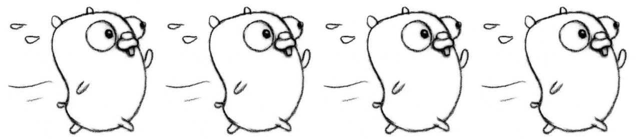
Updating the Go Memory Model

Memory Models, Part 3

Russ Cox

July 12, 2021

research.swtch.com/gomm



The current Go language memory model was written in 2009, with minor updates since. It is clear that there are at least a few details that we should add to the current memory model, among them an explicit endorsement of race detectors and a clear statement of how the APIs in `sync/atomic` synchronize programs.

This post restates Go's overall philosophy and the current memory model and then outlines the relatively small adjustments I believe that we should make to the Go memory model. It assumes the background presented in the earlier posts "Hardware Memory Models" and "Programming Language Memory Models."

I have opened a GitHub discussion to collect feedback on the ideas presented here. Based on that feedback, I intend to prepare a formal Go proposal later this month. The use of GitHub discussions is itself a bit of an experiment, continuing to try to find a reasonable way to scale discussions of important changes.

Go's Design Philosophy

Go aims to be a programming environment for building practical, efficient systems. It aims to be lightweight for small projects but also scale up gracefully to large projects and large engineering teams.

Go encourages approaching concurrency at a high level, in particular through communication. The first Go proverb is "Don't communicate by sharing memory. Share memory by communicating." Another popular proverb is that "Clear is better than clever." In other words, Go encourages avoiding subtle bugs by avoiding subtle code.

Go aims not just for understandable programs but also for an understandable language and understandable package APIs. Complex or subtle language features or APIs contradict that goal. As Tony Hoare said in his 1980 Turing award lecture:

> I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies.
>
> The first method is far more difficult. It demands the same skill, devotion, insight, and even inspiration as the discovery of the simple physical laws which underlie the complex phenomena of nature. It also requires a willingness to accept objectives which are limited by physical, logical, and technological constraints, and to accept a compromise when conflicting objectives cannot be met.

This aligns pretty well with Go's philosophy for APIs. We typically spend a long time during design to make sure an API is right, working to reduce it to its minimal, most useful essence.

Another aspect of Go being a useful programming environment is having well-defined semantics for the most common programming mistakes, which aids both understandability and debugging. This idea is hardly new. Quoting Tony Hoare again, this time from his 1972 “Quality of Software” checklist:

> As well as being very simple to use, a software program must be very difficult to misuse; it must be kind to programming errors, giving clear indication of their occurrence, and never becoming unpredictable in its effects.

The common sense of having well-defined semantics for buggy programs is not as common as one might expect. In C/C++, undefined behavior has evolved into a kind of compiler writer’s *carte blanche* to turn slightly buggy programs into very differently buggy programs, in ever more interesting ways. Go does not take this approach: there is no “undefined behavior.” In particular, bugs like null pointer dereferences, integer overflow, and unintentional infinite loops all have defined semantics in Go.

Go’s Memory Model Today

Go’s memory model begins with the following advice, consistent with Go’s overall philosophy:

> 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.

This remains good advice. The advice is also consistent with other languages’ encouragement of DRF-SC: synchronize to eliminate data races, and then programs will behave as if sequentially consistent, leaving no need to understand the remainder of the memory model.

After this advice, the Go memory model defines a conventional happens-before-based definition of racing reads and writes. Like in Java and JavaScript, a read in Go can observe any earlier but not yet overwritten write in the happens-before order, or any racing write; arranging to have only one such write forces a specific outcome.

The memory model then goes on to define the synchronization operations that establish cross-goroutine happens-before edges. The operations are the usual ones, with some Go-specific flavoring:

- If a package `p` imports package `q`, the completion of `q`’s `init` functions happens before the start of any of `p`’s.
- The start of the function `main.main` happens after all `init` functions have finished.
- The `go` statement that starts a new goroutine happens before the goroutine’s execution begins.
- A send on a channel happens before the corresponding receive from that channel completes.
- The closing of a channel happens before a receive that returns a zero value because the channel is closed.

- A receive from an unbuffered channel happens before the send on that channel completes.
- The *k*'th receive on a channel with capacity *C* happens before the *k*+*C*'th send from that channel completes.
- For any `sync.Mutex` or `sync.RWMutex` variable `l` and $n < m$, call *n* of `l.Unlock()` happens before call *m* of `l.Lock()` returns.
- A single call of `f()` from `once.Do(f)` happens (returns) before any call of `once.Do(f)` returns.

