Is variable assignment atomic in go?

Asked 8 years ago Modified 1 year, 1 month ago Viewed 7k times

14

If I have two threads concurrently modifying a string field on a struct, will I always see one or the other string assigned to the field, but nothing else?

go concurrency

Share Improve this question Follow

asked Jan 12, 2016 at 17:39

Filip Haglund
14.1k ● 15 ● 70 ● 119

Two goroutines (not threads) concurrently modifying anything is a data race, and there are no benign data races. Always check your concurrent coded with the race detector. – JimB Jan 12, 2016 at 17:42

I think the best way to use this is to use channels to modify the string. Have a goroutine listening on the channel for the change you want to do to the string, and where you are currently modifying the string just write the change you want to the channel. – Topo Jan 12, 2016 at 20:06

@Topo: no, that's not the "best way". There are often very good reasons to use mutexes or atomic operations. – JimB Jan 12, 2016 at 22:30

@JimB I know there are good reasons for using mutexes, my comment wasn't on the best way to do something like this, but on this specific example. – Topo Jan 13, 2016 at 1:22

2 Answers

Sorted by: Highest score (default)

20

No. If you need atomic operations, there is `sync/atomic`.

The Go Memory Model will have all the related details. From the top of the Memory Model document:

> 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.

Share Improve this answer Follow

edited Jan 12, 2016 at 18:27

answered Jan 12, 2016 at 17:40

JimB
106k ● 15 ● 269 ● 259

1 You didn't quote the next sentence: "If you must read the rest of this document to understand the behavior of your program, you are being too clever." The quoted section isn't describing the memory model, but suggesting the reader not be clever rather than rely on the details of the memory model. Is there anything in the document that actually says variable assignment is or is not atomic? If not, you haven't actually answered the question. – Phil Frost Sep 30, 2020 at 19:42

3 Assignment is not atomic, which is implied by the first sentence in the quoted documentation, and reinforced through the document as the conditions on observability are described. If that is unsatisfactory, I suggest you raise an issue with the go project for clarification in the documentation. – JimB Nov 3, 2020 at 15:40

5

As of today, the version on June 6, 2022 of the Go memory model guarantees that a memory access not larger than a machine word is atomic.

> Otherwise, a read r of a memory location x 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.

However, a string is definitely larger than a machine word, so your accesses are not guaranteed to be atomic. In this case, the result is unspecified, but it is most likely to be an interleave of different parts from different writes.

> 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.

Note that `sync/atomic` provides not only atomicity, but also sequential consistency. As a result, for accesses not larger than machine word, `sync/atomic` only provides additional sequential consistency.

> 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.

Share Improve this answer Follow

edited Dec 7, 2022 at 8:41

answered Dec 7, 2022 at 8:33

Quân Anh Mai

469 ● 4 ● 6

Can you explain how `string is definitely larger than a machine word` is relevant here? At the end of the day, string is an immutable slice pointer? – ahmet alp balkan Feb 23, 2023 at 20:50

2 @ahmetalpbalkan A string is not an immutable slice pointer. Although the language does not specified, the current implementation of a string in Go is a pointer to the payload and a field indicating the string length, which combined make an object double the size of a machine word. – Quân Anh Mai Feb 24, 2023 at 3:11

Go 语言标准库中 atomic.Value 的前世今生

2019-03-15 约 4420 字

在 Go 语言标准库中，sync/atomic包将底层硬件提供的原子操作封装成了 Go 的函数。但这些操作只支持几种基本数据类型，因此为了扩大原子操作的适用范围，Go 语言在 1.4 版本的时候向sync/atomic包中添加了一个新的类型Value。此类型的值相当于一个容器，可以被用来“原子地"存储（Store）和加载（Load）**任意类型**的值。

历史起源

我在golang-dev邮件列表中翻到了14年的这段讨论，有用户报告了encoding/gob包在多核机器上（80-core）上的性能问题，认为encoding/gob之所以不能完全利用到多核的特性是因为它里面使用了大量的互斥锁（mutex），如果把这些互斥锁换成用atomic.LoadPointer/StorePointer来做并发控制，那性能将能提升20倍。

针对这个问题，有人提议在已有的atomic包的基础上封装出一个atomic.Value类型，这样用户就可以在不依赖 Go 内部类型unsafe.Pointer的情况下使用到atomic提供的原子操作。所以我们现在看到的atomic包中除了atomic.Value外，其余都是早期由汇编写成的，并且atomic.Value类型的底层实现也是建立在已有的atomic包的基础上。

那为什么在上面的场景中，atomic会比mutex性能好很多呢？作者 Dmitry Vyukov 总结了这两者的一个区别：

> Mutexes do no scale. Atomic loads do.

