Dynamic Data Race Detection

Aditya Thakur

Data race

Two threads (goroutines) concurrently access a shared memory location and at least one of the accesses is a Write (i.e. updates the value).

Data race - Example

```
var g int

func update() {
    g = g + 2
}

func main() {
    go update()
    go update()
}
```

Using mutex to synchronize

```
var g int
var mu sync.Mutex

func update() {
    mu.Lock()
    g = g + 2
    mu.Unlock()
}

func main() {

    go update()
    go update()
}
```

Using channels to synchronize

```
var g int
var ch chan int

func update() {
    ch <- 1
    g = g + 2
    <- ch
}

func main() {
    ch = make(chan int, 1)
    go update()
    go update()
}
```

Impact of data races

- Can be harmless
  - Do not lead to any real errors in the program
- Can cause the program to crash, produce incorrect answers, or worse
  - Data race in Therac-25 medical electron accelerator lead to loss of lives
  - Northeast blackout of 2003 caused by data race

Data race detection

Can we automatically detect data races in programs?

Two approaches

- Static program analysis
  - Automated techniques to analyze the source code
  - Akin to finding bugs by staring at the source
  - Able to prove the absence of bugs; reported bugs might be false positives
- Dynamic program analysis
  - Automated techniques to analyze trace of events generated by executing the code
  - Akin to finding bugs by inserting print statements
  - Unable to prove the absence of bugs; reports fewer false positives

Dynamic data race detection

Identify races by observing the **trace of events**

For each goroutine, sequence of

- reads/writes for each memory location
- locks/unlocks for each mutex
- Sends/receives for each channel

Dynamic data race detection

- Two approaches
  - Happens-before analysis
  - Lockset analysis

- The golang race detector uses a hybrid approach combining happens-before and lockset analysis

Happens-before analysis

- Happens-before is a partial order of all events of all threads in a concurrent execution.
- **Event X happens-before event Y** if
  X has been observed before Y,
  and
  - Events X and Y are in the same thread, OR
  - X is an Unlock() and Y is a Lock() for the same mutex OR
  - X is a send and Y is a receive for the same channel OR
  - There exists an event U such that X happens-before U, U happens-before Y

Data races and Happens-before

If two threads (or goroutines) access a shared variables,
and the accesses are *not* ordered by the *happens-before relation,*
then a data race **could have occurred**.

Detecting data race using happens-before

```
var g int

func update() {
    g = g + 2
}

func main() {
    go update()
    go update()
}
```

T1

T2

g = g+ 2

g = g+2

time

Event trace

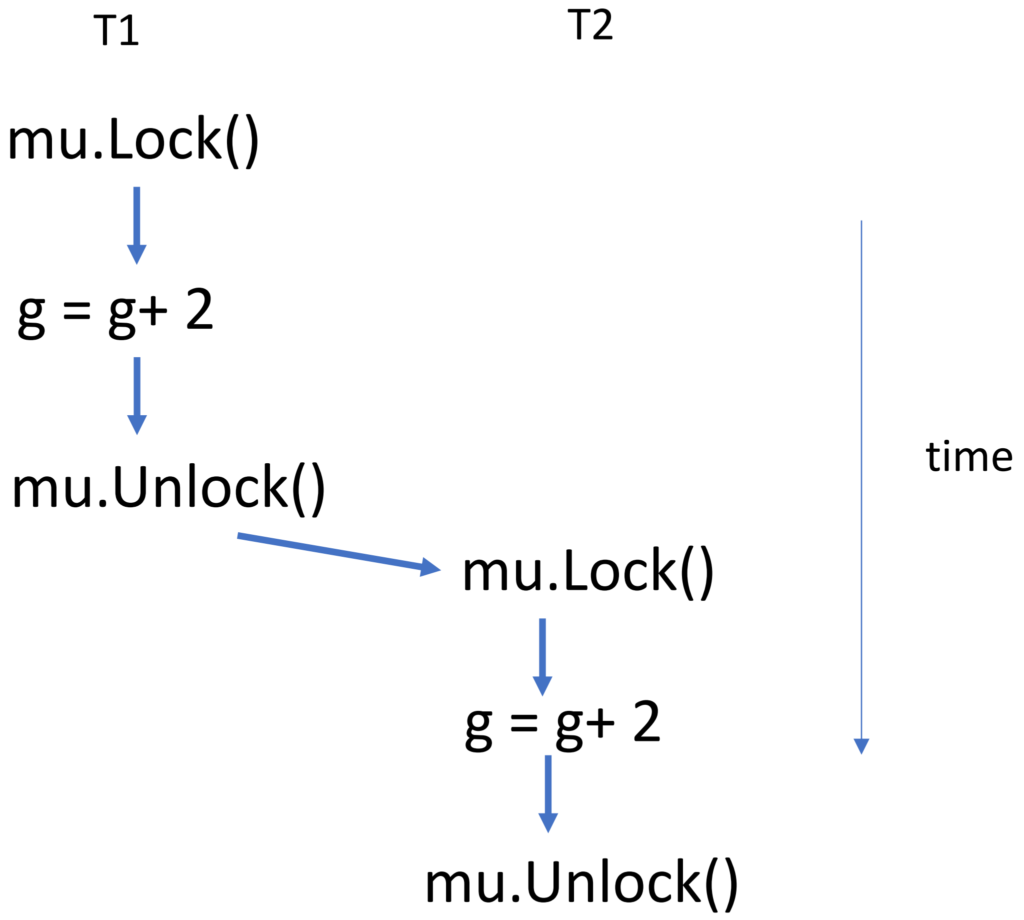
Using mutex to synchronize

```
var g int
var mu sync.Mutex

func update() {
    mu.Lock()
    g = g + 2
    mu.Unlock()
}

func main() {

    go update()
    go update()
}
```



