Looking Inside a Race Detector

kavya

@kavya719

data race detection

data races

"when two+ threads concurrently access a shared memory location, at least one access is a write."

data race

```
// Shared variable
var count = 0

func incrementCount() {
  if count == 0 {
    count ++
  }
}

func main() {
  // Spawn two "threads"
  go incrementCount()
  go incrementCount()
}
```

"g1"

"g2"

R	R	R
W	R	R
R	W	W
!W	W	W
count = 1	count = 2	count = 2
!concurrent	concurrent	concurrent

data races

"when two+ threads concurrently access a shared memory location, at least one access is a write."

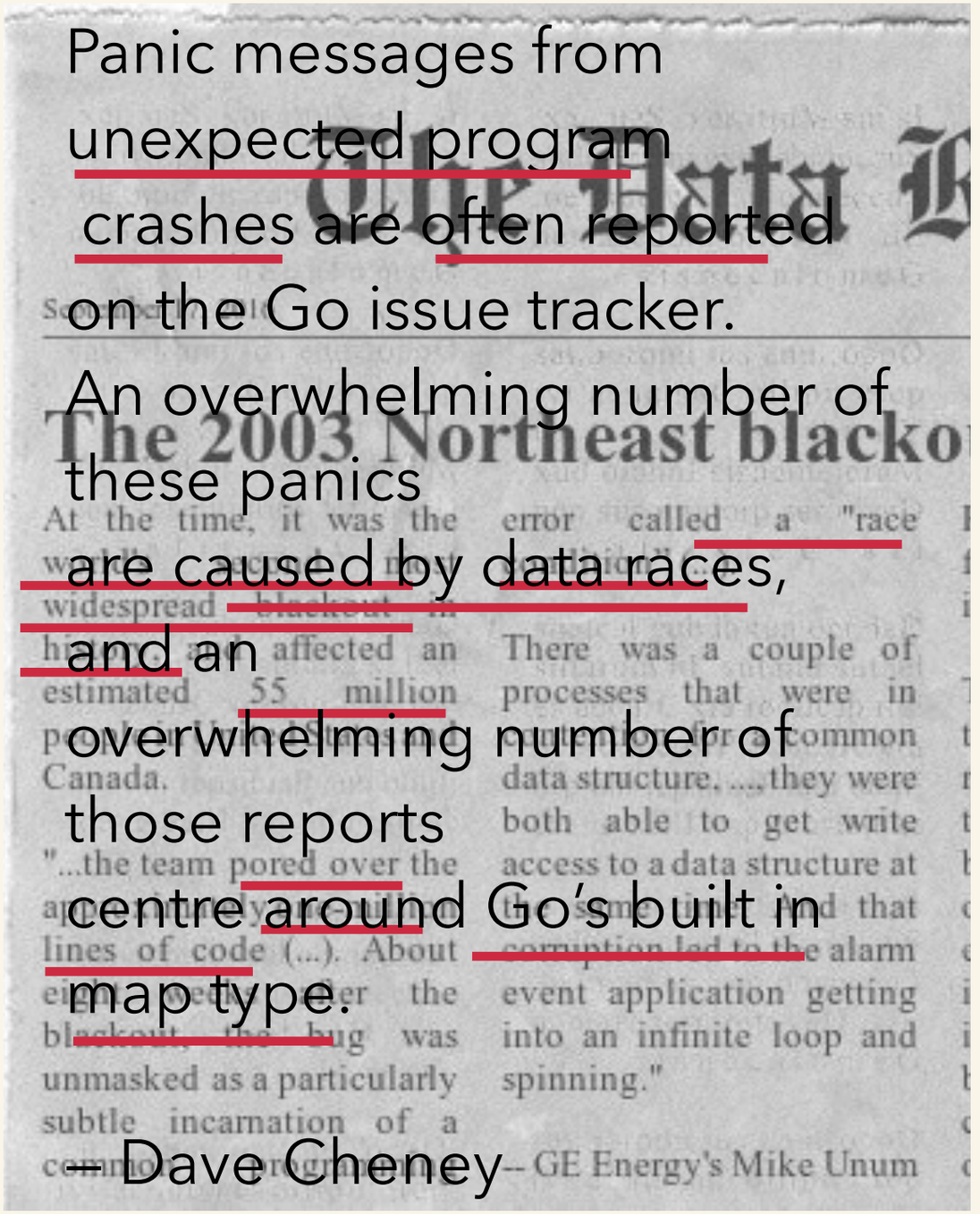
data race

```
// Shared variable
var count = 0

func incrementCount() {
  if count == 0 {
    count ++
  }
}

func main() {
  // Spawn two "threads"
  go incrementCount()
  go incrementCount()
}
```


!data race

```
Thread 1    Thread 2
lock(l)     lock(l)
count=1     count=2
unlock(l)   unlock(l)
```

- relevant
- elusive
- have undefined consequences
- easy to introduce in languages like Go



given we want to write multithreaded programs,

how may we protect our systems from the

unknown consequences of the

difficult-to-track-down data race bugs…

in a manner that is reliable and scalable?

race detectors

```
==================
WARNING: DATA RACE
Read by goroutine 7:
  main.incrementCount()
      /Users/kavyajoshi/strangeloop/test.go:11 +0x19b

Previous write by goroutine 6:
  main.incrementCount()
      /Users/kavyajoshi/strangeloop/test.go:11 +0x1b7

Goroutine 7 (running) created at:
  main.main()
      /Users/kavyajoshi/strangeloop/test.go:17 +0x50

Goroutine 6 (finished) created at:
  main.main()
      /Users/kavyajoshi/strangeloop/test.go:16 +0x38
==================
Found 1 data race(s)
```

read by goroutine 7

at incrementCount()

created at main()

…but how?

go race detector

- Go v1.1 (2013)

- Integrated with the Go tool chain –

```
> go run -race counter.go
```

- Based on C/ C++ ThreadSanitizer dynamic race detection library

- As of August 2015,

1200+ races in Google's codebase,
~100 in the Go stdlib,
100+ in Chromium,
+ LLVM, GCC, OpenSSL, WebRTC, Firefox

core concepts

internals

evaluation

wrap-up

core concepts

concurrency in go

The unit of concurrent execution : goroutines

- user-space threads
- use as you would threads

```
> go handle_request(r)
```

- Go memory model specified in terms of goroutines
  - within a goroutine: reads + writes are ordered
  - with multiple goroutines: shared data must be synchronized...else data races!

The synchronization primitives:

- channels

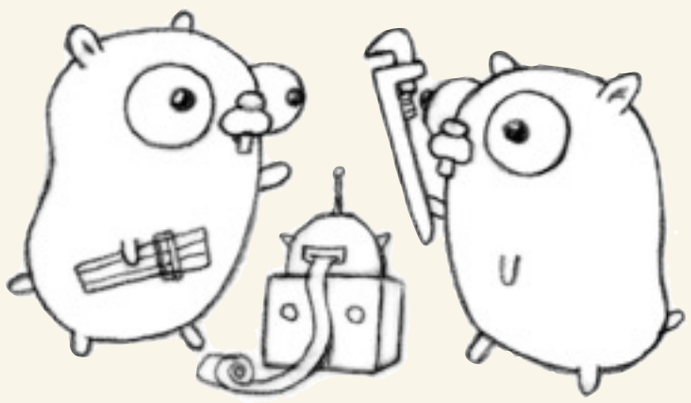
```
> ch <- value
```

- mutexes, conditional vars, ...

```
> import “sync”
> mu.Lock()
```

- atomics

```
> import “sync/ atomic"
> atomic.AddUint64(&myInt, 1)
```



concurrency

?

