False Sharing

What is False Sharing?

False sharing: Performance degradation when multiple CPUs access different variables that reside on the same cache line, causing unnecessary cache coherency traffic.

Critical insight: Even though goroutines access DIFFERENT variables (no logical sharing), they share the SAME cache line (hardware sharing).

Cache line: Smallest unit of memory transferred between CPU cache and main memory (typically 64 bytes on x86/ARM).

How CPU Caches Work

Cache Hierarchy

```
CPU Core 1                    CPU Core 2
    ↓                             ↓
L1 Cache (32 KB)              L1 Cache (32 KB)
    ↓                             ↓
L2 Cache (256 KB)             L2 Cache (256 KB)
    ↓                             ↓
        L3 Cache (8 MB, shared)
                ↓
            Main Memory
```

Latencies (approximate):

- L1: ~~4 cycles (~~1ns)
- L2: ~~12 cycles (~~3ns)
- L3: ~~40 cycles (~~10ns)
- Main memory: ~~200 cycles (~~50ns)

Cache Coherency (MESI Protocol)

When one CPU writes to a cache line, protocol ensures other CPUs' caches are invalidated.

States:

- **M**odified: This CPU has exclusive, modified copy
- **E**xclusive: This CPU has exclusive, clean copy
- **S**hared: Multiple CPUs have clean copies
- **I**nvalid: Cache line is stale

Example:

```
Initial: Variable x in Shared state in CPU1 and CPU2 caches

CPU1 writes x:
1. CPU1 cache line → Modified
2. CPU2 cache line → Invalid (invalidation message sent)
3. CPU2 must reload from memory on next access
```

False sharing: CPUs invalidate each other's caches even when accessing different variables.

Example 1: Counter Array False Sharing

```go
type Counter struct {
    a int64  // Offset 0-7
    b int64  // Offset 8-15
    // Both in same 64-byte cache line
}

var counter Counter

// Goroutine 1 on CPU 1
func incrementA() {
    for i := 0; i < 1000000; i++ {
        atomic.AddInt64(&counter.a, 1)
    }
}

// Goroutine 2 on CPU 2
func incrementB() {
    for i := 0; i < 1000000; i++ {
        atomic.AddInt64(&counter.b, 1)
    }
}

// False sharing:
// CPU1 writes counter.a → invalidates CPU2's cache line
// CPU2 writes counter.b → invalidates CPU1's cache line
// Constant cache line bouncing → slow
```

Benchmark:

```go
func BenchmarkFalseSharing(b *testing.B) {
    var counter Counter
    var wg sync.WaitGroup

    wg.Add(2)
    go func() {
        defer wg.Done()
        for i := 0; i < b.N; i++ {
            atomic.AddInt64(&counter.a, 1)
        }
    }()
    go func() {
        defer wg.Done()
        for i := 0; i < b.N; i++ {
            atomic.AddInt64(&counter.b, 1)
        }
    }()
```

```
    wg.Wait()
}

// Result: ~50ns per operation (cache line bouncing)
```

Fix: Padding

```
type Counter struct {
    a int64
    _ [56]byte  // Padding: 64 - 8 = 56 bytes
    b int64
    // a and b now in different cache lines
}

// Result: ~5ns per operation (10x faster!)
```

Example 2: Slice Element False Sharing

```
type Item struct {
    value int64
}

var items [8]Item

// 8 goroutines, each updates its own item
for i := 0; i < 8; i++ {
    go func(idx int) {
        for j := 0; j < 1000000; j++ {
            atomic.AddInt64(&items[idx].value, 1)
        }
    }(i)
}

// False sharing:
// Each Item is 8 bytes
// 8 Items = 64 bytes = 1 cache line
// All goroutines contend for same cache line
```

Benchmark:

```
func BenchmarkSliceFalseSharing(b *testing.B) {
    var items [8]Item
    var wg sync.WaitGroup

    for i := 0; i < 8; i++ {
        wg.Add(1)
        go func(idx int) {
            defer wg.Done()
            for j := 0; j < b.N; j++ {
```

```go
                atomic.AddInt64(&items[idx].value, 1)
            }
        }(i)
    }
    wg.Wait()
}

// Result: ~200ns per operation (severe contention)
```