Event trace

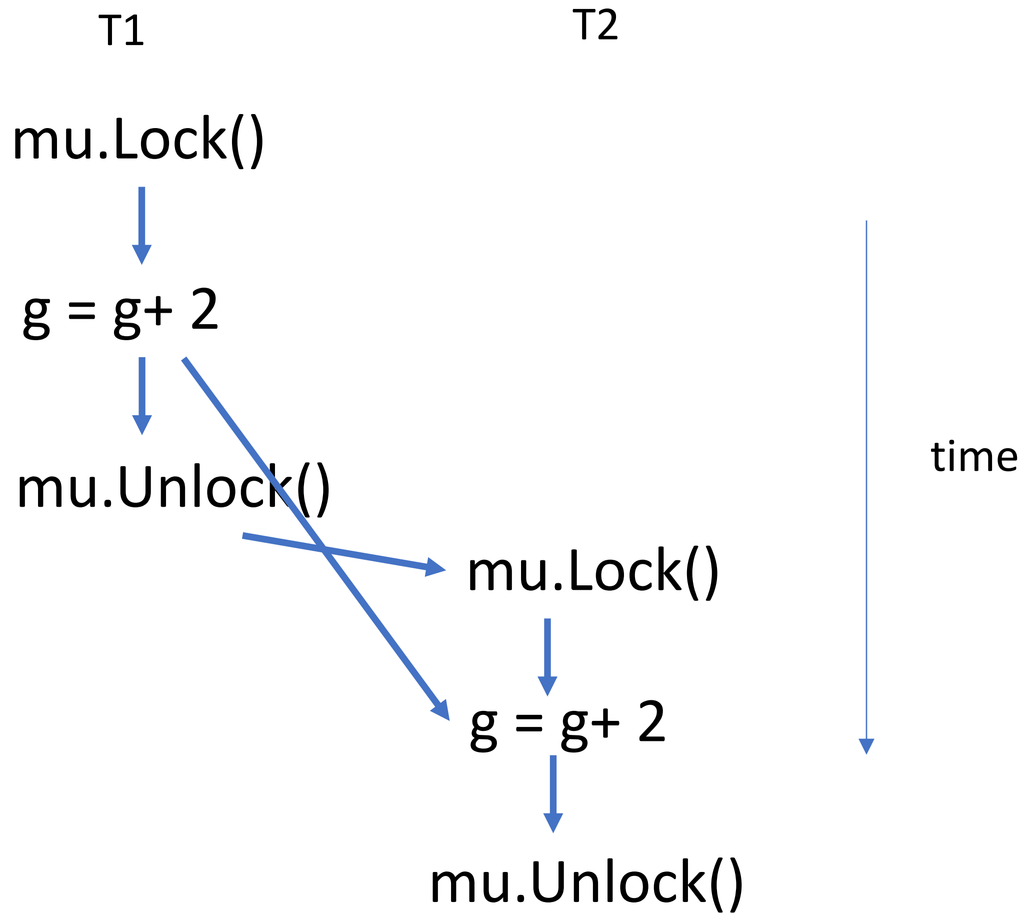
Using mutex to synchronize

```
var g int
var mu sync.Mutex

func update() {
    mu.Lock()
    g = g + 2
    mu.Unlock()
}

func main() {

    go update()
    go update()
}
```



