Code Review: Go Concurrency

This page is an addition to the [Go Code Review Comments](#) list. The goal of this list is to help to find concurrency-related bugs when reviewing Go code.

You may also read through this list just once to refresh your memory and to make sure you are aware of all these concurrency gotchas.

> ⚠️ *This page is community authored and maintained. It includes information that is disputed and may be misleading or incorrect.*

Insufficient synchronisation and race conditions

- [HTTP handler functions are thread-safe?](#)
- [Global functions and variables are protected by mutexes or otherwise thread-safe?](#)
- [*Reads* of fields and variables are protected?](#)
- [The loop variable is passed into the goroutine function as an argument?](#)
- [Methods on thread-safe types don't return pointers to protected structures?](#)
- [`Load()` or `Delete()` calls on a `sync.Map` after `Load()` is not a race condition?](#)

Testing

- [Running tests with `-race` flag in CI/CD?](#)

Scalability

- [A channel is intentionally created with zero capacity?](#)
- [*Not* using `sync.RWMutex` to protect very short operations?](#)

Time

- [`time.Ticker` is stopped using `defer tick.Stop()`?](#)
- [Comparing `time.Time` using `Equal()`, not `==`?](#)
- [Keeping the monotonic component in `time.Time` argument of `time.Since()`?](#)
- [When comparing *system times* via `t.Before(u)`, the monotonic component is stripped from the argument?](#)

Insufficient synchronisation and race conditions

[#](#) RC.1. **The HTTP handler function is safe to call concurrently from multiple goroutines?** It's easy to overlook HTTP handlers should be thread-safe because they are usually not called explicitly from anywhere in the project code, but only from the internals of the HTTP server.

[#](#) RC.2. Is there some **field or variable access not protected by a mutex** where the field or the variable is a primitive or of a type that is not explicitly thread-safe (such as `atomic.Value`), while this field can be updated from a concurrent goroutine? It's not safe to skip synchronising reads even to primitive variables because of non-atomic hardware writes and potential memory visibility problems.

See also a [Typical Data Race: Primitive unprotected variable](#).

[#](#) RC.3. **A method on a thread-safe type doesn't return a pointer to a protected structure?** This is a subtle bug which leads to the unprotected access problem described in the previous item. Example:

```go
type Counters struct {
    mu   sync.Mutex
    vals map[Key]*Counter
}


func (c *Counters) Add(k Key, amount int) {
    c.mu.Lock()
    defer c.mu.Unlock()
    count, ok := c.vals[k]
    if !ok {
        count = &Counter{sum: 0, num: 0}
        c.vals[k] = count
    }
    count.sum += amount
    count.n += 1
}


func (c *Counters) GetCounter(k Key) *Counter {
    c.mu.Lock()
    defer c.mu.Unlock()
    return c.vals[k] // BUG! Returns a pointer to the structure which must be
protected
}
```

One possible solution is to return a copy, not a pointer to the structure in `GetCounter()`:

```go
type Counters struct {
    mu   sync.Mutex
    vals map[Key]Counter // Note that now we are storing the Counters directly, not
pointers.
}

...

func (c *Counters) GetCounter(k Key) (count Counter, exists bool) {
    c.mu.Lock()
    defer c.mu.Unlock()
    return c.vals[k]
}
```

RC.4. If there is more than one goroutine that can update a `sync.Map`, **you don't call `m.Store()` or `m.Delete()` depending on the success of a previous `m.Load()` call?** In other words, the following code is racy:

```go
var m sync.Map

// Can be called concurrently from multiple goroutines
func DoSomething(k Key, v Value) {
    existing, ok := m.Load(k)
    if !ok {
```

```
        m.Store(k, v) // RACE CONDITION - two goroutines can execute this in
parallel
        ... some other logic, assuming the value in `k` is now `v` in the map
    }
    ...
}
```

Such a race condition could be benign in some cases: for example, the logic between the `Load()` and `Store()` calls computes the value to be cached in the map, and this computation always returns the same result and doesn't have side effects.

> ⚠️ **Potentially misleading information**. *"Race condition" can refer to logic errors, like this example, which can be benign. But the phrase is also commonly used to refer to violations of the memory model, which are never benign.*

If the race condition is not benign, use methods <u>sync.Map.LoadOrStore()</u> and <u>LoadAndDelete()</u> to fix it.

Scalability

[#](#) Sc.1. **Is this intentional that a channel is created with zero capacity**, like `make(chan *Foo)` ? A goroutine which sends a message to a zero-capacity channel is blocked until another goroutine receives this message. Omitting the capacity in the `make()` call might be just a mistake which will limit the scalability of the code, and it's likely unit test won't find such a bug.

