The Go Memory Model

Version of June 6, 2022

Table of Contents

Introduction

The Go memory model specifies the conditions under which reads of a variable in one goroutine can be guaranteed to observe values produced by writes to the same variable in a different goroutine.

Advice

Programs that modify data being simultaneously accessed by multiple goroutines must serialize such access.

To serialize access, protect the data with channel operations or other synchronization primitives such as those in the `sync` and `sync/atomic` packages.

If you must read the rest of this document to understand the behavior of your program, you are being too clever.

Don't be clever.

Informal Overview

Go approaches its memory model in much the same way as the rest of the language, aiming to keep the semantics simple, understandable, and useful. This section gives a general overview of the approach and should suffice for most programmers. The memory model is specified more formally in the next section.

A *data race* is defined as a write to a memory location happening concurrently with another read or write to that same location, unless all the accesses involved are atomic data accesses as provided by the `sync/atomic` package. As noted already, programmers are strongly encouraged to use appropriate synchronization to avoid data races. In the absence of data races, Go programs behave as if all the goroutines were multiplexed onto a single processor. This property is sometimes referred to as DRF-SC: data-race-free programs execute in a sequentially consistent manner.

While programmers should write Go programs without data races, there are limitations to what a Go implementation can do in response to a data race. An implementation may always react to a data race by reporting the race and terminating the program. Otherwise, each read of a single-word-sized or sub-word-sized memory location must observe a value actually written to that location (perhaps by a concurrent executing goroutine) and not yet overwritten. These implementation constraints make Go more like Java or JavaScript, in that most races have a limited number of outcomes, and less like C and C++, where the meaning of any program with a race is entirely undefined, and the compiler may do anything at all. Go's approach aims to make errant programs more reliable and easier to debug, while still insisting that races are errors and that tools can diagnose and report them.

Memory Model

The following formal definition of Go's memory model closely follows the approach presented by Hans-J. Boehm and Sarita V. Adve in "Foundations of the C++ Concurrency Memory Model", published in PLDI 2008. The definition of data-race-free programs and the guarantee of sequential consistency for race-free programs are equivalent to the ones in that work.

The memory model describes the requirements on program executions, which are made up of goroutine executions, which in turn are made up of memory operations.

A *memory operation* is modeled by four details:

- its kind, indicating whether it is an ordinary data read, an ordinary data write, or a *synchronizing operation* such as an atomic data access, a mutex operation, or a channel operation,
- its location in the program,
- the memory location or variable being accessed, and

- the values read or written by the operation.

Some memory operations are *read-like*, including read, atomic read, mutex lock, and channel receive. Other memory operations are *write-like*, including write, atomic write, mutex unlock, channel send, and channel close. Some, such as atomic compare-and-swap, are both read-like and write-like.

A *goroutine execution* is modeled as a set of memory operations executed by a single goroutine.

Requirement 1: The memory operations in each goroutine must correspond to a correct sequential execution of that goroutine, given the values read from and written to memory. That execution must be consistent with the *sequenced before* relation, defined as the partial order requirements set out by the Go language specification for Go's control flow constructs as well as the order of evaluation for expressions.

A Go *program execution* is modeled as a set of goroutine executions, together with a mapping W that specifies the write-like operation that each read-like operation reads from. (Multiple executions of the same program can have different program executions.)

Requirement 2: For a given program execution, the mapping W, when limited to synchronizing operations, must be explainable by some implicit total order of the synchronizing operations that is consistent with sequencing and the values read and written by those operations.

The *synchronized before* relation is a partial order on synchronizing memory operations, derived from W. If a synchronizing read-like memory operation r observes a synchronizing write-like memory operation w (that is, if $W(r) = w$), then w is synchronized before r. Informally, the synchronized before relation is a subset of the implied total order mentioned in the previous paragraph, limited to the information that W directly observes.

The *happens before* relation is defined as the transitive closure of the union of the sequenced before and synchronized before relations.

Requirement 3: For an ordinary (non-synchronizing) data read r on a memory location x, $W(r)$ must be a write w that is *visible* to r, where visible means that both of the following hold:

1. w happens before r.
2. w does not happen before any other write w' (to x) that happens before r.

A *read-write data race* on memory location x consists of a read-like memory operation r on x and a write-like memory operation w on x, at least one of which is non-synchronizing, which are unordered by happens before (that is, neither r happens before w nor w happens before r).

A *write-write data race* on memory location *x* consists of two write-like memory operations *w* and *w*' on *x*, at least one of which is non-synchronizing, which are unordered by happens before.

Note that if there are no read-write or write-write data races on memory location *x*, then any read *r* on *x* has only one possible *W*(*r*): the single *w* that immediately precedes it in the happens before order.