"...goroutines concurrently access a shared memory location, at least one access is a write."

```
var count = 0

func incrementCount() {
   if count == 0 {
      count ++
   }
}

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

"g1"

"g2"

R	R	R
W	R	R
R	W	W
W	W	W
count = 1	count = 2	count = 2
!concurrent	concurrent	concurrent

how can we determine

"concurrent"

memory accesses?

```
var count = 0

func incrementCount() {
  if count == 0 {
    count++
  }
}
```

```
func main() {
  incrementCount()
  incrementCount()
}
```

not concurrent – same goroutine

```
var count = 0

func incrementCount() {
  mu.Lock()
  if count == 0 {
    count ++
  }
  mu.Unlock()
}
```

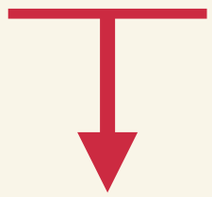
```
func main() {
  go incrementCount()
  go incrementCount()
}
```

not concurrent –

lock draws a “dependency edge”

happens-before

orders events across goroutines



memory accesses
i.e. reads, writes

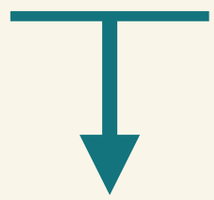
```
a := b
```

synchronization
via locks or lock-free sync

```
mu.Unlock()
ch <- a
```

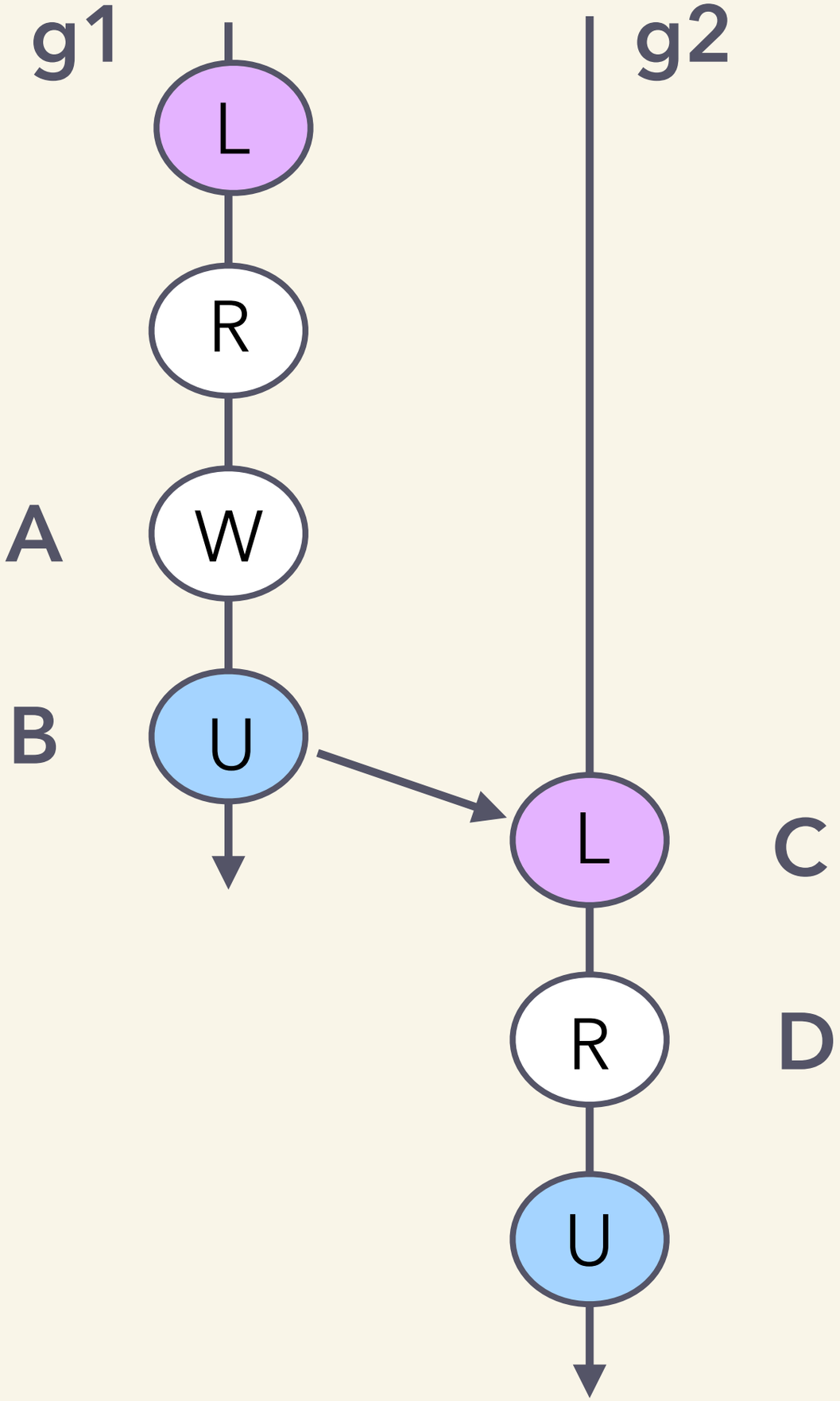
X ≺ Y IF one of:

– same goroutine
– are a synchronization-pair
– X ≺ E ≺ Y



IF X not ≺ Y and Y not ≺ X ,

concurrent!



A < B

same goroutine

B < C

lock-unlock on same object

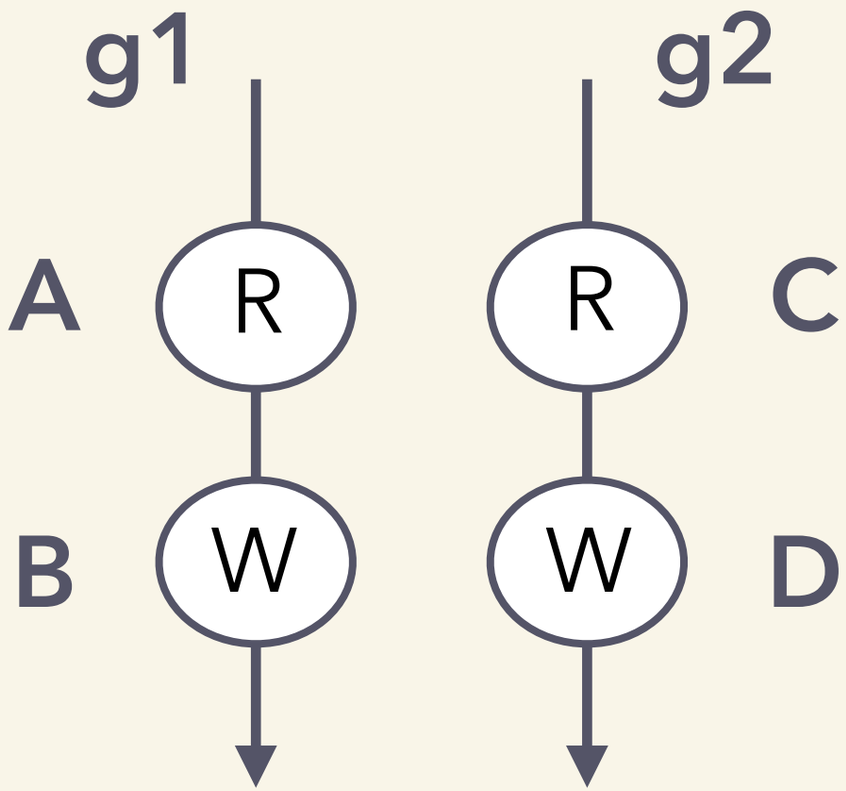
A < D

transitivity