> ⚠️ **Misleading information**. *Buffered channels do not inherently increase "scalability" versus unbuffered channels. However, buffered channels can easily obscure deadlocks and other fundamental design errors that would be immediately apparent with unbuffered channels.*

[#](#) Sc.2. Locking with `RWMutex` incurs extra overhead compared to plain `sync.Mutex` , and, furthermore, there might be some <u>scalability issue with the current implementation of</u> `RWMutex` <u>in Go</u>. Unless the case is very clear (such as an `RWMutex` used to synchronize many read-only operations each lasting hundreds of milliseconds or more, and the writes which require an exclusive lock seldom happen), **there should be some benchmarks proving that `RWMutex` indeed helps to improve the performance.** A typical example where `RWMutex` certainly does more harm than good is a simple protection of a variable in a struct:

```
type Box struct {
    mu sync.RWMutex // DON'T DO THIS -- use a simple Mutex instead.
    x  int
}

func (b *Box) Get() int {
    b.mu.RLock()
    defer b.mu.RUnlock()
    return b.x
}

func (b *Box) Set(x int) {
    b.mu.Lock()
    defer b.mu.Unlock()
    b.x = x
```

Time

Tm.1. **Is `time.Ticker` stopped using `defer tick.Stop()`?** It's a memory leak not to stop the ticker when the function which uses the ticker in a loop returns.

Tm.2. **Are `time.Time` structs compared using `Equal()` method, not just `==`?** Quoting the [documentation for](#) [`time.Time`](#):

> Note that the Go `==` operator compares not just the time instant but also the Location and the monotonic clock reading. Therefore, `Time` values should not be used as map or database keys without first guaranteeing that the identical Location has been set for all values, which can be achieved through use of the `UTC()` or `Local()` method, and that the monotonic clock reading has been stripped by setting `t = t.Round(0)`. In general, prefer `t.Equal(u)` to `t == u`, since `t.Equal()` uses the most accurate comparison available and correctly handles the case when only one of its arguments has a monotonic clock reading.

Tm.3. **Before calling `time.Since(t)`, the monotonic component is *not* stripped from `t`?** This is a consequence of the previous item. If the monotonic component is stripped from the `time.Time` structure before passing it into `time.Since()` function (via calling either `UTC()`, `Local()`, `In()`, `Round()`, `Truncate()`, or `AddDate()`) **the result of `time.Since()` might be negative** on very occasions, such as if the system time has been synced via NTP between the moment when the start time was originally obtained and the moment when `time.Since()` is called. If the monotonic component is *not* stripped, `time.Since()` will always return a positive duration.

Tm.4. **If you want to compare *system times* via `t.Before(u)`, do you strip the monotonic component from the argument,** e.g. via `u.Round(0)`? This is another point related to [Tm.2](#). Sometimes, you need to compare two `time.Time` structs only by the system time stored in them, specifically. You may need this before storing one of these `Time` structs on disk or sending them over the network. Imagine, for example, some sort of telemetry agent which pushes a telemetry metric together with time periodically to some remote system:

```go
var latestSentTime time.Time

func pushMetricPeriodically(ctx context.Context) {
    t := time.NewTicker(time.Second)
    defer t.Stop()
    for {
        select {
        case <-ctx.Done: return
        case <-t.C:
            newTime := time.Now().Round(0) // Strip monotonic component to compare system time only
            // Check that the new time is later to avoid messing up the telemetry if the system time
            // is set backwards on an NTP sync.
            if latestSentTime.Before(newTime) {
                sendOverNetwork(NewDataPoint(newTime, metric()))
                latestSentTime = newTime
            }
        }
    }
}
```

This code would be wrong without calling `Round(0)`, i. e. stripping the monotonic component.

Reading List

[Go Code Review Comments](): a checklist for reviewing Go code, not concurrency-specific.

Go concurrency:

- [The Go Memory Model]()
- [Section about concurrency in *Effective Go*]()
- Posts in The Go Blog:
    - [Share Memory By Communicating]()
    - [Go Concurrency Patterns: Timing out, moving on]()
    - [Go Concurrency Patterns: Context]()
    - [Go Concurrency Patterns: Pipelines and cancellation]()
    - [Advanced Go Concurrency Patterns]() (video)
    - [Rethinking Classical Concurrency Patterns]() (video)
- [Understanding Real-World Concurrency Bugs in Go]()

Concurrency, but not specific to Go:

- [Mechanical Sympathy: Single Writer Principle]()