Data Race Detector

Table of Contents

Introduction

Data races are among the most common and hardest to debug types of bugs in concurrent systems. A data race occurs when two goroutines access the same variable concurrently and at least one of the accesses is a write. See the The Go Memory Model for details.

Here is an example of a data race that can lead to crashes and memory corruption:

```
func main() {
	c := make(chan bool)
	m := make(map[string]string)
	go func() {
		m["1"] = "a" // First conflicting access.
		c <- true
	}()
	m["2"] = "b" // Second conflicting access.
	<-c
	for k, v := range m {
		fmt.Println(k, v)
	}
}
```

Usage

To help diagnose such bugs, Go includes a built-in data race detector. To use it, add the `-race` flag to the go command:

```
$ go test -race mypkg    // to test the package
$ go run -race mysrc.go  // to run the source file
$ go build -race mycmd   // to build the command
$ go install -race mypkg // to install the package
```

Report Format

When the race detector finds a data race in the program, it prints a report. The report contains stack traces for conflicting accesses, as well as stacks where the involved goroutines were created. Here is an example:

```
WARNING: DATA RACE
Read by goroutine 185:
  net.(*pollServer).AddFD()
      src/net/fd_unix.go:89 +0x398
  net.(*pollServer).WaitWrite()
      src/net/fd_unix.go:247 +0x45
  net.(*netFD).Write()
      src/net/fd_unix.go:540 +0x4d4
  net.(*conn).Write()
      src/net/net.go:129 +0x101
  net.func·060()
      src/net/timeout_test.go:603 +0xaf

Previous write by goroutine 184:
  net.setWriteDeadline()
      src/net/sockopt_posix.go:135 +0xdf
  net.setDeadline()
      src/net/sockopt_posix.go:144 +0x9c
  net.(*conn).SetDeadline()
      src/net/net.go:161 +0xe3
  net.func·061()
      src/net/timeout_test.go:616 +0x3ed

Goroutine 185 (running) created at:
  net.func·061()
      src/net/timeout_test.go:609 +0x288

Goroutine 184 (running) created at:
  net.TestProlongTimeout()
      src/net/timeout_test.go:618 +0x298
  testing.tRunner()
      src/testing/testing.go:301 +0xe8
```

Options

The `GORACE` environment variable sets race detector options. The format is:

```
GORACE="option1=val1 option2=val2"
```

The options are:

- `log_path` (default `stderr`): The race detector writes its report to a file named `log_path.`*pid*. The special names `stdout` and `stderr` cause reports to be written to standard output and standard error, respectively.

- exitcode (default 66): The exit status to use when exiting after a detected race.
- strip_path_prefix (default ""): Strip this prefix from all reported file paths, to make reports more concise.
- history_size (default 1): The per-goroutine memory access history is 32K * 2**history_size elements. Increasing this value can avoid a "failed to restore the stack" error in reports, at the cost of increased memory usage.
- halt_on_error (default 0): Controls whether the program exits after reporting first data race.
- atexit_sleep_ms (default 1000): Amount of milliseconds to sleep in the main goroutine before exiting.

Example:

```
$ GORACE="log_path=/tmp/race/report strip_path_prefix=/my/go/sources/" go test -
```

Excluding Tests

When you build with -race flag, the go command defines additional build tag race. You can use the tag to exclude some code and tests when running the race detector. Some examples:

```
// +build !race

package foo

// The test contains a data race. See issue 123.
func TestFoo(t *testing.T) {
        // ...
}

// The test fails under the race detector due to timeouts.
func TestBar(t *testing.T) {
        // ...
}

// The test takes too long under the race detector.
func TestBaz(t *testing.T) {
        // ...
}
```

How To Use

To start, run your tests using the race detector (go test -race). The race detector only finds races that happen at runtime, so it can't find races in code paths that are not executed. If your tests have incomplete coverage, you may find more races by running a binary built with -race under a realistic workload.

Typical Data Races

Here are some typical data races. All of them can be detected with the race detector.

Race on loop counter

```
func main() {
	var wg sync.WaitGroup
	wg.Add(5)
	for i := 0; i < 5; i++ {
		go func() {
			fmt.Println(i) // Not the 'i' you are looking for.
			wg.Done()
		}()
	}
	wg.Wait()
}
```

The variable `i` in the function literal is the same variable used by the loop, so the read in the goroutine races with the loop increment. (This program typically prints 55555, not 01234.) The program can be fixed by making a copy of the variable:

```
func main() {
	var wg sync.WaitGroup
	wg.Add(5)
	for i := 0; i < 5; i++ {
		go func(j int) {
			fmt.Println(j) // Good. Read local copy of the loop coun
			wg.Done()
		}(i)
	}
	wg.Wait()
}
```