Fix: Padding

```go
type Item struct {
    value int64
    _     [56]byte  // Padding
}

// Result: ~10ns per operation (20x faster!)
```

Example 3: Per-Goroutine State False Sharing

```go
type Worker struct {
    id    int
    count int64
}

var workers [4]Worker

// Each worker increments its own count
for i := 0; i < 4; i++ {
    go func(w *Worker) {
        for j := 0; j < 1000000; j++ {
            atomic.AddInt64(&w.count, 1)
        }
    }(&workers[i])
}

// False sharing: All Worker structs in same cache lines
```

Fix: Use sync.Pool or separate slices

```go
// Option 1: Padding
type Worker struct {
    id    int
    count int64
    _     [48]byte  // 64 - 16 = 48
}

// Option 2: Separate arrays
var workerIDs [4]int
```

```go
var workerCounts [4]int64
var _ [8]int64  // Padding between arrays
```

Cache Line Size in Go

```go
// Get cache line size
import "unsafe"

const CacheLinePad = 64  // Typical cache line size

// Or use runtime
import "runtime"

func getCacheLineSize() int {
    // Not directly exposed, assume 64 bytes
    return 64
}
```

Real-World Failure: High-Frequency Trading (2019)

Company: Financial trading firm
Issue: Latency spikes in order execution

Simplified scenario:

```go
type OrderBook struct {
    bidCount int64  // Offset 0
    askCount int64  // Offset 8
    // Both in same cache line
}

var book OrderBook

// Thread 1: Update bids
func updateBids() {
    for {
        atomic.AddInt64(&book.bidCount, 1)
        // Process bid
    }
}

// Thread 2: Update asks
func updateAsks() {
    for {
        atomic.AddInt64(&book.askCount, 1)
        // Process ask
    }
}
```

```
// False sharing: Cache line bouncing between threads
// Latency: 100ns → 2000ns (20x slower)
```

Impact: Missed trading opportunities (microseconds matter).

Fix: Padding

```go
type OrderBook struct {
    bidCount int64
    _        [56]byte
    askCount int64
    _        [56]byte
}

// Latency: 100ns (consistent)
```

Detecting False Sharing

1. Performance Profiling

```
go test -bench=. -cpuprofile=cpu.out
go tool pprof -http=:8080 cpu.out
```

Look for:

- High contention on atomic operations
- Cache misses (requires `perf` on Linux)

2. Linux `perf` Tool

```
# Record cache misses
perf record -e cache-misses,cache-references ./program

# Analyze
perf report
```

Metrics:

- **cache-misses:** High value indicates false sharing
- **cache-references:** Total cache accesses
- **Miss rate:** cache-misses / cache-references

False sharing symptoms:

- Miss rate > 10%
- High L1/L2 miss rates
- CPU time in atomic operations

3. Manual Inspection

Check struct layouts:

```
go build -gcflags="-m"  # Escape analysis
```

Look for:

- Small structs (< 64 bytes) accessed by multiple goroutines
- Arrays of small structs
- Adjacent atomic variables

Preventing False Sharing

Strategy 1: Padding

```go
type PaddedCounter struct {
    value int64
    _     [56]byte  // 64 - 8 = 56
}
```

Trade-off: Memory usage vs. performance.

Strategy 2: Alignment

```go
// Ensure 64-byte alignment
type AlignedCounter struct {
    _     [0]uint64  // Force 8-byte alignment
    value int64
    _     [7]uint64  // Pad to 64 bytes
}
```

Strategy 3: Separate Caching

```go
// Don't share data
type LocalCounter struct {
    local map[int64]int64  // Goroutine ID → count
}

func (c *LocalCounter) Increment() {
    id := goroutineID()
    c.local[id]++
}

func (c *LocalCounter) Total() int64 {
    sum := int64(0)
    for _, v := range c.local {
        sum += v
    }
    return sum
}
```

Strategy 4: Use sync.Pool

```
var pool = sync.Pool{
    New: func() interface{} {
        return &Counter{}
    },
}

func work() {
    c := pool.Get().(*Counter)
    defer pool.Put(c)

    // Each goroutine gets own counter (no sharing)
    c.value++
}
```

When to Worry About False Sharing

Optimize if:

1. High contention on atomic variables
2. Benchmarks show cache-line bouncing
3. Multiple goroutines update adjacent data
4. Latency-critical code (< 100ns per operation)

Don't optimize if:

1. Low contention
2. Data not "hot" (infrequent access)
3. Readability/maintainability matters more
4. Unsure (profile first!)