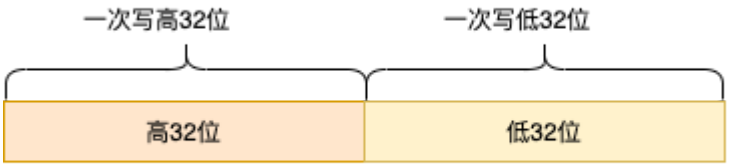
Mutex由**操作系统**实现，而atomic包中的原子操作则由**底层硬件**直接提供支持。在 CPU 实现的指令集里，有一些指令被封装进了atomic包，这些指令在执行的过程中是不允许中断（interrupt）的，因此原子操作可以在lock-free的情况下保证并发安全，并且它的性能也能做到随 CPU 个数的增多而线性扩展。

好了，说了这么多的原子操作，我们先来看看什么样的操作能被叫做*原子操作*。

原子性

一个或者多个操作在 CPU 执行的过程中不被中断的特性，称为*原子性（atomicity）*。这些操作对外表现成一个不可分割的整体，他们要么都执行，要么都不执行，外界不会看到他们只执行到一半的状态。而在现实世界中，CPU 不可能不中断的执行一系列操作，但如果我们在执行多个操作时，能让他们的**中间状态对外不可见**，那我们就可以宣称他们拥有了"不可分割”的原子性。

有些同学可能不知道，在 Go（甚至是大部分语言）中，一条普通的赋值语句其实不是一个原子操作。例如，在32位机器上写int64类型的变量就会有中间状态，因为它会被拆成两次写操作（MOV）——写低 32 位和写高 32 位，如下图所示：



如果一个线程刚写完低32位，还没来得及写高32位时，另一个线程读取了这个变量，那它得到的就是一个毫无逻辑的中间变量，这很有可能使我们的程序出现诡异的 Bug。

这还只是一个基础类型，如果我们对一个大小超过 Cache Line 的结构体进行赋值，那它出现并发问题的概率就更高了。很可能写线程刚写完一小半的字段，读线程就来读取这个变量，那么就只能读到仅修改了一部分的值。这显然破坏了变量的完整性，读出来的值也是完全错误的。

面对这种多线程下变量的读写问题，我们的主角——`atomic.Value`登场了，它使得我们可以不依赖于不保证兼容性的`unsafe.Pointer`类型，同时又能将任意数据类型的读写操作封装成原子性操作（让中间状态对外不可见）。

使用姿势

`atomic.Value`类型对外暴露的方法就两个：

- `v.Store(c)` - 写操作，将原始的变量`c`存放到一个`atomic.Value`类型的`v`里。
- `c = v.Load()` - 读操作，从线程安全的`v`中读取上一步存放的内容。