More generally, it can be shown that any Go program that is data-race-free, meaning it has no program executions with read-write or write-write data races, can only have outcomes explained by some sequentially consistent interleaving of the goroutine executions. (The proof is the same as Section 7 of Boehm and Adve's paper cited above.) This property is called DRF-SC.

The intent of the formal definition is to match the DRF-SC guarantee provided to race-free programs by other languages, including C, C++, Java, JavaScript, Rust, and Swift.

Certain Go language operations such as goroutine creation and memory allocation act as synchronization operations. The effect of these operations on the synchronized-before partial order is documented in the “Synchronization” section below. Individual packages are responsible for providing similar documentation for their own operations.

Implementation Restrictions for Programs Containing Data Races

The preceding section gave a formal definition of data-race-free program execution. This section informally describes the semantics that implementations must provide for programs that do contain races.

Any implementation can, upon detecting a data race, report the race and halt execution of the program. Implementations using ThreadSanitizer (accessed with “`go build -race`”) do exactly this.

A read of an array, struct, or complex number may by implemented as a read of each individual sub-value (array element, struct field, or real/imaginary component), in any order. Similarly, a write of an array, struct, or complex number may be implemented as a write of each individual sub-value, in any order.

A read *r* of a memory location *x* holding a value that is not larger than a machine word must observe some write *w* such that *r* does not happen before *w* and there is no write *w*' such that *w* happens before *w*' and *w*' happens before *r*. That is, each read must observe a value written by a preceding or concurrent write.

Additionally, observation of acausal and “out of thin air” writes is disallowed.

Reads of memory locations larger than a single machine word are encouraged but not required to meet the same semantics as word-sized memory locations, observing a single allowed write *w*. For performance reasons, implementations may instead treat larger operations as a set of individual machine-word-sized operations in an unspecified order. This means that races on multiword data structures can lead to inconsistent values not corresponding to a single write. When the values depend on the consistency of internal (pointer, length) or (pointer, type) pairs, as can be the case for interface values, maps, slices, and strings in most Go implementations, such races can in turn lead to arbitrary memory corruption.

Examples of incorrect synchronization are given in the “Incorrect synchronization” section below.

Examples of the limitations on implementations are given in the “Incorrect compilation” section below.

Synchronization

Initialization

Program initialization runs in a single goroutine, but that goroutine may create other goroutines, which run concurrently.

If a package `p` imports package `q`, the completion of `q`'s `init` functions happens before the start of any of `p`'s.

The completion of all `init` functions is synchronized before the start of the function `main.main`.

Goroutine creation

The go statement that starts a new goroutine is synchronized before the start of the goroutine's execution.

For example, in this program:

```
var a string

func f() {
        print(a)
}

func hello() {
        a = "hello, world"
```

```
        go f()
}
```

calling `hello` will print `"hello, world"` at some point in the future (perhaps after `hello` has returned).

Goroutine destruction

The exit of a goroutine is not guaranteed to be synchronized before any event in the program. For example, in this program:

```
var a string

func hello() {
        go func() { a = "hello" }()
        print(a)
}
```

the assignment to a is not followed by any synchronization event, so it is not guaranteed to be observed by any other goroutine. In fact, an aggressive compiler might delete the entire go statement.

If the effects of a goroutine must be observed by another goroutine, use a synchronization mechanism such as a lock or channel communication to establish a relative ordering.

Channel communication

Channel communication is the main method of synchronization between goroutines. Each send on a particular channel is matched to a corresponding receive from that channel, usually in a different goroutine.

A send on a channel is synchronized before the completion of the corresponding receive from that channel.

This program:

```
var c = make(chan int, 10)
var a string

func f() {
```

```
        a = "hello, world"
        c <- 0
}

func main() {
        go f()
        <-c
        print(a)
}
```

is guaranteed to print `"hello, world"`. The write to `a` is sequenced before the send on `c`, which is synchronized before the corresponding receive on `c` completes, which is sequenced before the `print`.

The closing of a channel is synchronized before a receive that returns a zero value because the channel is closed.

In the previous example, replacing `c <- 0` with `close(c)` yields a program with the same guaranteed behavior.

A receive from an unbuffered channel is synchronized before the completion of the corresponding send on that channel.

This program (as above, but with the send and receive statements swapped and using an unbuffered channel):

```
var c = make(chan int)
var a string

func f() {
        a = "hello, world"
        <-c
}

func main() {
        go f()
        c <- 0
        print(a)
}
```

is also guaranteed to print "hello, world". The write to a is sequenced before the receive on c, which is synchronized before the corresponding send on c completes, which is sequenced before the print.

If the channel were buffered (e.g., c = make(chan int, 1)) then the program would not be guaranteed to print "hello, world". (It might print the empty string, crash, or do something else.)

