sync/atomic

Definition (Precise)

Atomic operations are indivisible, lock-free operations on memory that provide synchronization without mutexes. They execute as a single, uninterruptible unit—no other goroutine can observe the operation half-complete.

Key insight: Atomics provide **both atomicity AND memory ordering** (happens-before guarantees).

Syntax

```
import "sync/atomic"

var counter int64

// Read
val := atomic.LoadInt64(&counter)

// Write
atomic.StoreInt64(&counter, 42)

// Add
atomic.AddInt64(&counter, 1)  // counter++
atomic.AddInt64(&counter, -1) // counter--

// Compare-and-swap
swapped := atomic.CompareAndSwapInt64(&counter, oldVal, newVal)

// Swap
oldVal := atomic.SwapInt64(&counter, newVal)
```

Mental Model

Think of atomic operations as **hardware-level transactions**:

- **Mutex:** Software lock (goroutine waits if locked)
- **Atomic:** Hardware instruction (CPU ensures atomicity)

```
Regular operations:
Read counter → Add 1 → Write counter
  ↑            ↑          ↑
  │            │          └─ Interruptible (race possible)
  │            └──────────── Between steps

Atomic operation:
[Read + Add + Write] ← Single, indivisible step
```

Supported Types

```
// Signed integers
atomic.AddInt32(addr *int32, delta int32) int32
atomic.AddInt64(addr *int64, delta int64) int64

// Unsigned integers
atomic.AddUint32(addr *uint32, delta uint32) uint32
atomic.AddUint64(addr *uint64, delta uint64) uint64

// Pointers
atomic.LoadPointer(addr *unsafe.Pointer) unsafe.Pointer
atomic.StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)

// Uintptr (used for pointers)
atomic.AddUintptr(addr *uintptr, delta uintptr) uintptr

// Generic (Go 1.19+)
var v atomic.Int64  // Type-safe wrapper
v.Add(1)
v.Load()
v.Store(42)
```

Note: No atomic operations for `float64`, `string`, `map`, `slice`. Mutexes only.

When to Use Atomic Operations

✅ **Use atomic for:**

- **Simple counters** (metrics, stats)
- **Flags** (boolean state)
- **Lock-free algorithms** (advanced use cases)
- **Performance-critical paths** (10x faster than mutex)

❌ **Don't use atomic for:**

- **Complex state** (multiple related fields)
- **Non-integer types** (must use mutex)
- **Operations requiring context** (read-modify-write on multiple values)

Correct Usage Patterns

Pattern 1: Simple Counter

```
type Metrics struct {
    requestCount int64  // atomic
}

func (m *Metrics) RecordRequest() {
    atomic.AddInt64(&m.requestCount, 1)
```

```
}

func (m *Metrics) RequestCount() int64 {
    return atomic.LoadInt64(&m.requestCount)
}
```

Why atomic?

- Single value (not complex state)
- High frequency (thousands/sec)
- No need for mutex overhead

Pattern 2: Configuration Pointer (Lock-Free Updates)

```
type Server struct {
    config atomic.Value  // Holds *Config
}

func (s *Server) UpdateConfig(newConfig *Config) {
    s.config.Store(newConfig)
}

func (s *Server) GetConfig() *Config {
    return s.config.Load().(*Config)
}

// Usage:
config := server.GetConfig()
// Read config fields (no lock needed)
// Consistent snapshot (config pointer is atomic)
```

Critical: Store entire config as pointer, not individual fields.

Pattern 3: Boolean Flag

```
type Worker struct {
    stopped uint32  // atomic (0 = running, 1 = stopped)
}

func (w *Worker) Stop() {
    atomic.StoreUint32(&w.stopped, 1)
}

func (w *Worker) IsStopped() bool {
    return atomic.LoadUint32(&w.stopped) == 1
}

func (w *Worker) Run() {
    for {
        if w.IsStopped() {
            return
```

```
        }
        doWork()
    }
}
```

Note: Go has no `atomic.Bool` (use `uint32` with 0/1).

Update (Go 1.19+):

```
type Worker struct {
    stopped atomic.Bool
}

func (w *Worker) Stop() {
    w.stopped.Store(true)
}

func (w *Worker) IsStopped() bool {
    return w.stopped.Load()
}
```

Pattern 4: Compare-and-Swap (CAS) Loop

```
// Atomic increment with CAS (educational; use AddInt64 in practice)
func incrementCAS(addr *int64) {
    for {
        old := atomic.LoadInt64(addr)
        new := old + 1
        if atomic.CompareAndSwapInt64(addr, old, new) {
            return  // Success
        }
        // CAS failed (another goroutine modified), retry
    }
}
```

Use case: Building lock-free data structures (advanced).

Common Bugs

Bug 1: Mixing Atomic and Non-Atomic Access

```
// WRONG
var counter int64

// Goroutine 1: atomic
atomic.AddInt64(&counter, 1)

// Goroutine 2: non-atomic
counter++  // DATA RACE
```

```
// Goroutine 3: non-atomic read
fmt.Println(counter)  // DATA RACE
```

Rule: Once you use atomic for a variable, **all accesses must be atomic**.