This list notably omits any mention of `sync/atomic` as well as newer APIs in package `sync`.

The memory model ends with some examples of incorrect synchronization. It contains no examples of incorrect compilation.

Changes to Go's Memory Model

In 2009, as we set out to write Go's memory model, the Java memory model was newly revised, and the C/C++11 memory model was being finalized. We were strongly encouraged by some to adopt the C/C++11 model, taking advantage of all the work that had gone into it. That seemed risky to us. Instead, we decided on a more conservative approach to what guarantees we would make, a decision confirmed by the subsequent decade of papers detailing very subtle problems in the Java/C/C++ line of memory models. Defining enough of a memory model to guide programmers and compiler writers is important, but defining one in complete formality—correctly!—seems still just beyond the grasp of the most talented researchers. It should suffice for Go to continue to say the minimum needed to be useful.

This section list the adjustments I believe we should make. As noted earlier, I have opened a GitHub discussion to collect feedback. Based on that feedback, I plan to prepare a formal Go proposal later this month.

Document Go's overall approach

The "don't be clever" advice is important and should stay, but we also need to say more about Go's overall approach before diving into the details of happens-before. I have seen multiple incorrect summaries of Go's approach, such as claiming that Go's model is C/C++'s "DRF-SC or Catch Fire." Misreadings are understandable: the document doesn't say what the approach is, and it is so short (and the material so subtle) that people see what they expect to see rather than what is or is not there.

The text to be added would be something along the lines of:

> **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.
>
> 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.

> Other programming languages typically take one of two approaches to programs containing data races. The first, exemplified by C and C++, is that programs with data races are invalid: a compiler may break them in arbitrarily surprising ways. The second, exemplified by Java and JavaScript, is that programs with data races have defined semantics, limiting the possible impact of a race and making programs more reliable and easier to debug. Go's approach sits between these two. Programs with data races are invalid in the sense that an implementation may report the race and terminate the program. But otherwise, programs with data races have defined semantics with a limited number of outcomes, making errant programs more reliable and easier to debug.

This text should make clear how Go is and is not like other languages, correcting any prior expectations on the part of the reader.

At the end of the "Happens Before" section, we should also clarify that certain races can still lead to corruption. It currently ends with:

> Reads and writes of values larger than a single machine word behave as multiple machine-word-sized operations in an unspecified order.

We should add:

> Note that 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 is the case for interface values, maps, slices, and strings in most Go implementations, such races can in turn lead to arbitrary memory corruption.

This will more clearly state the limitations of the guarantees on programs with data races.

Document happens-before for sync libraries

New APIs have been added to the `sync` package since the memory model was written. We need to add them to the memory model (issue #7948). Thankfully, the additions seem straightforward. I believe they are as follows.

- For `sync.Cond`: `Broadcast` or `Signal` happens before the return of any `Wait` call that it unblocks.
- For `sync.Map`: `Load`, `LoadAndDelete`, and `LoadOrStore` are read operations. `Delete`, `LoadAndDelete`, and `Store` are write operations. `LoadOrStore` is a write operation when it returns `loaded` set to `false`. A write operation happens before any read operation that observes the effect of the write.
- For `sync.Pool`: A call to `Put(x)` happens before a call to `Get` returning that same value `x`. Similarly, a call to `New` returning `x` happens before a call to `Get` returning that same value `x`.
- For `sync.WaitGroup`: A call to `Done` happens before the return of any `Wait` call that it unblocks.

Users of these APIs need to know the guarantees in order to use them effectively. Therefore, while we should keep the text in the memory model for illustrative purposes, we should also include it in the doc comments in package `sync`. This will also help set an example for third-party synchronization primitives of the importance of documenting the ordering guarantees established by an API.

Document happens-before for sync/atomic

Atomic operations are missing from the memory model. We need to add them (issue #5045). I believe we should say:

> 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 happens before B. All the atomic operations executed in a program behave as though executed in some sequentially consistent order.

This is what Dmitri Vyukov suggested in 2013 and what I informally promised in 2016. It also has the same semantics as Java's `volatiles` and C++'s default atomics.

In terms of the C/C++ menu, there are only two choices for synchronizing atomics: sequentially consistent or acquire/release. (Relaxed atomics do not create happens-before edges and therefore have no synchronizing effect.) The decision between those comes down to, first, how important it is to be able to reason about the relative order of atomic operations on multiple locations, and, second, how much more expensive sequentially consistent atomics are compared to acquire/release atomics.