Premature optimization is root of all evil. Profile, then optimize.

Benchmarking False Sharing

```
package main

import (
    "sync"
    "sync/atomic"
    "testing"
)

// Without padding
type UnpaddedCounter struct {
    a, b int64
}

// With padding
type PaddedCounter struct {
```

```go
    a int64
    _ [56]byte
    b int64
}

func BenchmarkUnpadded(b *testing.B) {
    var c UnpaddedCounter
    var wg sync.WaitGroup

    wg.Add(2)
    go func() {
        defer wg.Done()
        for i := 0; i < b.N; i++ {
            atomic.AddInt64(&c.a, 1)
        }
    }()
    go func() {
        defer wg.Done()
        for i := 0; i < b.N; i++ {
            atomic.AddInt64(&c.b, 1)
        }
    }()
    wg.Wait()
}

func BenchmarkPadded(b *testing.B) {
    var c PaddedCounter
    var wg sync.WaitGroup

    wg.Add(2)
    go func() {
        defer wg.Done()
        for i := 0; i < b.N; i++ {
            atomic.AddInt64(&c.a, 1)
        }
    }()
    go func() {
        defer wg.Done()
        for i := 0; i < b.N; i++ {
            atomic.AddInt64(&c.b, 1)
        }
    }()
    wg.Wait()
}

// Run:
// go test -bench=. -benchmem
// BenchmarkUnpadded-8      20000000     ~80 ns/op
// BenchmarkPadded-8        200000000    ~8 ns/op
```

Interview Traps

Trap 1: "No data race, so no problem"

Wrong: "Goroutines access different variables, no synchronization needed."
Correct: "Even without logical sharing, false sharing causes performance degradation due to cache-line bouncing. Need padding to separate hot variables."

Trap 2: "Padding fixes all performance problems"

Wrong: "Add padding everywhere."
Correct: "Padding trades memory for performance. Only optimize hot paths with high contention. Profile first. Padding increases struct size, can hurt cache locality for other access patterns."

Trap 3: "Cache line is 32 bytes"

Wrong: Assumes outdated architecture.
Correct: "Modern x86/ARM CPUs have 64-byte cache lines. Always use 64 bytes for padding. Cache line size can vary (128 bytes on some systems), but 64 is safe default."

Trap 4: "Atomics eliminate false sharing"

Wrong: "atomic.AddInt64 is thread-safe, so no false sharing."
Correct: "Atomics prevent data races but don't prevent false sharing. Atomics on adjacent variables still cause cache-line invalidation. Need padding or separate cache lines."

Key Takeaways

1. **False sharing = hardware sharing, not logical sharing**
2. **Cache line is typically 64 bytes**
3. **Cache coherency causes invalidation** (MESI protocol)
4. **Padding separates hot variables into different cache lines**
5. **Profile before optimizing** (cache-misses with perf)
6. **Trade memory for performance** (padding increases size)
7. **Affects latency-critical code** (< 100ns operations)
8. **Go's atomic operations still have false sharing**

Exercises

1. Write benchmark comparing padded vs. unpadded counters. Measure speedup.

2. Create array of 8 counters. Benchmark with/without padding between elements.

3. Use Linux `perf` to measure cache-miss rate on padded vs. unpadded.

4. Calculate memory overhead of padding 1000 structs with 64-byte padding.

5. Explain why padding might HURT performance in some cases (hint: cache locality for sequential access).

Next: Section 04: Patterns - Learn production-ready concurrency patterns.

Congratulations!

You've completed **Section 03: Classic Problems**. You now understand:

- Race conditions (data races vs. logic bugs)
- Deadlocks (circular wait conditions)
- Livelocks (active but no progress)
- Starvation (resource denial)
- False sharing (cache-line bouncing)

These are the foundational bugs you'll encounter in concurrent systems. Always keep them in mind when writing goroutines and channels.

Next up: Patterns for building robust, scalable concurrent systems.