```
var count = 0

func incrementCount() {
  if count == 0 {
    count ++
  }
}
```

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

concurrent ?



A < B and C < D

same goroutine

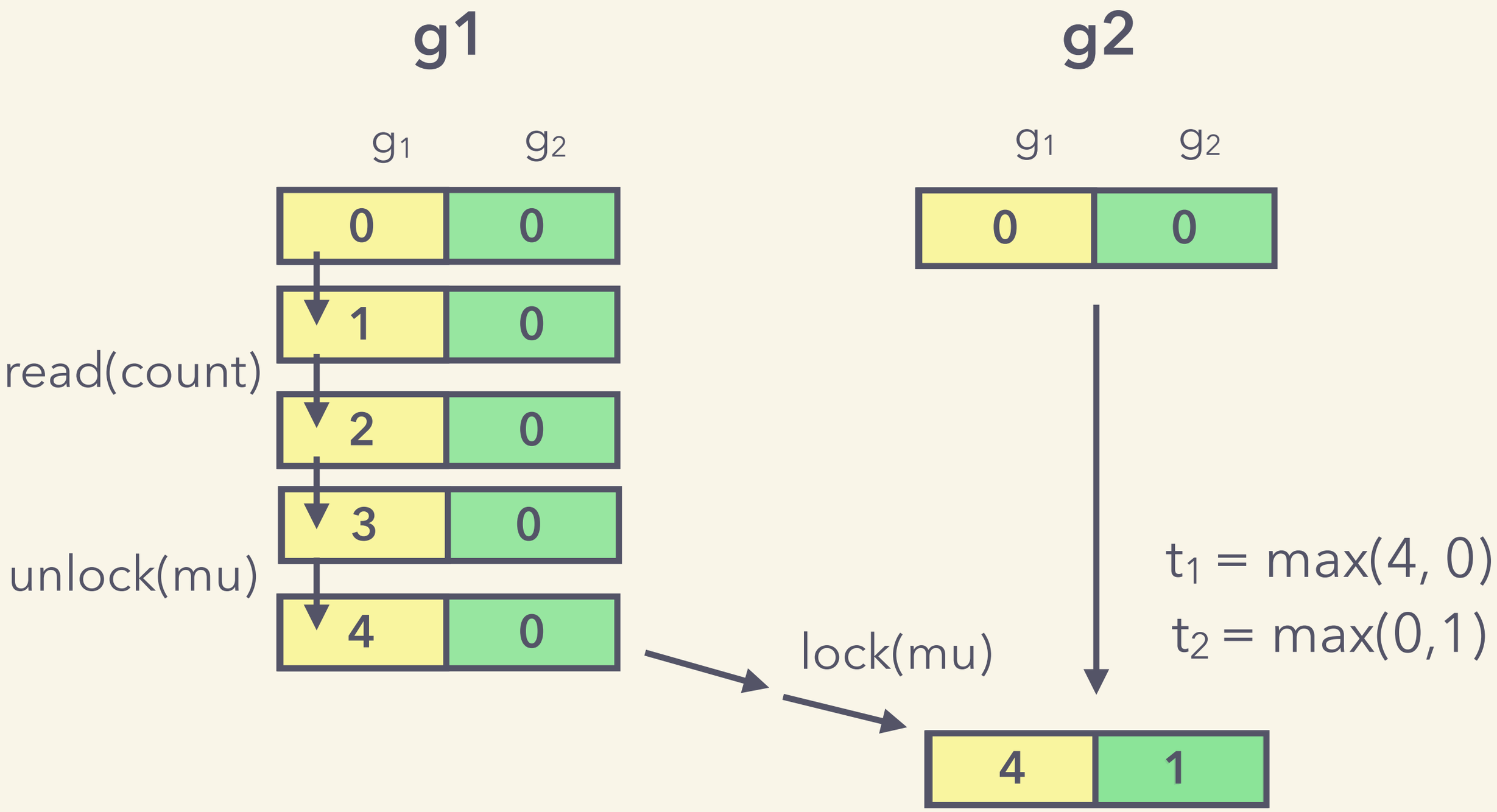
but A ? C and C ? A

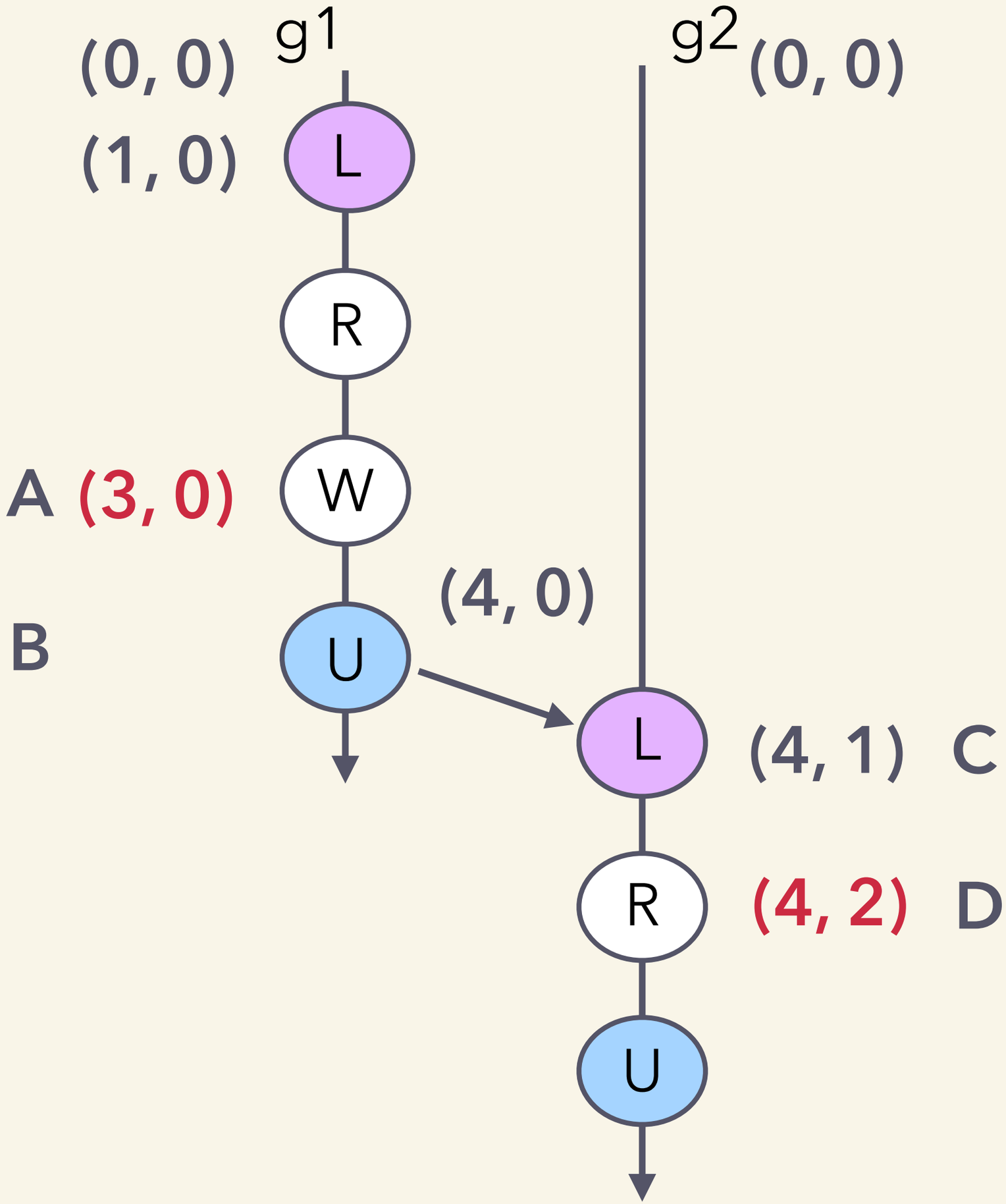
concurrent

how can we implement happens-before?

vector clocks

means to establish happens-before edges

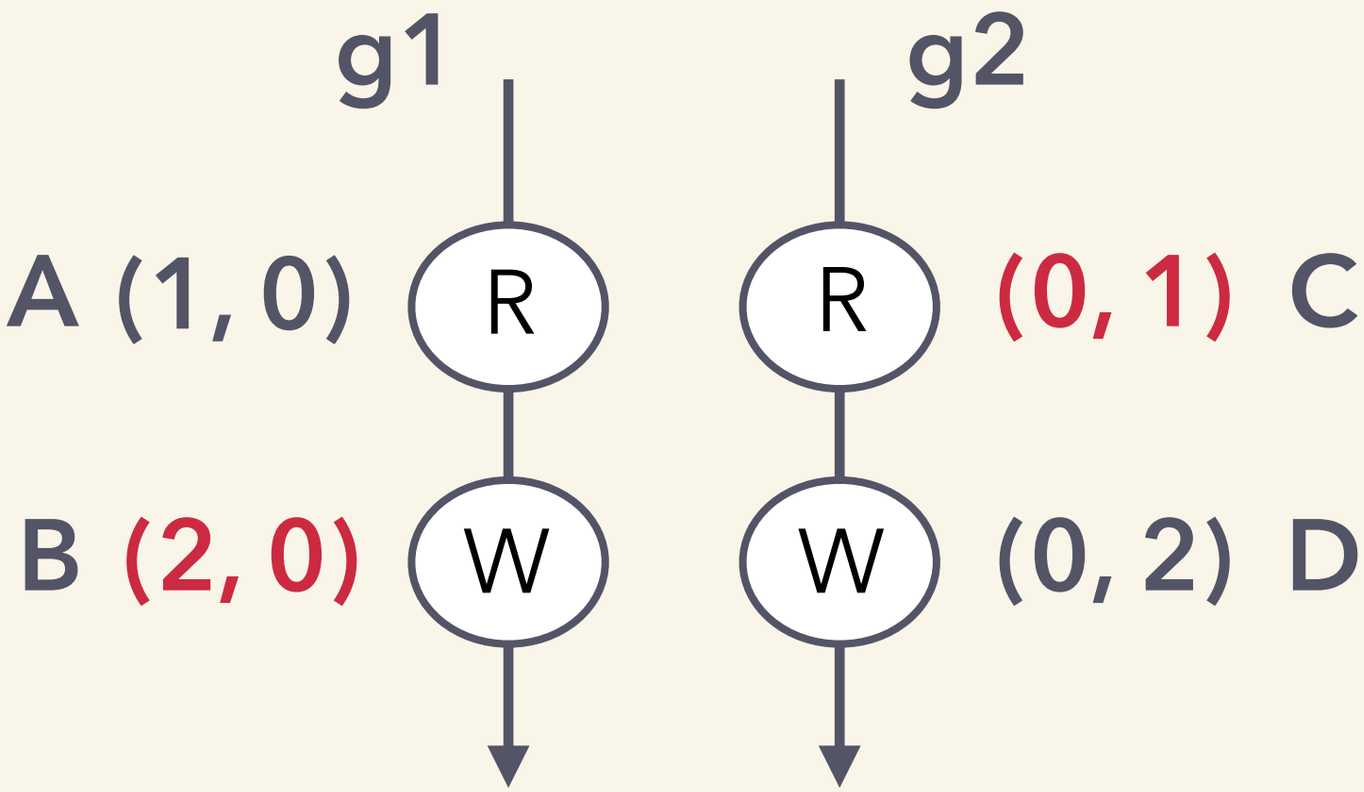




A < D ?

(3, 0) < (4, 2) ?

so **yes**.



B < C ?

(2, 0) < (0, 1) ?

no.

C < B ?

no.

so, **concurrent**

pure happens-before detection

This is what the Go Race Detector does!