简洁的接口使得它的使用也很简单，只需将需要作并发保护的变量读取和赋值操作用`Load()`和`Store()`代替就行了。

下面是一个常见的使用场景：应用程序定期的从外界获取最新的配置信息，然后更改自己内存中维护的配置变量。工作线程根据最新的配置来处理请求。

```
 1 package main
 2
 3 import (
 4   "sync/atomic"
 5   "time"
 6 )
 7
 8 func loadConfig() map[string]string {
 9   // 从数据库或者文件系统中读取配置信息，然后以map的形式存放在内存里
10   return make(map[string]string)
11 }
12
13 func requests() chan int {
14   // 将从外界中接受到的请求放入到channel里
15   return make(chan int)
16 }
17
18 func main() {
19   // config变量用来存放该服务的配置信息
20   var config atomic.Value
21   // 初始化时从别的地方加载配置文件，并存到config变量里
22   config.Store(loadConfig())
23   go func() {
24     // 每10秒钟定时的拉取最新的配置信息，并且更新到config变量里
25     for {
```

```
26        time.Sleep(10 * time.Second)
27        // 对应于赋值操作 config = loadConfig()
28        config.Store(loadConfig())
29      }
30    }()
31    // 创建工作线程，每个工作线程都会根据它所读取到的最新的配置信息来处理请求
32    for i := 0; i < 10; i++ {
33      go func() {
34        for r := range requests() {
35          // 对应于取值操作 c := config
36          // 由于Load()返回的是一个interface{}类型，所以我们要先强制转换一下
37          c := config.Load().(map[string]string)
38          // 这里是根据配置信息处理请求的逻辑...
39          _, _ = r, c
40        }
41      }()
42    }
43  }
```

内部实现

罗永浩浩曾说过：

> Simplicity is the hidden complexity

我们来看看在简单的外表下，它到底有哪些 hidden complexity。

数据结构

`atomic.Value`被设计用来存储任意类型的数据，所以它内部的字段是一个`interface{}`类型，非常的简单粗暴。

```
1 type Value struct {
2   v interface{}
3 }
```

除了`Value`外，这个文件里还定义了一个`ifaceWords`类型，这其实是一个空interface (`interface{}`) 的内部表示格式（参见runtime/runtime2.go中eface的定义）。它的作用是将`interface{}`类型分解，得到其中的两个字段。

```
1 type ifaceWords struct {
2   typ  unsafe.Pointer
3   data unsafe.Pointer
4 }
```

写入（Store）操作

在介绍写入之前，我们先来看一下 Go 语言内部的`unsafe.Pointer`类型。

unsafe.Pointer

出于安全考虑，Go 语言并不支持直接操作内存，但它的标准库中又提供一种*不安全（不保证向后兼容性）* 的指针类型`unsafe.Pointer`，让程序可以灵活的操作内存。

unsafe.Pointer的特别之处在于，它可以绕过 Go 语言类型系统的检查，与任意的指针类型互相转换。也就是说，如果两种类型具有相同的内存结构（layout），我们可以将unsafe.Pointer当做桥梁，让这两种类型的指针相互转换，从而实现同一份内存拥有两种不同的**解读**方式。

比如说，[]byte和string其实内部的存储结构都是一样的，但 Go 语言的类型系统禁止他俩互换。如果借助unsafe.Pointer，我们就可以实现在零拷贝的情况下，将[]byte数组直接转换成string类型。

```
1 bytes := []byte{104, 101, 108, 108, 111}
2
3 p := unsafe.Pointer(&bytes) //强制转换成unsafe.Pointer，编译器不会报错
4 str := *(*string)(p) //然后强制转换成string类型的指针，再将这个指针的值当做string类型取出来
5 fmt.Println(str) //输出 "hello"
```

知道了unsafe.Pointer的作用，我们可以直接来看代码了：

```
 1 func (v *Value) Store(x interface{}) {
 2   if x == nil {
 3     panic("sync/atomic: store of nil value into Value")
 4   }
 5   vp := (*ifaceWords)(unsafe.Pointer(v))  // Old value
 6   xp := (*ifaceWords)(unsafe.Pointer(&x)) // New value
 7   for {
 8     typ := LoadPointer(&vp.typ)
 9     if typ == nil {
10       // Attempt to start first store.
11       // Disable preemption so that other goroutines can use
12       // active spin wait to wait for completion; and so that
13       // GC does not see the fake type accidentally.
14       runtime_procPin()
15       if !CompareAndSwapPointer(&vp.typ, nil, unsafe.Pointer(^uintptr(0))) {
16         runtime_procUnpin()
17         continue
18       }
19       // Complete first store.
20       StorePointer(&vp.data, xp.data)
21       StorePointer(&vp.typ, xp.typ)
22       runtime_procUnpin()
23       return
24     }
25     if uintptr(typ) == ^uintptr(0) {
26       // First store in progress. Wait.
27       // Since we disable preemption around the first store,
28       // we can wait with active spinning.
29       continue
30     }
31     // First store completed. Check type and overwrite data.
32     if typ != xp.typ {
33       panic("sync/atomic: store of inconsistently typed value into Value")
34     }
35     StorePointer(&vp.data, xp.data)
36     return
37   }
38 }
```