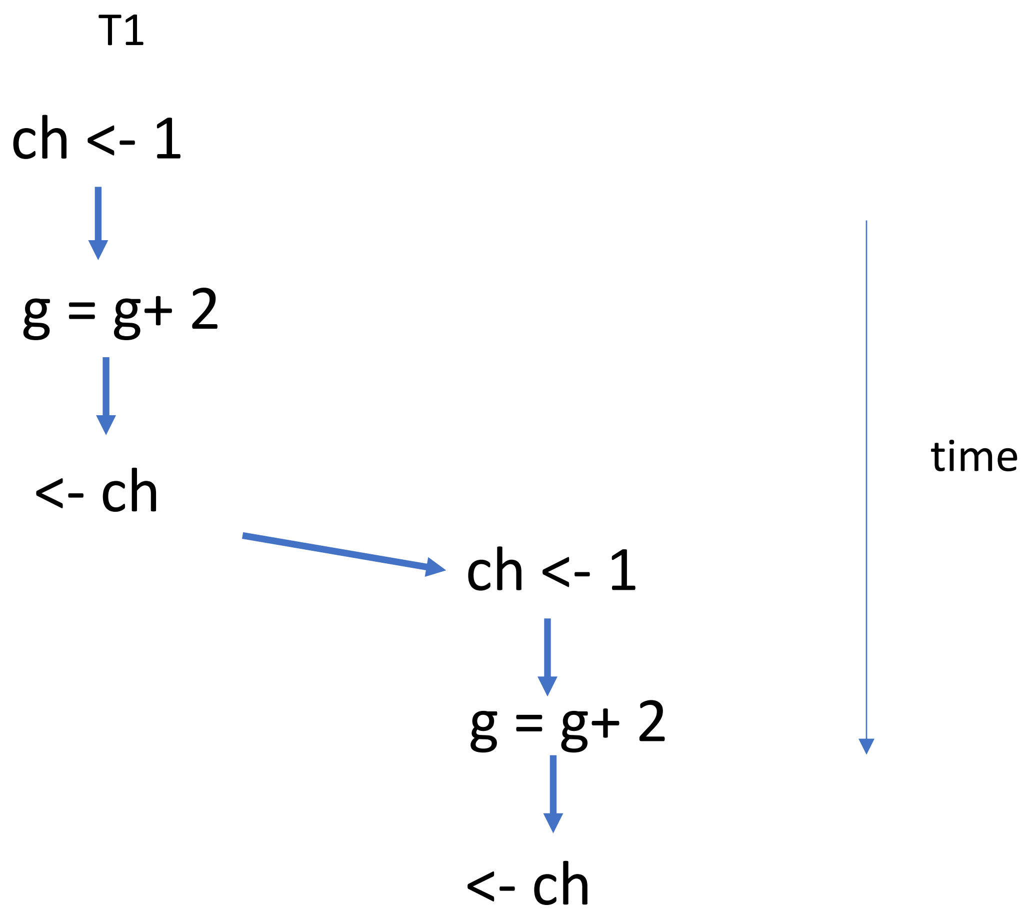
Event trace

Using channels to synchronize

```
var g int
var ch chan int

func update() {
    ch <- 1
    g = g + 2
    <- ch
}

func main() {
    ch = make(chan int, 1)
    go update()
    go update()
}
```