Determines if the accesses to a memory location can be ordered by happens-before, using vector clocks.

internals

go run -race

to implement happens-before detection, need to:

- [ ] create vector clocks for goroutines

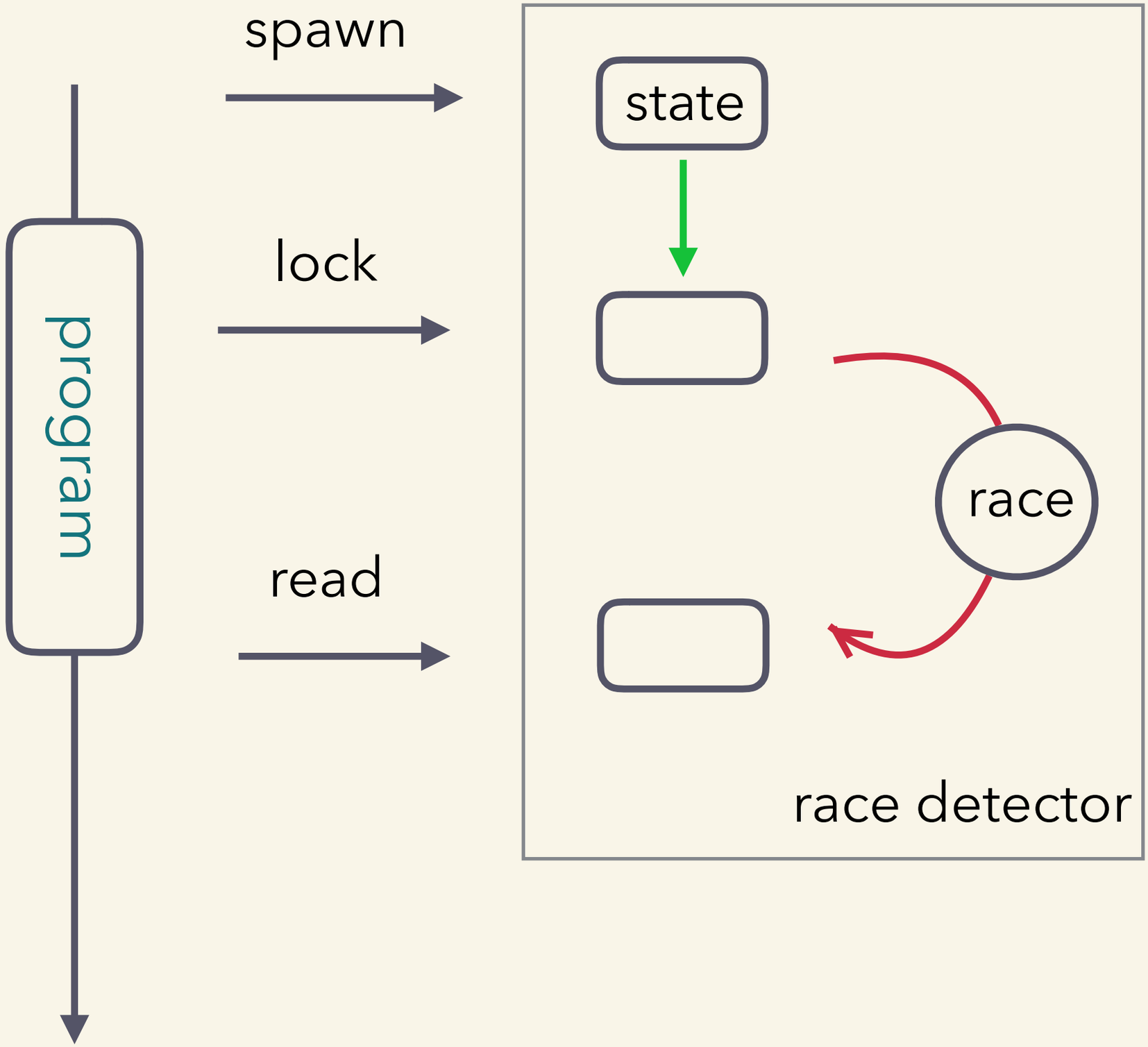
  …at goroutine creation

- [ ] update vector clocks based on memory access, synchronization events

  …when these events occur

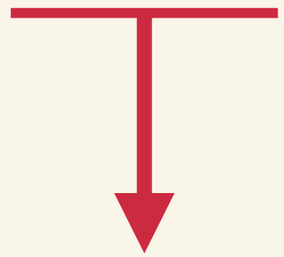
- [ ] compare vector clocks to detect happens-before relations.

  …when a memory access occurs



race detector state machine

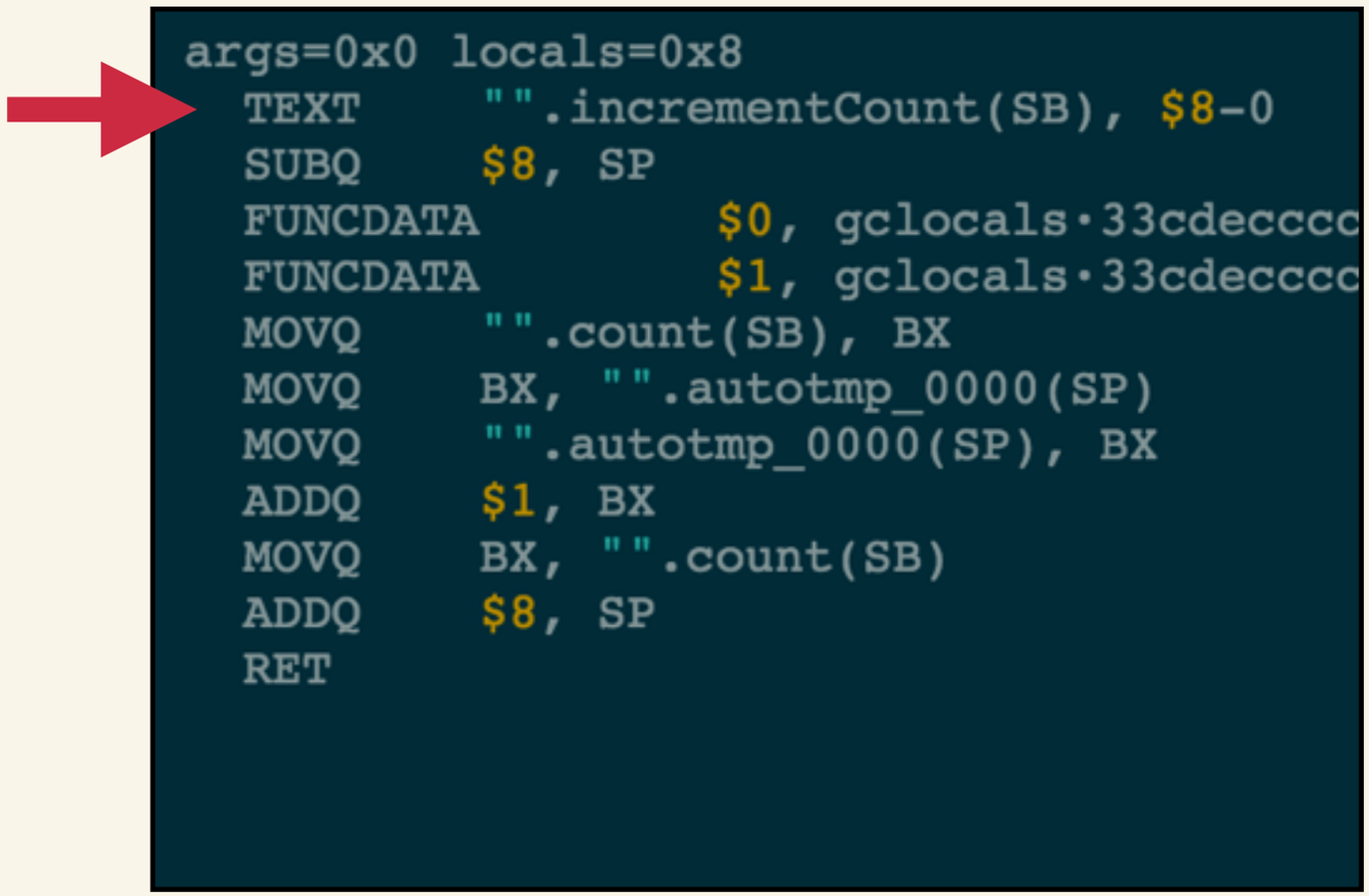
do we have to modify
our programs then,
to generate the events?



memory accesses
synchronizations
goroutine creation

nope.