Accidentally shared variable

```
// ParallelWrite writes data to file1 and file2, returns the errors.
func ParallelWrite(data []byte) chan error {
	res := make(chan error, 2)
	f1, err := os.Create("file1")
	if err != nil {
		res <- err
	} else {
		go func() {
			// This err is shared with the main goroutine,
			// so the write races with the write below.
			_, err = f1.Write(data)
			res <- err
			f1.Close()
		}()
```

```
	}
	f2, err := os.Create("file2") // The second conflicting write to err.
	if err != nil {
		res <- err
	} else {
		go func() {
			_, err = f2.Write(data)
			res <- err
			f2.Close()
		}()
	}
	return res
}
```

The fix is to introduce new variables in the goroutines (note the use of `:=`):

```
			...
			_, err := f1.Write(data)
			...
			_, err := f2.Write(data)
			...
```

Unprotected global variable

If the following code is called from several goroutines, it leads to races on the `service` map. Concurrent reads and writes of the same map are not safe:

```
var service map[string]net.Addr

func RegisterService(name string, addr net.Addr) {
	service[name] = addr
}

func LookupService(name string) net.Addr {
	return service[name]
}
```

To make the code safe, protect the accesses with a mutex:

```
var (
	service   map[string]net.Addr
	serviceMu sync.Mutex
)

func RegisterService(name string, addr net.Addr) {
	serviceMu.Lock()
	defer serviceMu.Unlock()
	service[name] = addr
}

func LookupService(name string) net.Addr {
```

```
	serviceMu.Lock()
	defer serviceMu.Unlock()
	return service[name]
}
```

Primitive unprotected variable

Data races can happen on variables of primitive types as well (`bool`, `int`, `int64`, etc.), as in this example:

```
type Watchdog struct{ last int64 }

func (w *Watchdog) KeepAlive() {
	w.last = time.Now().UnixNano() // First conflicting access.
}

func (w *Watchdog) Start() {
	go func() {
		for {
			time.Sleep(time.Second)
			// Second conflicting access.
			if w.last < time.Now().Add(-10*time.Second).UnixNano() {
				fmt.Println("No keepalives for 10 seconds. Dying
				os.Exit(1)
			}
		}
	}()
}
```

Even such "innocent" data races can lead to hard-to-debug problems caused by non-atomicity of the memory accesses, interference with compiler optimizations, or reordering issues accessing processor memory .

A typical fix for this race is to use a channel or a mutex. To preserve the lock-free behavior, one can also use the `sync/atomic` package.

```
type Watchdog struct{ last int64 }

func (w *Watchdog) KeepAlive() {
	atomic.StoreInt64(&w.last, time.Now().UnixNano())
}

func (w *Watchdog) Start() {
	go func() {
		for {
			time.Sleep(time.Second)
			if atomic.LoadInt64(&w.last) < time.Now().Add(-10*time.S
				fmt.Println("No keepalives for 10 seconds. Dying
				os.Exit(1)
			}
		}
```

```
        }()
}
```

Unsynchronized send and close operations

As this example demonstrates, unsynchronized send and close operations on the same channel can also be a race condition:

```
c := make(chan struct{}) // or buffered channel

// The race detector cannot derive the happens before relation
// for the following send and close operations. These two operations
// are unsynchronized and happen concurrently.
go func() { c <- struct{}{} }()
close(c)
```

According to the Go memory model, a send on a channel happens before the corresponding receive from that channel completes. To synchronize send and close operations, use a receive operation that guarantees the send is done before the close:

```
c := make(chan struct{}) // or buffered channel

go func() { c <- struct{}{} }()
<-c
close(c)
```

Requirements

The race detector requires cgo to be enabled, and on non-Darwin systems requires an installed C compiler. The race detector supports `linux/amd64`, `linux/ppc64le`, `linux/arm64`, `freebsd/amd64`, `netbsd/amd64`, `darwin/amd64`, `darwin/arm64`, and `windows/amd64`.

On Windows, the race detector runtime is sensitive to the version of the C compiler installed; as of Go 1.21, building a program with `-race` requires a C compiler that incorporates version 8 or later of the `mingw-w64` runtime libraries. You can test your C compiler by invoking it with the arguments `--print-file-name libsynchronization.a`. A newer compliant C compiler will print a full path for this library, whereas older C compilers will just echo the argument.

Runtime Overhead

The cost of race detection varies by program, but for a typical program, memory usage may increase by 5-10x and execution time by 2-20x.

The race detector currently allocates an extra 8 bytes per `defer` and `recover` statement. Those extra allocations are not recovered until the goroutine exits. This means that if you have a long-running goroutine that is periodically issuing `defer` and `recover` calls, the program memory usage may grow without bound. These memory allocations will not show up in the output of `runtime.ReadMemStats` or `runtime/pprof`.