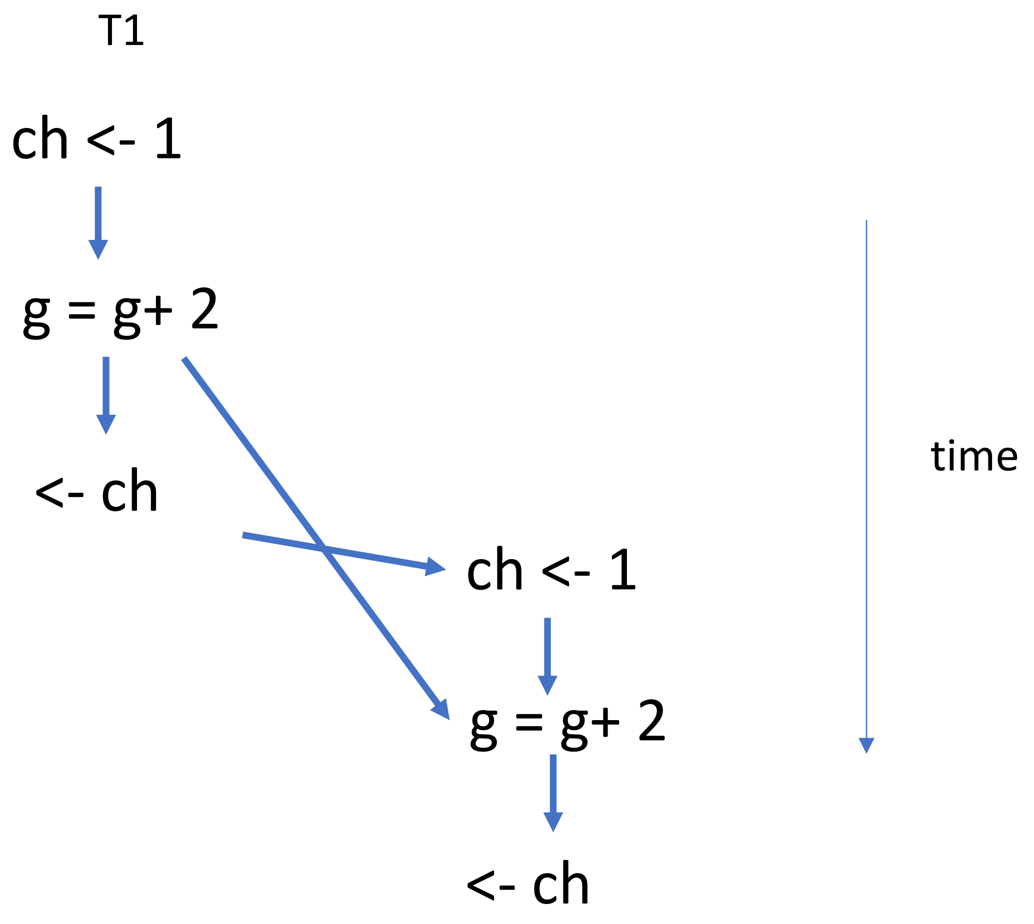


Using channels to synchronize

```
var g int
var ch chan int

func update() {
     ch <- 1
     g = g + 2
     <- ch
}

func main() {
     ch = make(chan int, 1)
     go update()
     go update()
}
```



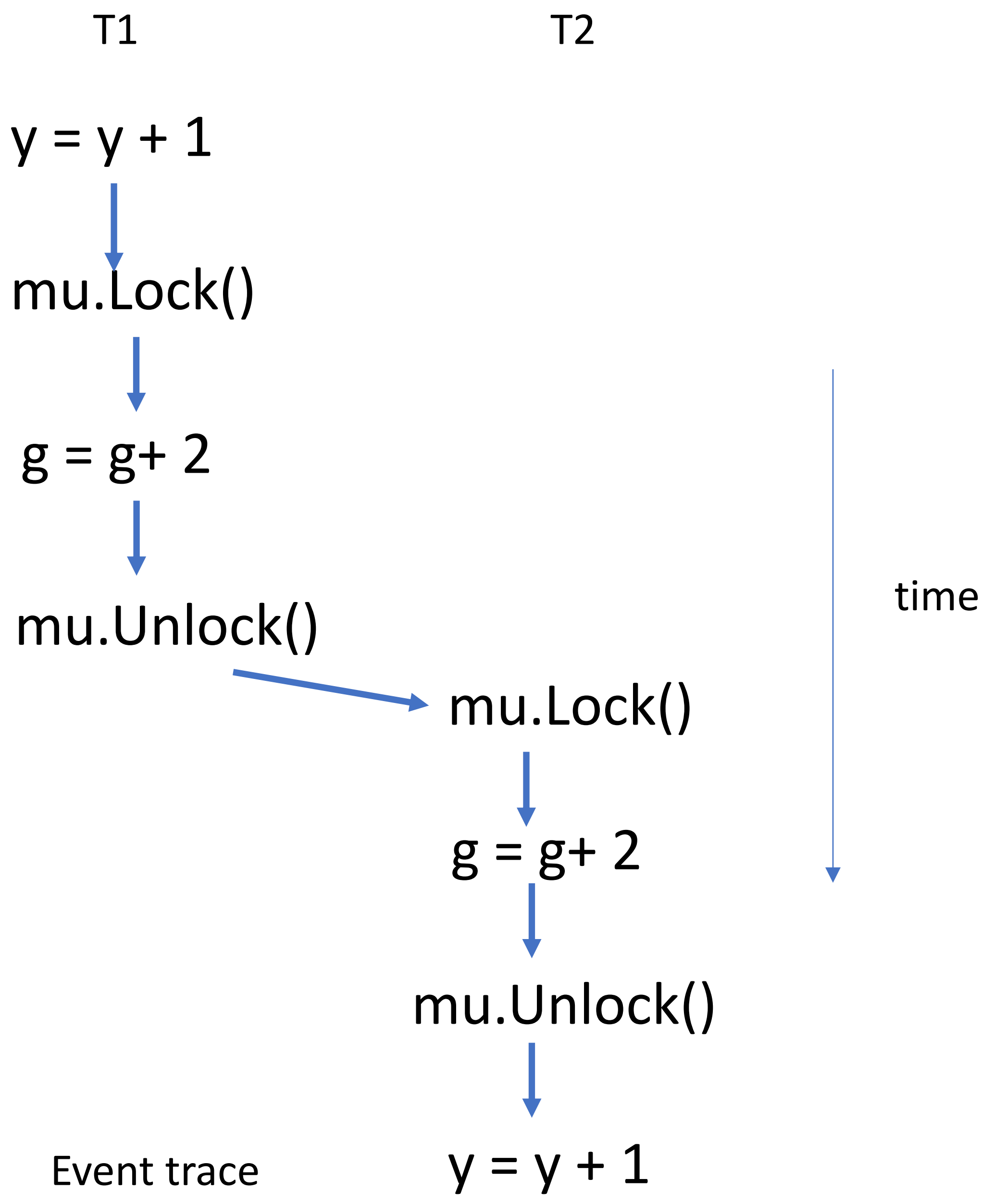
```
var g, y int
var mu sync.Mutex

func update1() {
    y = y + 1
    mu.Lock()
    g = g + 2
    mu.Unlock()
}

func update2() {
    mu.Lock()
    g = g + 2
    mu.Unlock()
    y = y + 1
}

func main() {

    go update1()
    go update2()
}
```