```go
var count = 0

func incrementCount() {
   if count == 0 {
      count ++
   }
}

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



```
args=0x0 locals=0x8
   TEXT    "".incrementCount(SB), $8-0
   SUBQ    $8, SP
   FUNCDATA        $0, gclocals·33cdeccccebe
   FUNCDATA        $1, gclocals·33cdeccccebe
   MOVQ    "".count(SB), BX
   MOVQ    BX, "".autotmp_0000(SP)
   MOVQ    "".autotmp_0000(SP), BX
   ADDQ    $1, BX
   MOVQ    BX, "".count(SB)
   ADDQ    $8, SP
   RET
```

```go
var count = 0

func incrementCount() {
    raceread()
    if count == 0 {
        racewrite()
        count ++
    }
    racefuncexit()
}

func main() {
    go incrementCount()
    go incrementCount()
```

```
args=0x0 locals=0x10
  TEXT    "".incrementCount(SB), $16-0
  MOVQ    (TLS), CX
  CMPQ    SP, 16(CX)
  JLS     103
  SUBQ    $16, SP
  FUNCDATA        $0, gclocals·33cdeccccebe80329f1f
  FUNCDATA        $1, gclocals·33cdeccccebe80329f1f
  MOVQ    ""..fp+16(FP), BX
  MOVQ    BX, (SP)
  PCDATA  $0, $0
  CALL    runtime.racefuncenter(SB)
  LEAQ    "".count(SB), BX
  MOVQ    BX, (SP)
  PCDATA  $0, $0
  CALL    runtime.raceread(SB)
  MOVQ    "".count(SB), BX
  MOVQ    BX, "".autotmp_0000+8(SP)
  LEAQ    "".count(SB), BX
  MOVQ    BX, (SP)
  PCDATA  $0, $0
  CALL    runtime.racewrite(SB)
  MOVQ    "".autotmp_0000+8(SP), BX
  ADDQ    $1, BX
  MOVQ    BX, "".count(SB)
  PCDATA  $0, $0
  CALL    runtime.racefuncexit(SB)
  ADDQ    $16, SP
  RET
```