On the first consideration, reasoning about the relative order of atomic operations on multiple locations is very important. In an earlier post I gave an example of a condition variable with a lock-free fast path implemented using two atomic variables, broken by using acquire/release atomics. This pattern appears again and again. For example, a past implementation of `sync.WaitGroup` used a pair of atomic `uint32` values, `wg.counter` and `wg.waiters`. The Go runtime implementation of semaphores also depends on two separate atomic words, namely the semaphore value `*addr` and the corresponding waiter count `root.nwait`. There are more. In the absence of sequentially consistent semantics (that is, if we instead adopt acquire/release semantics), people will still write code like this; it will just fail mysteriously, and only in certain contexts.

The fundamental problem is that using acquire/release atomics to make a program data-race-free does not result in a program that behaves in a sequentially consistent manner, because the atomics themselves don't. That is, such programs do not provide DRF-SC. This makes such programs very difficult to reason about and therefore difficult to write correctly.

On the second consideration, as noted in the earlier post, hardware designers are starting to provide direct support for sequentially consistent atomics. For example, ARMv8 adds the `ldar` and `stlr` instructions for implementing sequentially consistent atomics, and they are also the recommended implementation of acquire/release atomics. If we adopted acquire/release semantics for `sync/atomic`, programs written on ARMv8 would be getting sequential consistency anyway. This would undoubtedly lead to programs that rely on the stronger ordering accidentally, breaking on weaker platforms. This may even happen on a single architecture, if the difference between acquire/release and sequentially consistent atomics is difficult to observe in practice due to race windows being small.

Both considerations strongly suggest we should adopt sequentially consistent atomics over acquire/release atomics: sequentially consistent atomics are more useful, and some chips have already completely closed the gap between the two levels. Presumably others will do the same if the gap is significant.

The same considerations, along with Go's overall philosophy of having minimal, easily understood APIs, argue against providing acquire/release as an additional, parallel set of APIs. It seems best to provide only the most understandable, most useful, least misusable set of atomic operations.

Another possibility would be to provide raw barriers instead of atomic operations. (C++ provides both, of course.) Barriers have the drawback of making expectations less clear and being somewhat more architecture-specific. Hans Boehm's page "Why atomics have integrated ordering constraints" presents the arguments for providing atomics instead of barriers (he uses the term fences). Generally, the atomics are far easier to understand than fences, and since we already provide atomic operations today, we can't easily remove them. Better to have one mechanism than two.

Maybe: Add a typed API to sync/atomic

The definitions above say that when a particular piece of memory must be accessed concurrently by multiple goroutines without other synchronization, the only way to eliminate the race is to make all the accesses use atomics. It is not enough to make only some of the accesses use atomics. For example, a non-atomic write concurrent with atomic reads or writes is still a race, and so is an atomic write concurrent with non-atomic reads or writes.

Whether a particular value should be accessed with atomics is therefore a property of the value and not of a particular access. Because of this, most languages put this information in the type system, like Java's `volatile int` and C++'s `atomic<int>`. Go's current APIs do not, meaning that correct usage requires careful annotation of which fields of a struct or global variables are expected to only be accessed using atomic APIs.

To improve program correctness, I am starting to think that Go should define a set of typed atomic values, analogous to the current `atomic.Value`: `Bool`, `Int`, `Uint`, `Int32`, `Uint32`, `Int64`, `Uint64`, and `Uintptr`. Like `Value`, these would have `CompareAndSwap`, `Load`, `Store`, and `Swap` methods. For example:

```
type Int32 struct { v int32 }

func (i *Int32) Add(delta int32) int32 {
    return AddInt32(&i.v, delta)
}

func (i *Int32) CompareAndSwap(old, new int32) (swapped bool) {
    return CompareAndSwapInt32(&i.v, old, new)
}

func (i *Int32) Load() int32 {
    return LoadInt32(&i.v)
}

func (i *Int32) Store(v int32) {
    return StoreInt32(&i.v, v)
}

func (i *Int32) Swap(new int32) (old int32) {
    return SwapInt32(&i.v, new)
}
```

I've included `Bool` on the list because we have constructed atomic booleans out of atomic integers multiple times in the Go standard library (in unexported APIs). There is clearly a need.