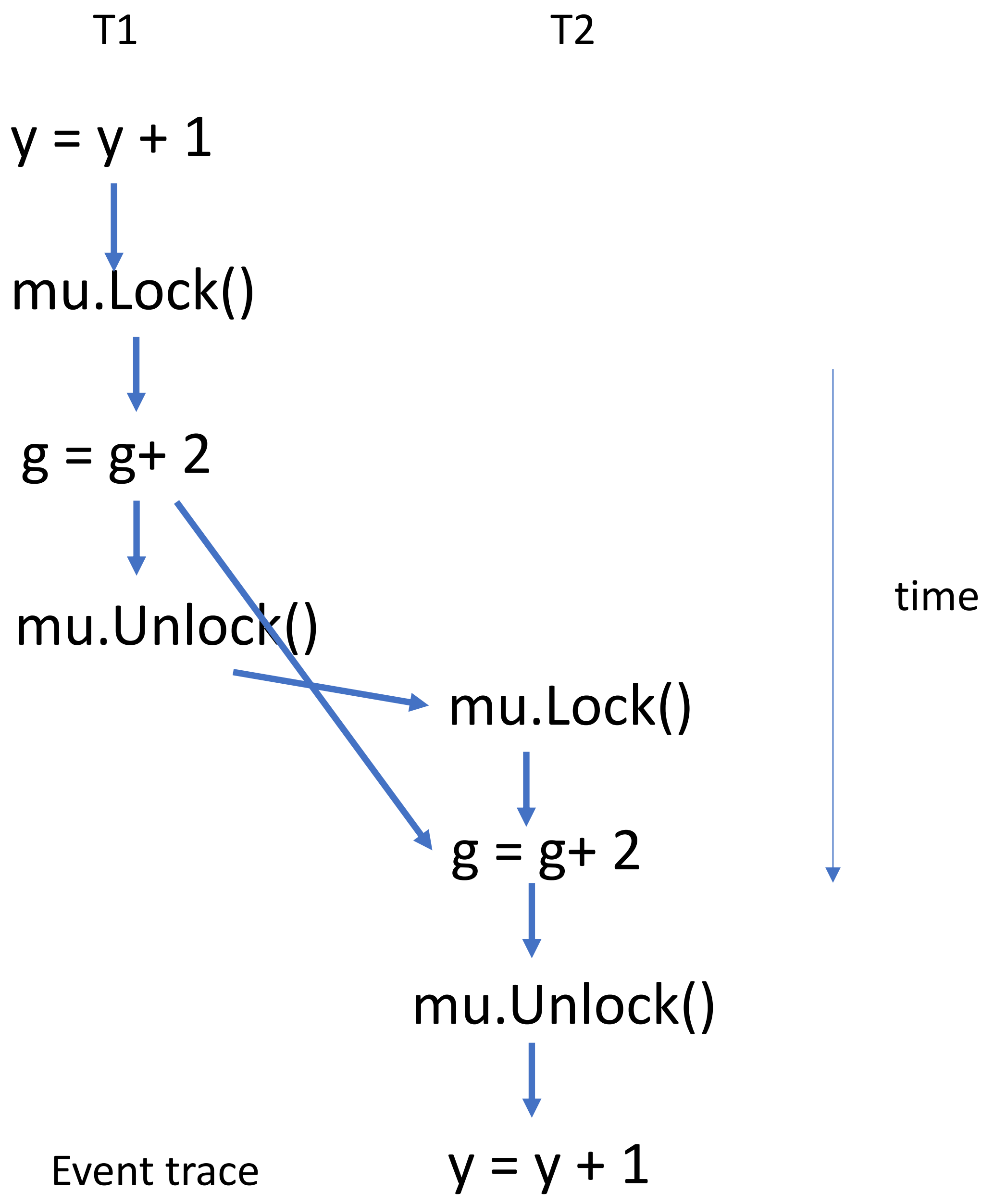
```go
var g, y int
var mu sync.Mutex

func update1() {
    y = y + 1
    mu.Lock()
    g = g + 2
    mu.Unlock()
}

func update2() {
    mu.Lock()
    g = g + 2
    mu.Unlock()
    y = y + 1
}

func main() {

    go update1()
    go update2()
}
```