-race

`go tool compile -race`

the gc compiler instruments memory accesses

adds an instrumentation pass over the IR.

```
func compile(fn *Node)
{
   ...

   order(fn)

   walk(fn)

   if instrumenting {
      instrument(Curfn)
   }
    ...
}
```

```
after walk incrementCount
.   AS u(2) l(7) tc(1)
.   .   NAME-main.autotmp_0000 u(1) a(true) l(7) x(0+0)
.   .   NAME-main.count u(1) a(true) l(4) x(0+0) class(P

.   AS u(2) l(7) tc(1)
.   .   NAME-main.count u(1) a(true) l(4) x(0+0) class(P
.   .   ADD u(2) l(7) tc(1) int
.   .   .   NAME-main.autotmp_0000 u(1) a(true) l(7) x(0
.   .   .   LITERAL-1 u(1) a(true) l(7) tc(1) int

.   VARKILL l(7) tc(1)
.   .   NAME-main.autotmp_0000 u(1) a(true) l(7) x(0+0)

before main
```

```
after instrument incrementCount
.   AS-init
.   .   CALLFUNC u(100) l(7) tc(1)
.   .   .   NAME-runtime.raceread u(1) a(true) l(2) x(0+0) class
.   .   CALLFUNC-list
.   .   .   AS u(2) l(7) tc(1)
.   .   .   .   INDREG-SP a(true) l(7) x(0+0) tc(1) uintptr
.   .   .   .   CONVNOP u(2) l(7) tc(1) uintptr
.   .   .   .   .   CONVNOP u(2) l(7) tc(1) E-30-unsafe.Pointer
.   .   .   .   .   .   ADDR u(2) l(7) tc(1) PTR64-*int
.   .   .   .   .   .   .   NAME-main.count u(1) a(true) l(4) x(
.   AS u(2) l(7) tc(1)
.   .   NAME-main.autotmp_0000 u(1) a(true) l(7) x(0+0) class(PA
.   .   NAME-main.count u(1) a(true) l(4) x(0+0) class(PEXTERN)
.   AS u(2) l(7) tc(1)
.   .   NAME-main.autotmp_0000 u(1) a(true) l(7) x(0+0) class(PA
.   .   NAME-main.count u(1) a(true) l(4) x(0+0) class(PEXTERN)
```

This is awesome.
We don’t have to modify our programs to track memory accesses.

What about synchronization events, and goroutine creation?

mutex.go

```
package sync

import “internal/race"

func (m *Mutex) Lock() {
  if race.Enabled {
    race.Acquire(…)
  }
  ...
}
```

```
  raceacquire(addr)
```

proc.go

```
package runtime

func newproc1() {
  if race.Enabled {
    newg.racectx =
      racegostart(…)
  }
  ...
}
```

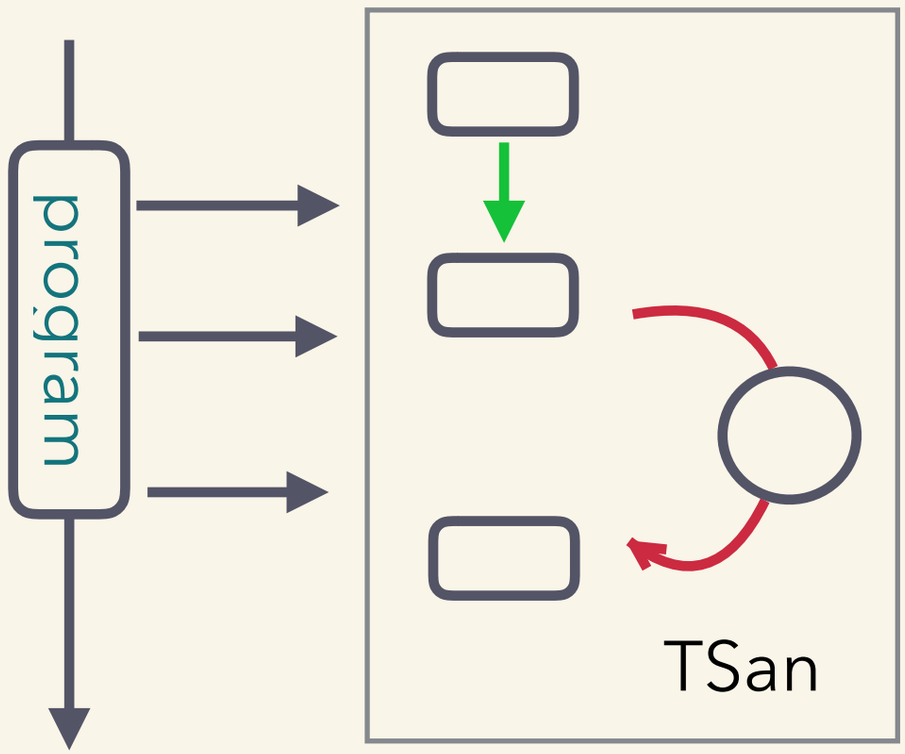
`runtime.raceread()`

```
TEXT    runtime·raceread(SB), NOSPLIT, $0-8
        MOVQ    addr+0(FP), RARG1
        MOVQ    (SP), RARG2
        // void __tsan_read(ThreadState *thr, void *addr, void *pc);
        MOVQ    $__tsan_read(SB), AX
        JMP     racecalladdr<>(SB)
```

ThreadSanitizer (TSan) library

C++ race-detection library

(.asm file because it's calling into C++)



threadsanitizer

TSan implements the happens-before race detection:

- creates, updates vector clocks for goroutines -> **ThreadState**
- keeps track of memory access, synchronization events -> **Shadow State, Meta Map**
- compares vector clocks to detect data races.

`go incrementCount()`

```
func newproc1() {
  if race.Enabled {
    newg.racectx =
      racegostart(…)
  }
  ...
}
```

proc.go

```
struct ThreadState {
  ThreadClock clock;
}
```

contains a fixed-size vector clock

(size == max(# threads))

`count == 0`

`raceread(…)` → 1. data race with a previous access?
2. store information about this access for future detections

by compiler instrumentation

shadow state

stores information about memory accesses.