大概的逻辑：

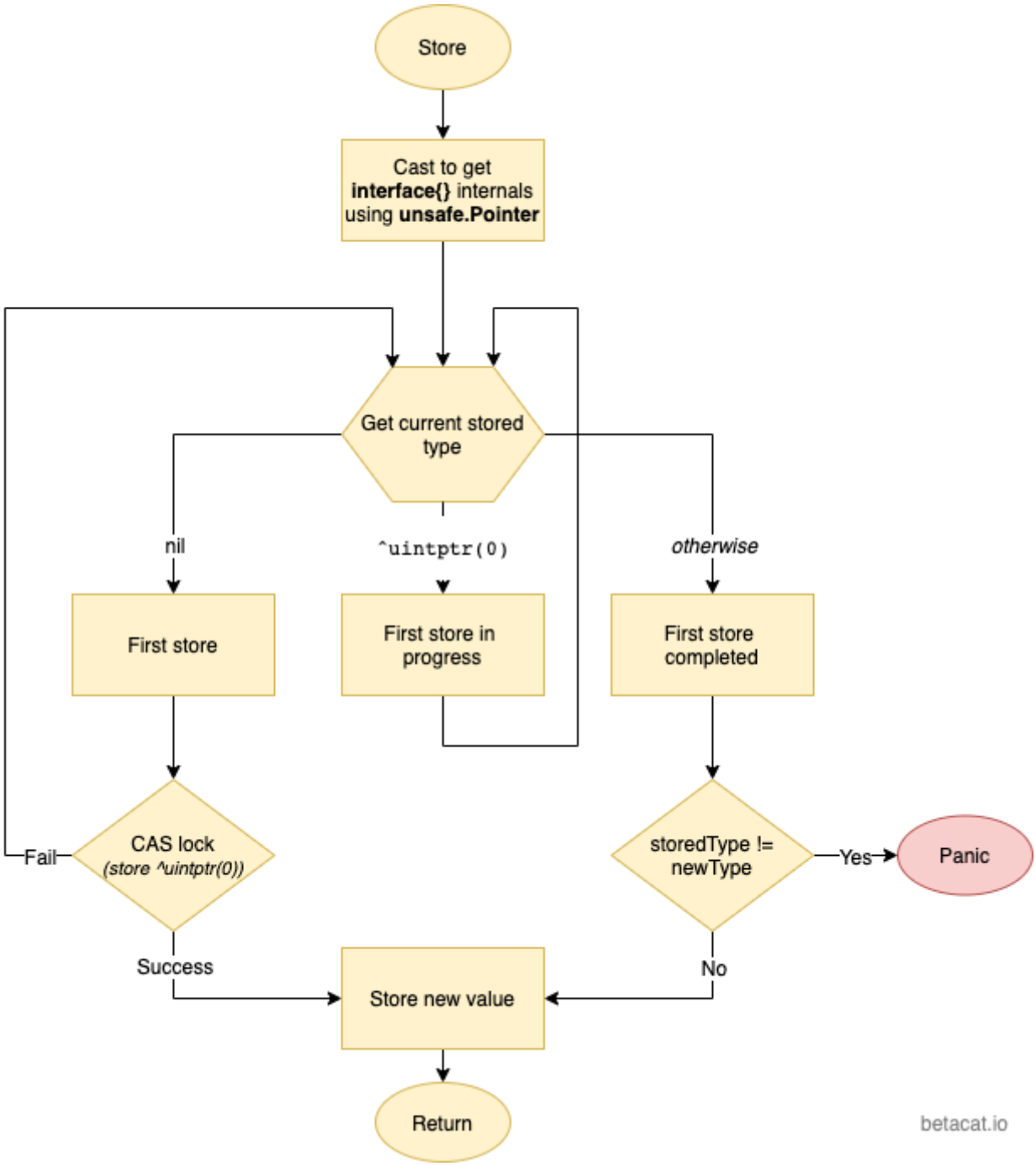
- 第5~6行 - 通过unsafe.Pointer将**现有的**和**要写入的**值分别转成ifaceWords类型，这样我们下一步就可以得到这两个interface{}的原始类型（typ）和真正的值（data）。
- 从第7行开始就是一个无限 for 循环。配合CompareAndSwap食用，可以达到乐观锁的功效。

