The Go Memory Model

Version of May 31, 2014

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.

Happens Before

Within a single goroutine, reads and writes must behave as if they executed in the order specified by the program. That is, compilers and processors may reorder the reads and writes executed within a single goroutine only when the reordering does not change the behavior within that goroutine as defined by the language specification. Because of this reordering, the execution order observed by one goroutine may differ from the order perceived by another. For example, if one goroutine executes `a = 1; b = 2;`, another might observe the updated value of `b` before the updated value of `a`.

To specify the requirements of reads and writes, we define *happens before*, a partial order on the execution of memory operations in a Go program. If event e_1 happens before event e_2, then we say

execution of memory operations in a Go program. If event e_1 happens before event e_2, then we say that e_2 happens after e_1. Also, if e_1 does not happen before e_2 and does not happen after e_2, then we say that e_1 and e_2 happen concurrently.

Within a single goroutine, the happens-before order is the order expressed by the program.

A read *r* of a variable v is *allowed* to observe a write *w* to v if both of the following hold:

1. *r* does not happen before *w*.
2. There is no other write *w'* to v that happens after *w* but before *r*.

To guarantee that a read *r* of a variable v observes a particular write *w* to v, ensure that *w* is the only write *r* is allowed to observe. That is, *r* is *guaranteed* to observe *w* if both of the following hold:

1. *w* happens before *r*.
2. Any other write to the shared variable v either happens before *w* or after *r*.

This pair of conditions is stronger than the first pair; it requires that there are no other writes happening concurrently with *w* or *r*.

Within a single goroutine, there is no concurrency, so the two definitions are equivalent: a read *r* observes the value written by the most recent write *w* to v. When multiple goroutines access a shared variable v, they must use synchronization events to establish happens-before conditions that ensure reads observe the desired writes.

The initialization of variable v with the zero value for v's type behaves as a write in the memory model.

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

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 start of the function `main.main` happens after all `init` functions have finished.

Goroutine creation

The `go` statement that starts a new goroutine happens before the goroutine's execution begins.

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 happen 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 happens before the corresponding receive from that channel completes.

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 happens before the send on `c`, which happens before the corresponding receive on `c` completes, which happens before the `print`.

The closing of a channel happens 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 happens before the send on that channel completes.

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 happens before the receive on `c`, which happens before the corresponding send on `c` completes, which happens 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 happens before 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()` happens 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) happens before the second call to l.Lock() (in main) returns, which happens before the print.

For any call to l.RLock on a sync.RWMutex variable l, there is an n such that the l.RLock happens (returns) after call n to l.Unlock and the matching l.RUnlock happens before call n+1 to l.Lock.

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.

A single call of f() from once.Do(f) happens (returns) before any call of once.Do(f) returns.

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.

Incorrect synchronization

Note that a read *r* may observe the value written by a write *w* that happens 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:

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.