The kth receive on a channel with capacity C is synchronized before the completion of the k+Cth send from that channel completes.

This rule generalizes the previous rule to buffered channels. It allows a counting semaphore to be modeled by a buffered channel: the number of items in the channel corresponds to the number of active uses, the capacity of the channel corresponds to the maximum number of simultaneous uses, sending an item acquires the semaphore, and receiving an item releases the semaphore. This is a common idiom for limiting concurrency.

This program starts a goroutine for every entry in the work list, but the goroutines coordinate using the limit channel to ensure that at most three are running work functions at a time.

```
var limit = make(chan int, 3)

func main() {
        for _, w := range work {
                go func(w func()) {
                        limit <- 1
                        w()
                        <-limit
                }(w)
        }
        select{}
}
```

Locks

The sync package implements two lock data types, sync.Mutex and sync.RWMutex.

For any sync.Mutex or sync.RWMutex variable l and n < m, call n of l.Unlock() is synchronized before call m of l.Lock() returns.

This program:

```
var l sync.Mutex
var a string

func f() {
        a = "hello, world"
        l.Unlock()
}

func main() {
        l.Lock()
        go f()
        l.Lock()
        print(a)
}
```

is guaranteed to print `"hello, world"`. The first call to `l.Unlock()` (in `f`) is synchronized before the second call to `l.Lock()` (in `main`) returns, which is sequenced before the `print`.

For any call to `l.RLock` on a `sync.RWMutex` variable `l`, there is an n such that the nth call to `l.UnLock` is synchronized before the return from `l.RLock`, and the matching call to `l.RUnLock` is synchronized before the return from call n+1 to `l.Lock`.

A successful call to `l.TryLock` (or `l.TryRLock`) is equivalent to a call to `l.Lock` (or `l.RLock`). An unsuccessful call has no synchronizing effect at all. As far as the memory model is concerned, `l.TryLock` (or `l.TryRLock`) may be considered to be able to return false even when the mutex l is unlocked.

Once

The `sync` package provides a safe mechanism for initialization in the presence of multiple goroutines through the use of the `Once` type. Multiple threads can execute `once.Do(f)` for a particular `f`, but only one will run `f()`, and the other calls block until `f()` has returned.

The completion of a single call of `f()` from `once.Do(f)` is synchronized before the return of any call of `once.Do(f)`.

In this program:

```
var a string
var once sync.Once

func setup() {
        a = "hello, world"
}

func doprint() {
        once.Do(setup)
        print(a)
}

func twoprint() {
        go doprint()
        go doprint()
}
```

calling `twoprint` will call `setup` exactly once. The `setup` function will complete before either call of `print`. The result will be that `"hello, world"` will be printed twice.

Atomic Values

The APIs in the `sync/atomic` package are collectively "atomic operations" that can be used to synchronize the execution of different goroutines. If the effect of an atomic operation *A* is observed by atomic operation *B*, then *A* is synchronized before *B*. All the atomic operations executed in a program behave as though executed in some sequentially consistent order.

The preceding definition has the same semantics as C++'s sequentially consistent atomics and Java's `volatile` variables.

Finalizers

The `runtime` package provides a `SetFinalizer` function that adds a finalizer to be called when a particular object is no longer reachable by the program. A call to `SetFinalizer(x, f)` is synchronized before the finalization call `f(x)`.

Additional Mechanisms

The sync package provides additional synchronization abstractions, including condition variables, lock-free maps, allocation pools, and wait groups. The documentation for each of these specifies the guarantees it makes concerning synchronization.

Other packages that provide synchronization abstractions should document the guarantees they make too.

Incorrect synchronization

Programs with races are incorrect and can exhibit non-sequentially consistent executions. In particular, note that a read *r* may observe the value written by any write *w* that executes concurrently with *r*. Even if this occurs, it does not imply that reads happening after *r* will observe writes that happened before *w*.

In this program:

```
var a, b int

func f() {
        a = 1
        b = 2
}

func g() {
        print(b)
        print(a)
}

func main() {
        go f()
        g()
}
```

it can happen that g prints 2 and then 0.

This fact invalidates a few common idioms.

Double-checked locking is an attempt to avoid the overhead of synchronization. For example, the `twoprint` program might be incorrectly written as:

```
var a string
var done bool

func setup() {
        a = "hello, world"
        done = true
}

func doprint() {
        if !done {
                once.Do(setup)
        }
        print(a)
}

func twoprint() {
        go doprint()
        go doprint()
}
```

but there is no guarantee that, in `doprint`, observing the write to `done` implies observing the write to `a`. This version can (incorrectly) print an empty string instead of `"hello, world"`.