Lockset Analysis

- Keep track of which locks are used when accessing shared variables
- Report warning if no lock is consistently used when accessing a shared variable v

Lockset Algorithm

- Let $locks\_held(t)$ be the set of locks held by thread t
- For each v, initialize $C(v)$ to the set of all locks.
- On each access to v by thread t,
  - $C(v) := C(v) \cap locks\_held(t)$
  - If $C(v) = \emptyset$, then issue warning

What are the locksets for g and y ?

```
var g, y int
var mu sync.Mutex

func update1() {
    y = y + 1
    mu.Lock()
    g = g + 2
    mu.Unlock()
}

func update2() {
    mu.Lock()
    g = g + 2
    mu.Unlock()
    y = y + 1
}

func main() {

    go update1()
    go update2()
}
```

T1	T2
y = y + 1	
mu.Lock()	
g = g+ 2	
mu.Unlock()	
	mu.Lock()
	g = g+ 2
	mu.Unlock()
	y = y + 1

time

Event trace

What is the lockset for g?

```go
var g int
var mu1, mu2, mu3 sync.Mutex
```

```go
func update1() {
    mu1.Lock()
    mu2.Lock()
    g = g + 2
    mu2.Unlock()
    mu1.Unlock()
}
```

```go
func update2() {
    mu2.Lock()
    mu3.Lock()
    g = g + 2
    mu3.Unlock()
    mu2.Unlock()
}
```

```go
func update3() {
    mu1.Lock()
    mu3.Lock()
    g = g + 2
    mu3.Unlock()
    mu1.Unlock()
}
```

```go
func main() {

    go update1()
    go update2()
    go update3()
}
```

Is there a race?

What about channels?

```
var g int
var ch chan int

func update() {
    ch <- 1
    g = g + 2
    <- ch
}

func main() {
    ch = make(chan int, 1)
    go update()
    go update()
}
```

T1

ch <- 1

g = g+ 2

<- ch

ch <- 1

g = g+ 2

<- ch

time

Golang data race detector

- The golang race detector uses a hybrid approach combining happens-before and lockset analysis

References

- Lamport, Leslie. "Time, clocks, and the ordering of events in a distributed system." *Communications of the ACM* 21.7 (1978): 558-565.
- Savage, Stefan, et al. "Eraser: A dynamic data race detector for multithreaded programs." *ACM Transactions on Computer Systems (TOCS)* 15.4 (1997): 391-411.
- Serebryany, Konstantin, and Timur Iskhodzhanov. "ThreadSanitizer: data race detection in practice." *Proceedings of the Workshop on Binary Instrumentation and Applications*. ACM, 2009.