We could also take advantage of upcoming generics support and define an API for atomic pointers that is typed and free of package `unsafe` in its API:

```
type Pointer[T any] struct { v *T }

func (p *Pointer[T]) CompareAndSwap(old, new *T) (swapped bool) {
    return CompareAndSwapPointer(... lots of unsafe ...)
}
```

(And so on.) To answer an obvious suggestion, I don't see a clean way to use generics to provide just a single `atomic.Atomic[T]` that would let us avoid introducing `Bool`, `Int`, and so on as separate types, at least not without special cases in the compiler. And that's okay.

Maybe: Add unsynchronized atomics

All other modern programming languages provide a way to make concurrent memory reads and writes that do not synchronize the program but also don't invalidate it (don't count as a data race). C, C++, Rust, and Swift have relaxed atomics. Java has `VarHandle`'s "plain" mode. JavaScript has non-atomic accesses to the `SharedArrayBuffer` (the only shared memory). Go has no way to do this. Perhaps it should. I don't know.

If we wanted to add unsynchronized atomic reads and writes, we could add `UnsyncAdd`, `UnsyncCompareAndSwap`, `UnsyncLoad`, `UnsyncStore`, and `UnsyncSwap` methods to the typed atomics. Naming them "unsync" avoids a few problems with the name "relaxed." First, some people use relaxed as a relative comparison, as in "acquire/release is a more relaxed memory order than sequential consistency." You can argue that's not proper usage of the term, but it happens. Second, and more important, the critical detail about these operations is not the memory ordering of the operations themselves but the fact that they have *no effect* on the synchronization of the rest of the program. To people who are not experts in memory models, seeing `UnsyncLoad` should make clear that there is no synchronization, whereas `RelaxedLoad` probably would not. It's also nice that `Unsync` looks at a glance like `Unsafe`.

With the API out of the way, the real question is whether to add these at all. The usual argument for providing an unsynchronized atomic is that it really does matter for the performance of fast paths in certain data structures. My general impression is that it matters most on non-x86 architectures, although I don't have data to back this up. Not providing an unsynchronized atomic can be argued to penalize those architectures.

A possible argument against providing an unsynchronized atomic is that on x86, ignoring the effect of potential compiler reorderings, unsynchronized atomics are indistinguishable from acquire/release atomics. They might therefore be abused to write code that only works on x86. The counterargument is that such subterfuge would not pass muster with the race detector, which implements the actual memory model and not the x86 memory model.

With the lack of evidence we have today, we would not be justified in adding this API. If anyone feels strongly that we should add it, the way to make the case would be to gather evidence of both (1) general applicability in code that programmers need to write, and (2) significant performance improvements on widely used systems arising from using non-synchronizing atomics. (It would be fine to show this using programs in languages other than Go.)

Document disallowed compiler optimizations

The current memory model ends by giving examples of invalid programs. Since the memory model serves as a contract between the programmer and the compiler writers, we should add examples of invalid compiler optimizations. For example, we might add:

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 Go programs. In particular, a compiler must not introduce a data race in a race-free program. It must not allow a single read to observe multiple values. And it must not allow a single write to write multiple values.

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

```
i := 0
if cond {
    i = *p
}
```

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

```
i := *p
if !cond {
    i = 0
}
```

If `cond` is false and another goroutine is writing `*p`, then the original program is race-free but the rewritten program contains a race.

Not introducing data races also means not assuming that loops terminate. For example, a compiler must 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.

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 spill `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.

These categories and examples cover the most common C/C++ compiler optimizations that are incompatible with defined semantics for racing data accesses. They establish clearly that Go and C/C++ have different requirements.

Conclusion

Go's general approach of being conservative in its memory model has served us well and should be continued. There are, however, a few changes that are overdue, including defining the synchronization behavior of new APIs in the `sync` and `sync/atomic` packages. The atomics in particular should be documented to provide sequentially consistent behavior that creates happens-before edges synchronizing the non-atomic code around them. This would match the default atomics provided by all other modern systems languages.

Perhaps the most unique part of the update is the idea of clearly stating that programs with data races may be stopped to report the race but otherwise have well-defined semantics. This constrains both programmers and compilers, and it prioritizes the debuggability and correctness of concurrent programs over convenience for compiler writers.

Acknowledgements

This series of posts benefited greatly from discussions with and feedback from a long list of engineers I am lucky to work with at Google. My thanks to them. I take full responsibility for any mistakes or unpopular opinions.