Fix:

```
// All access atomic
atomic.AddInt64(&counter, 1)
atomic.AddInt64(&counter, 1)
atomic.LoadInt64(&counter)
```

Bug 2: Using Atomic for Complex State

```
// WRONG
type Account struct {
    balance int64  // atomic
    name    string // mutex-protected? or also atomic?
}

func (a *Account) Deposit(amount int64, newName string) {
    atomic.AddInt64(&a.balance, amount)
    a.name = newName  // RACE: name not protected
}
```

Problem: Can't atomically update both fields together.

Fix: Use mutex for complex state.

```
type Account struct {
    mu      sync.Mutex
    balance int64
    name    string
}

func (a *Account) Deposit(amount int64, newName string) {
    a.mu.Lock()
    defer амы.Unlock()
    a.balance += amount
    a.name = newName
}
```

Bug 3: Forgetting Memory Ordering

```
// !WRONG: Assumes atomics don't need ordering
var data string
var ready int32  // atomic flag
```

```
// Writer
func writer() {
    data = "hello"                      // (1)
    atomic.StoreInt32(&ready, 1)        // (2)
}

// Reader
func reader() {
    for atomic.LoadInt32(&ready) == 0 {  // (3)
        runtime.Gosched()
    }
    fmt.Println(data)                    // (4)
}
```

Question: Is this safe?

Answer: YES. Atomic operations have **happens-before semantics**:

- (1) happens-before (2)
- (2) happens-before (3)
- (3) happens-before (4)

The atomic load/store provides memory ordering, ensuring `data` write is visible.

But if `ready` wasn't atomic:

```
// WRONG
ready = 1  // Non-atomic
// Reader may see ready==1 but NOT see data update (reordering)
```

Bug 4: Atomic.Value with Non-Pointer Types

```
// DANGEROUS
var v atomic.Value
v.Store(42)  // int
v.Store("hello")  // string - PANIC: inconsistent type
```

Rule: `atomic.Value` must always store the **same type**.

Safe usage:

```
var v atomic.Value
v.Store(&Config{...})  // Always *Config
v.Store(&Config{...})  // OK
```

Performance Characteristics

Operation	Uncontended	Contended (10 goroutines)
Mutex Lock/Unlock	~20-30 ns	~500-1000 ns

Atomic Add	~5-10 ns	~20-50 ns
Atomic Load/Store	~1-5 ns	~5-15 ns
Atomic CAS	~10-20 ns	~50-200 ns

Speedup:

- Atomic is **2-5x faster** than mutex (uncontended)
- Atomic is **10-50x faster** than mutex (contended)

Atomic vs Mutex Decision Tree

```
Is it a single primitive value (int, uint, bool, pointer)?
├─ NO → Use sync.Mutex
│   └─ atomic doesn't support complex types
│
└─ YES → Continue
    │
    Is it accessed at high frequency (>1000/sec)?
    ├─ NO → Use sync.Mutex
    │   └─ Simplicity more important than performance
    │
    └─ YES → Continue
        │
        Does it require atomicity across multiple operations?
        ├─ YES → Use sync.Mutex
        │   └─ Example: read, compute, write based on read
        │
        └─ NO → Use atomic
            └─ Example: increment, set flag, swap pointer
```

Memory Ordering Guarantees

Key insight: Atomic operations are **not just about atomicity**—they also provide **memory ordering**.

Sequential Consistency for Synchronization

Go's memory model guarantees:

- Before an atomic read, all previous memory operations are visible
- After an atomic write, the write is visible to subsequent atomic reads

```
var x int
var flag int32  // atomic

// Goroutine 1
x = 42                          // (1)
atomic.StoreInt32(&flag, 1)     // (2)

// Goroutine 2
for atomic.LoadInt32(&flag) == 0 {}  // (3) Wait for flag
```

```
fmt.Println(x)                        // (4) Guaranteed to see 42

// Happens-before: (1) → (2) → (3) → (4)
```

Without atomics:

```
// WRONG
x = 42
flag = 1  // Non-atomic

// Reader
for flag == 0 {}
fmt.Println(x)  // Might see 0! Compiler/CPU can reorder
```

Advanced: Lock-Free Stack (CAS Example)

```
type Node struct {
    value int
    next  *Node
}

type LockFreeStack struct {
    head unsafe.Pointer  // atomic *Node
}

func (s *LockFreeStack) Push(value int) {
    node := &Node{value: value}

    for {
        oldHead := atomic.LoadPointer(&s.head)
        node.next = (*Node)(oldHead)

        if atomic.CompareAndSwapPointer(&s.head, oldHead, unsafe.Pointer(node)) {
            return  // Success
        }
        // CAS failed, retry
    }
}

func (s *LockFreeStack) Pop() (int, bool) {
    for {
        oldHead := atomic.LoadPointer(&s.head)
        if oldHead == nil {
            return 0, false  // Empty
        }

        node := (*Node)(oldHead)
        newHead := unsafe.Pointer(node.next)

        if atomic.CompareAndSwapPointer(&s.head, oldHead, newHead) {
```

```
            return node.value, true  // Success
        }
        // CAS failed, retry
    }
}
```

Warning: Lock-free algorithms are **hard to get right**. Use proven libraries.