- 第8行，我们可以通过LoadPointer这个原子操作拿到当前Value中存储的类型。下面根据这个类型的不同，分3种情况处理。

1. 第一次写入（第9~24行） - 一个Value实例被初始化后，它的typ字段会被设置为指针的零值nil，所以第9行先判断如果typ是 nil 那就证明这个Value还未被写入过数据。那之后就是一段初始写入的操作：
    - runtime_procPin()这是runtime中的一段函数，具体的功能我不是特别清楚，也没有找到相关的文档。这里猜测一下，一方面它禁止了调度器对当前 goroutine 的抢占（preemption），使得它在执行当前逻辑的时候不被打断，以便可以尽快地完成工作，因为别人一直在等待它。另一方面，在禁止抢占期间，GC 线程也无法被启用，这样可以防止 GC 线程看到一个莫名其妙的指向^uintptr(0)的类型（这是赋值过程中的中间状态）。
    - 使用CAS操作，先尝试将typ设置为^uintptr(0)这个中间状态。如果失败，则证明已经有别的线程抢先完成了赋值操作，那它就解除抢占锁，然后重新回到 for 循环第一步。
    - 如果设置成功，那证明当前线程抢到了这个"乐观锁"，它可以安全的把v设为传入的新值了（19~23行）。注意，这里是先写data字段，然后再写typ字段。因为我们是以typ字段的值作为写入完成与否的判断依据的。
2. 第一次写入还未完成（第25~30行）- 如果看到 typ字段还是^uintptr(0)这个中间类型，证明刚刚的第一次写入还没有完成，所以它会继续循环， “忙等"到第一次写入完成。
3. 第一次写入已完成（第31行及之后） - 首先检查上一次写入的类型与这一次要写入的类型是否一致，如果不一致则抛出异常。反之，则直接把这一次要写入的值写入到data字段。

这个逻辑的主要思想就是，为了完成多个字段的原子性写入，我们可以抓住其中的一个字段，以它的状态来标志整个原子写入的状态。这个想法我在 TiDB 的事务实现中看到过类似的，他们那边叫Percolator模型，主要思想也是先选出一个primaryRow，然后所有的操作也是以primaryRow的成功与否作为标志。嗯，果然是太阳底下没有新东西。

如果没有耐心看代码，没关系，这儿还有个简化版的流程图：



读取（Load）操作

先上代码：

```
1 func (v *Value) Load() (x interface{}) {
2   vp := (*ifaceWords)(unsafe.Pointer(v))
3   typ := LoadPointer(&vp.typ)
4   if typ == nil || uintptr(typ) == ^uintptr(0) {
5     // First store not yet completed.
6     return nil
7   }
8   data := LoadPointer(&vp.data)
9   xp := (*ifaceWords)(unsafe.Pointer(&x))
10  xp.typ = typ
11  xp.data = data
12  return
13 }
```

读取相对就简单很多了，它有两个分支：

1. 如果当前的`typ`是 nil 或者`^uintptr(0)`，那就证明第一次写入还没有开始，或者还没完成，那就直接返回 nil （不对外暴露中间状态）。
2. 否则，根据当前看到的`typ`和`data`构造出一个新的`interface{}`返回出去。

总结

本文从邮件列表中的一段讨论开始，介绍了`atomic.Value`的被提出来的历史缘由。然后由浅入深的介绍了它的使用姿势，以及内部实现。让大家不仅知其然，还能知其所以然。

另外，再强调一遍，原子操作由**底层硬件**支持，而锁则由操作系统的**调度器**实现。锁应当用来保护一段逻辑，对于一个变量更新的保护，原子操作通常会更有效率，并且更能利用计算机多核的优势，如果要更新的是一个复合对象，则应当使用`atomic.Value`封装好的实现。

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.

First, 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.

Otherwise, a read *r* of a memory location *x* 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.