Another incorrect idiom is busy waiting for a value, as in:

```
var a string
var done bool

func setup() {
        a = "hello, world"
        done = true
}
```

```
func main() {
        go setup()
        for !done {
        }
        print(a)
}
```

As before, there is no guarantee that, in `main`, observing the write to `done` implies observing the write to `a`, so this program could print an empty string too. Worse, there is no guarantee that the write to `done` will ever be observed by `main`, since there are no synchronization events between the two threads. The loop in `main` is not guaranteed to finish.

There are subtler variants on this theme, such as this program.

```
type T struct {
        msg string
}

var g *T

func setup() {
        t := new(T)
        t.msg = "hello, world"
        g = t
}

func main() {
        go setup()
        for g == nil {
        }
        print(g.msg)
}
```

Even if `main` observes `g != nil` and exits its loop, there is no guarantee that it will observe the initialized value for `g.msg`.

In all these examples, the solution is the same: use explicit synchronization.

Incorrect compilation

The Go memory model restricts compiler optimizations as much as it does Go programs. Some compiler optimizations that would be valid in single-threaded programs are not valid in all Go programs. In particular, a compiler must not introduce writes that do not exist in the original program, it must not allow a single read to observe multiple values, and it must not allow a single write to write multiple values.

All the following examples assume that `*p` and `*q` refer to memory locations accessible to multiple goroutines.

Not introducing data races into race-free programs means not moving writes out of conditional statements in which they appear. For example, a compiler must not invert the conditional in this program:

```
*p = 1
if cond {
        *p = 2
}
```

That is, the compiler must not rewrite the program into this one:

```
*p = 2
if !cond {
        *p = 1
}
```

If `cond` is false and another goroutine is reading `*p`, then in the original program, the other goroutine can only observe any prior value of `*p` and `1`. In the rewritten program, the other goroutine can observe `2`, which was previously impossible.

Not introducing data races also means not assuming that loops terminate. For example, a compiler must in general not move the accesses to `*p` or `*q` ahead of the loop in this program:

```
n := 0
for e := list; e != nil; e = e.next {
        n++
}
```

```
i := *p
*q = 1
```

If `list` pointed to a cyclic list, then the original program would never access `*p` or `*q`, but the rewritten program would. (Moving `*p` ahead would be safe if the compiler can prove `*p` will not panic; moving `*q` ahead would also require the compiler proving that no other goroutine can access `*q`.)

Not introducing data races also means not assuming that called functions always return or are free of synchronization operations. For example, a compiler must not move the accesses to `*p` or `*q` ahead of the function call in this program (at least not without direct knowledge of the precise behavior of `f`):

```
f()
i := *p
*q = 1
```

If the call never returned, then once again the original program would never access `*p` or `*q`, but the rewritten program would. And if the call contained synchronizing operations, then the original program could establish happens before edges preceding the accesses to `*p` and `*q`, but the rewritten program would not.

Not allowing a single read to observe multiple values means not reloading local variables from shared memory. For example, a compiler must not discard `i` and reload it a second time from `*p` in this program:

```
i := *p
if i < 0 || i >= len(funcs) {
        panic("invalid function index")
}
... complex code ...
// compiler must NOT reload i = *p here
funcs[i]()
```

If the complex code needs many registers, a compiler for single-threaded programs could discard `i` without saving a copy and then reload `i = *p` just before `funcs[i]()`. A Go compiler must not, because the value of `*p` may have changed. (Instead, the compiler could spill `i` to the stack.)

Not allowing a single write to write multiple values also means not using the memory where a local variable will be written as temporary storage before the write. For example, a compiler must not use `*p` as temporary storage in this program:

```
*p = i + *p/2
```

That is, it must not rewrite the program into this one:

```
*p /= 2
*p += i
```

If `i` and `*p` start equal to 2, the original code does `*p = 3`, so a racing thread can read only 2 or 3 from `*p`. The rewritten code does `*p = 1` and then `*p = 3`, allowing a racing thread to read 1 as well.

Note that all these optimizations are permitted in C/C++ compilers: a Go compiler sharing a back end with a C/C++ compiler must take care to disable optimizations that are invalid for Go.

Note that the prohibition on introducing data races does not apply if the compiler can prove that the races do not affect correct execution on the target platform. For example, on essentially all CPUs, it is valid to rewrite

```
n := 0
for i := 0; i < m; i++ {
        n += *shared
}
```

into:

```
n := 0
local := *shared
for i := 0; i < m; i++ {
        n += local
}
```

provided it can be proved that `*shared` will not fault on access, because the potential added read will not affect any existing concurrent reads or writes. On the other hand, the rewrite would not be valid in a source-to-source translator.

Conclusion

Go programmers writing data-race-free programs can rely on sequentially consistent execution of those programs, just as in essentially all other modern programming languages.

When it comes to programs with races, both programmers and compilers should remember the advice: don't be clever.