast**
7. **Most complex** and **most error-prone** sync primitive
8. **If unsure, use channels**

What You Should Be Thinking Now

- "How does the Go memory model formalize happens-before?"
- "Why do some operations provide synchronization and others don't?"
- "What are the exact memory visibility guarantees?"
- "How do I reason about correctness without running code?"

Next: ../02-memory-model/go-memory-model.md - The foundation for reasoning about concurrent correctness.

Exercises (Do These Before Moving On)

1. Implement a bounded queue with Cond (producer-consumer). Test with multiple producers and consumers.
2. Write code that **forgets the loop** around `Wait()`. Observe incorrect behavior (may take many runs).
3. Implement the same bounded queue using **channels**. Compare complexity and readability.
4. Use `Signal()` vs `Broadcast()` incorrectly (wake all when you meant one). Observe the effects.
5. Create a barrier using Cond where N goroutines wait until all arrive.

Don't continue until you can explain: "Why does Go prefer channels over Cond, even though Cond is more flexible?"