Real-World Failure: Non-Atomic Access

Company: Ad tech platform (2018)

What happened:
Metrics dashboard showed nonsensical values (negative counts, counts decreasing).

Root cause:

```
type Metrics struct {
    impressions int64  // Intended to be atomic
}

func (m *Metrics) RecordImpression() {
    // BUG: Sometimes atomic, sometimes not
    if fastPath {
        atomic.AddInt64(&m.impressions, 1)  // Atomic
    } else {
        m.impressions++  // NON-ATOMIC → RACE
    }
}

func (m *Metrics) GetImpressions() int64 {
    return m.impressions  // NON-ATOMIC → RACE
}
```

Problem:

- Mixed atomic and non-atomic access
- Race detector didn't catch it in tests (low probability)
- In production: corrupted reads (torn reads)

Example corruption:

```
Initial: 0x0000000000000100 (256)
Writer 1 (atomic): Writes 0x0000000000000200
Writer 2 (non-atomic): Writes 0x0000000000000300
Reader (non-atomic): Reads 0x0000000000000200 (or partial!)
                     Could read: 0x0000000000000300, 0x0000000000000200, or garbage
```

Fix:

```
func (m *Metrics) RecordImpression() {
    atomic.AddInt64(&m.impressions, 1)  // Always atomic
```

```
}

func (m *Metrics) GetImpressions() int64 {
    return atomic.LoadInt64(&m.impressions)  // Always atomic
}
```

Lessons:

1. **All access must be atomic** (no mixing)
2. Document atomicity requirement clearly
3. Use typed wrappers (Go 1.19+ `atomic.Int64`) to enforce

Go 1.19+ Typed Atomics (Preferred)

```
// Old way (error-prone)
var counter int64
atomic.AddInt64(&counter, 1)

// New way (Go 1.19+)
var counter atomic.Int64
counter.Add(1)
```

Benefits:

- Type-safe (can't mix atomic/non-atomic)
- Cleaner syntax (no pointers)
- Prevents misuse

Available types:

- `atomic.Int32`, `atomic.Int64`
- `atomic.Uint32`, `atomic.Uint64`
- `atomic.Uintptr`
- `atomic.Bool`
- `atomic.Pointer[T]`

Interview Traps

Trap 1: "Atomic operations don't need synchronization"

Wrong terminology. Atomics **are** synchronization.

Correct answer:
"Atomic operations provide synchronization without locks. They ensure atomicity (indivisibility) and memory ordering (happens-before guarantees). They're an alternative to mutexes for simple values, not a way to avoid synchronization altogether."

Trap 2: "I can use atomic for part of a struct"

Dangerous. Leads to partial protection.

Correct answer:
"While you can have atomic fields in a struct, you can't atomically update multiple fields together. If fields are

related (e.g., balance and lastUpdate time), you need a mutex to update them as a unit. Atomic fields should be independent or use atomic.Value for the entire struct pointer."

Trap 3: "Atomics are faster, so I should use them everywhere"

Wrong. Atomics are limited and harder to reason about.

Correct answer:
"Atomics are faster than mutexes but only work for simple types and operations. Mutexes are easier to reason about and support complex critical sections. Use atomics for hot paths with simple values (counters, flags). Use mutexes for everything else untilyou have proven performance issues."

Trap 4: "CompareAndSwap always succeeds eventually"

Wrong. Under high contention, CAS loops can livelock.

Correct answer:
"CompareAndSwap can fail repeatedly under high contention as other goroutines modify the value between load and CAS. While it typically succeeds quickly, pathological cases can cause many retries. For critical operations, consider mutex or limit retry attempts."

Key Takeaways

1. **Atomic = lock-free synchronization** for primitive types
2. **2-50x faster than mutexes** (depending on contention)
3. **All access must be atomic** (no mixing with non-atomic)
4. **Provides happens-before guarantees** (memory ordering)
5. **Limited to int32/64, uint32/64, uintptr, unsafe.Pointer**
6. **Use atomic.Value for pointers** (Go 1.19+ has atomic.Pointer[T])
7. **Go 1.19+ typed atomics** are preferred (safer)
8. **Best for counters and flags**, not complex state

What You Should Be Thinking Now

- "How do I wait for multiple goroutines to finish?"
- "What's sync.WaitGroup for?"
- "When should I use WaitGroup vs channels?"
- "How do I coordinate goroutine lifecycle?"

Next: waitgroup.md - Coordinating goroutine completion with WaitGroup.

Exercises (Do These Before Moving On)

1. Implement a counter using both mutex and atomic. Benchmark under contention (100 goroutines). Compare.
2. Write code that mixes atomic and non-atomic access. Run with race detector. Observe the warning.
3. Implement a bool flag using `uint32` atomics (pre-Go 1.19 style).
4. Use `atomic.Value` to implement a configuration hot-reload mechanism.
5. Write a CAS loop that increments a counter. Compare to `atomic.AddInt64`.

Don't continue until you can explain: "Why must ALL accesses to an atomic variable be atomic, even reads?"