8-byte shadow word for an access:

TID	clock	pos	wr

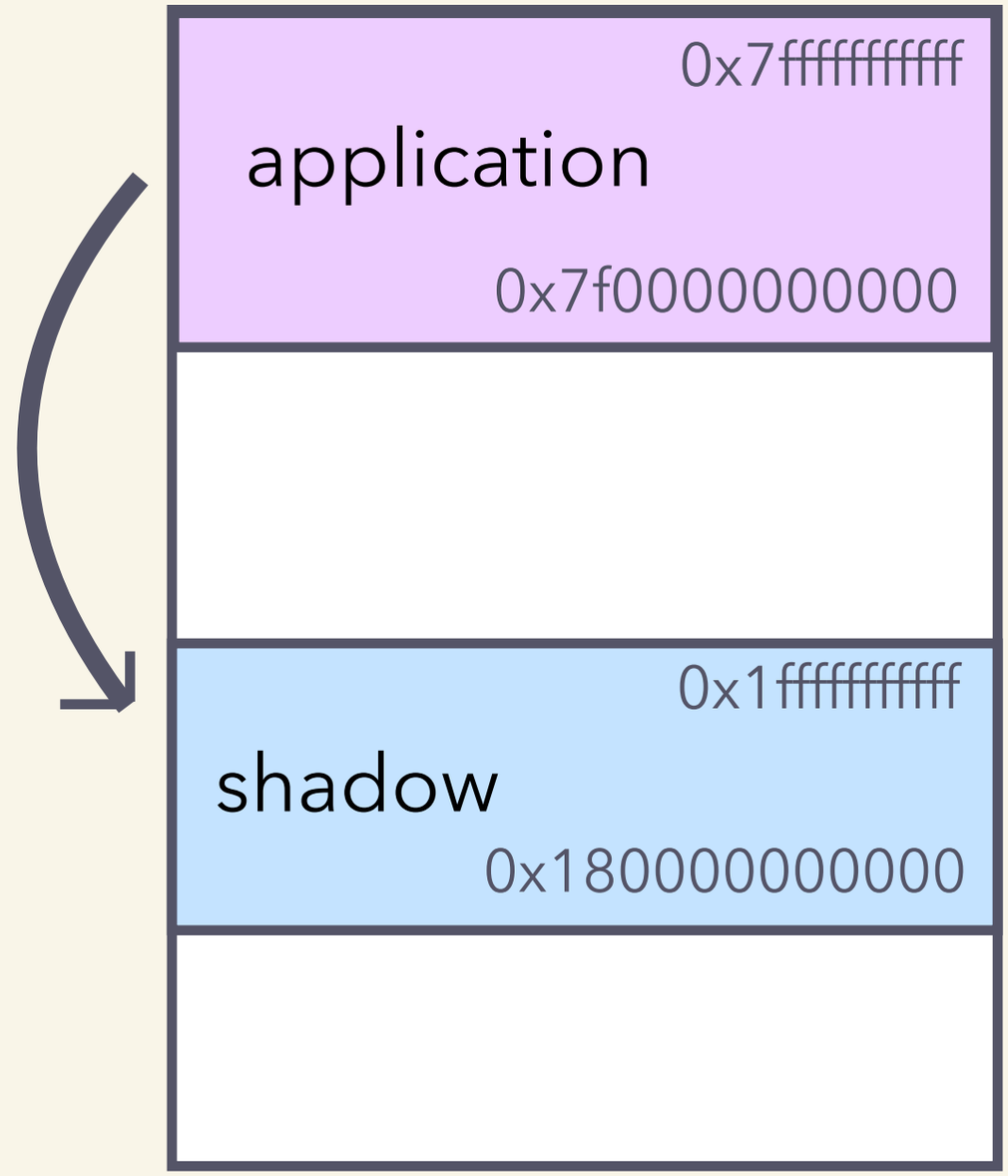
TID: accessor goroutine ID

clock: scalar clock of accessor , optimized vector clock

pos: offset, size in 8-byte word

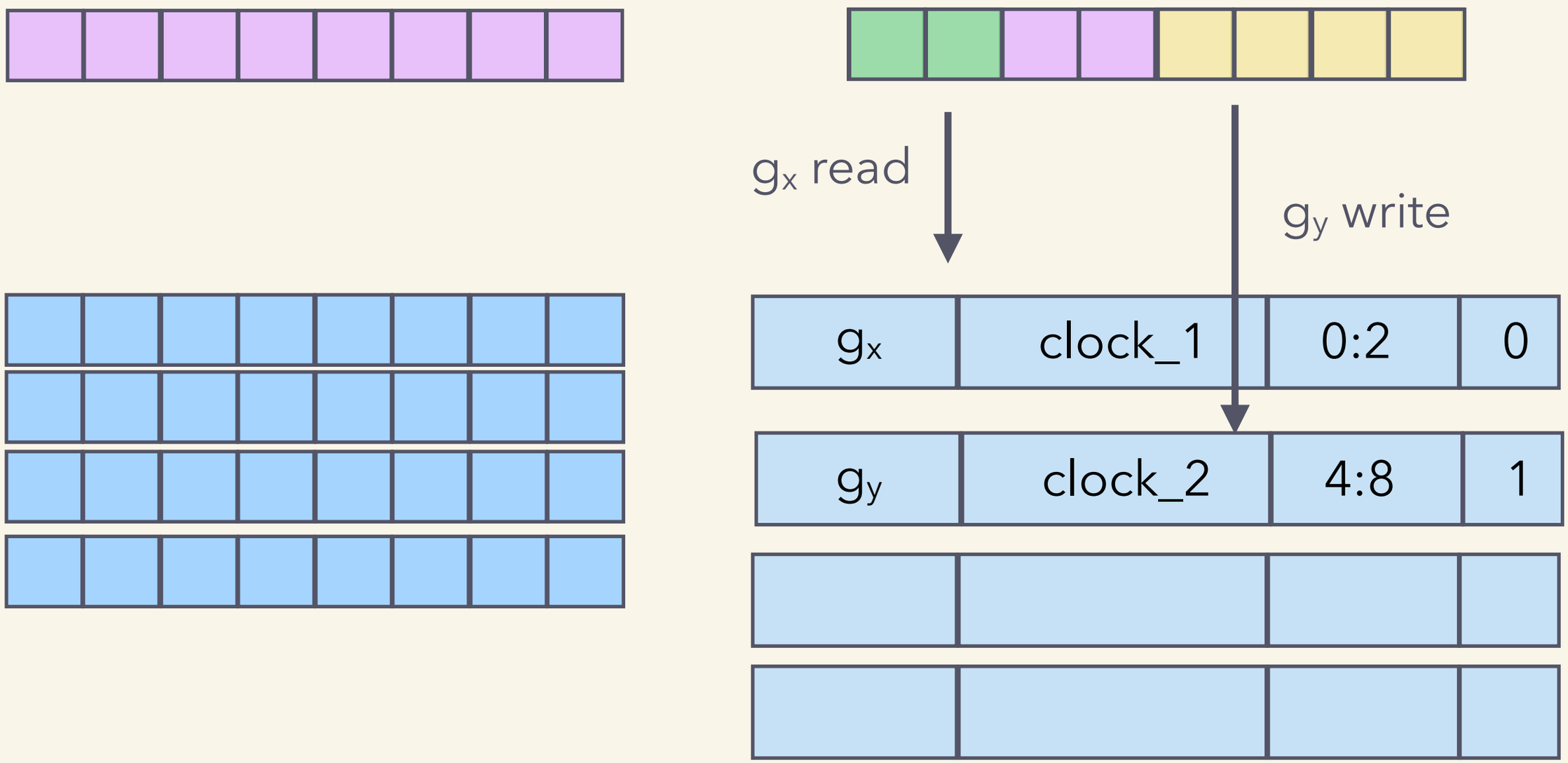
wr: IsWrite bit

directly-mapped:



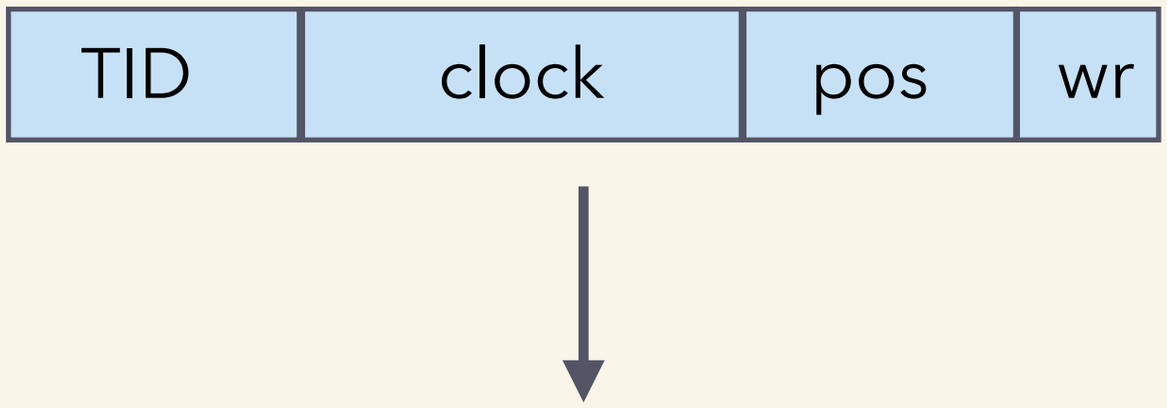
Optimization 1

N shadow cells per application word (8-bytes)



When shadow words are filled, evict one at random.

Optimization 2

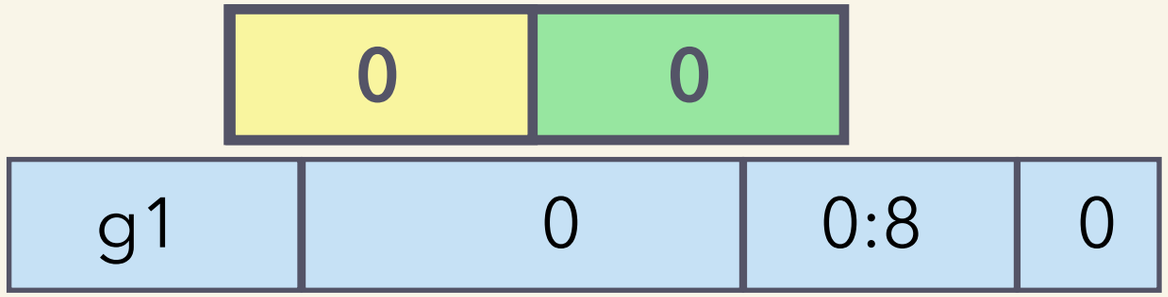


scalar clock, not full vector clock.

g_x access:

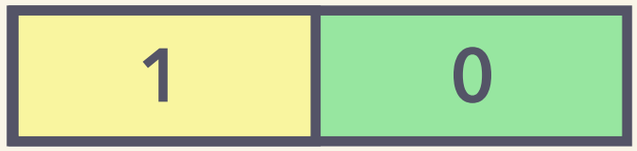
`g1: count == 0`

`raceread(…)` →



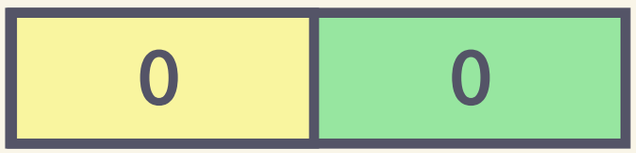
by compiler instrumentation

`g1: count++`



`racewrite(…)` →

`g2: count == 0`



`raceread(…)` →

and check for race

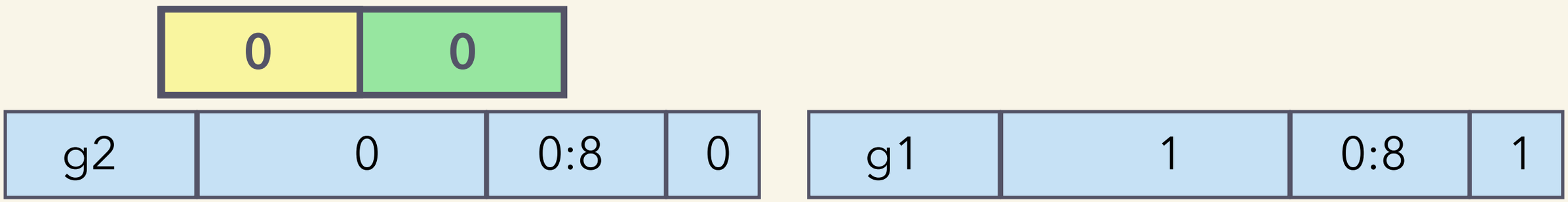
race detection

compare:

<accessor's vector clock,

new shadow word>

with: each existing shadow word



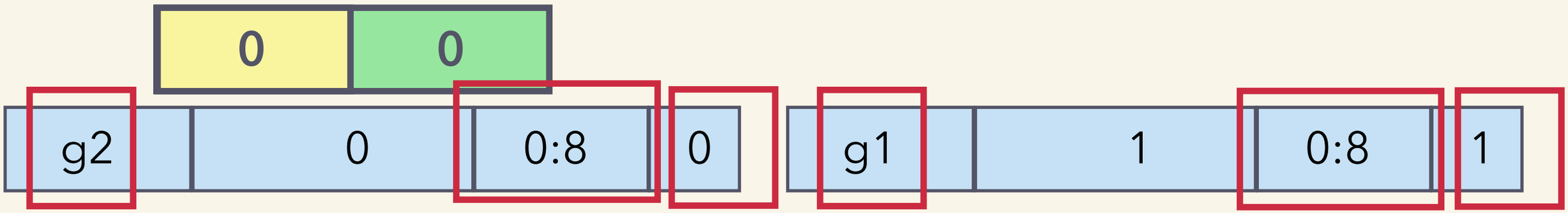
"…when two+ threads concurrently access a shared memory location, at least one access is a write."

race detection

compare:

<accessor's vector clock,
new shadow word>

with: each existing shadow word

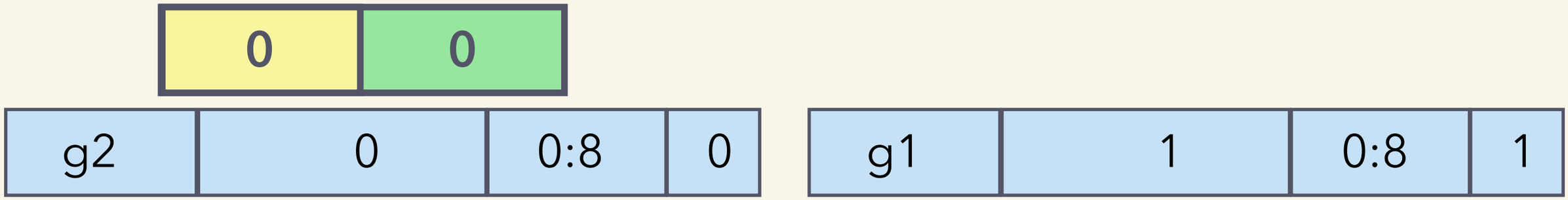


- ☑ do the access locations overlap?
- ☑ are any of the accesses a write?
- ☑ are the TIDS different?
- ☑ are they concurrent (no happens-before)?

g2's vector clock: (0, 0)
existing shadow word's clock: (1, ?)

race detection

compare (accessor's threadState, new shadow word) with each existing shadow word:



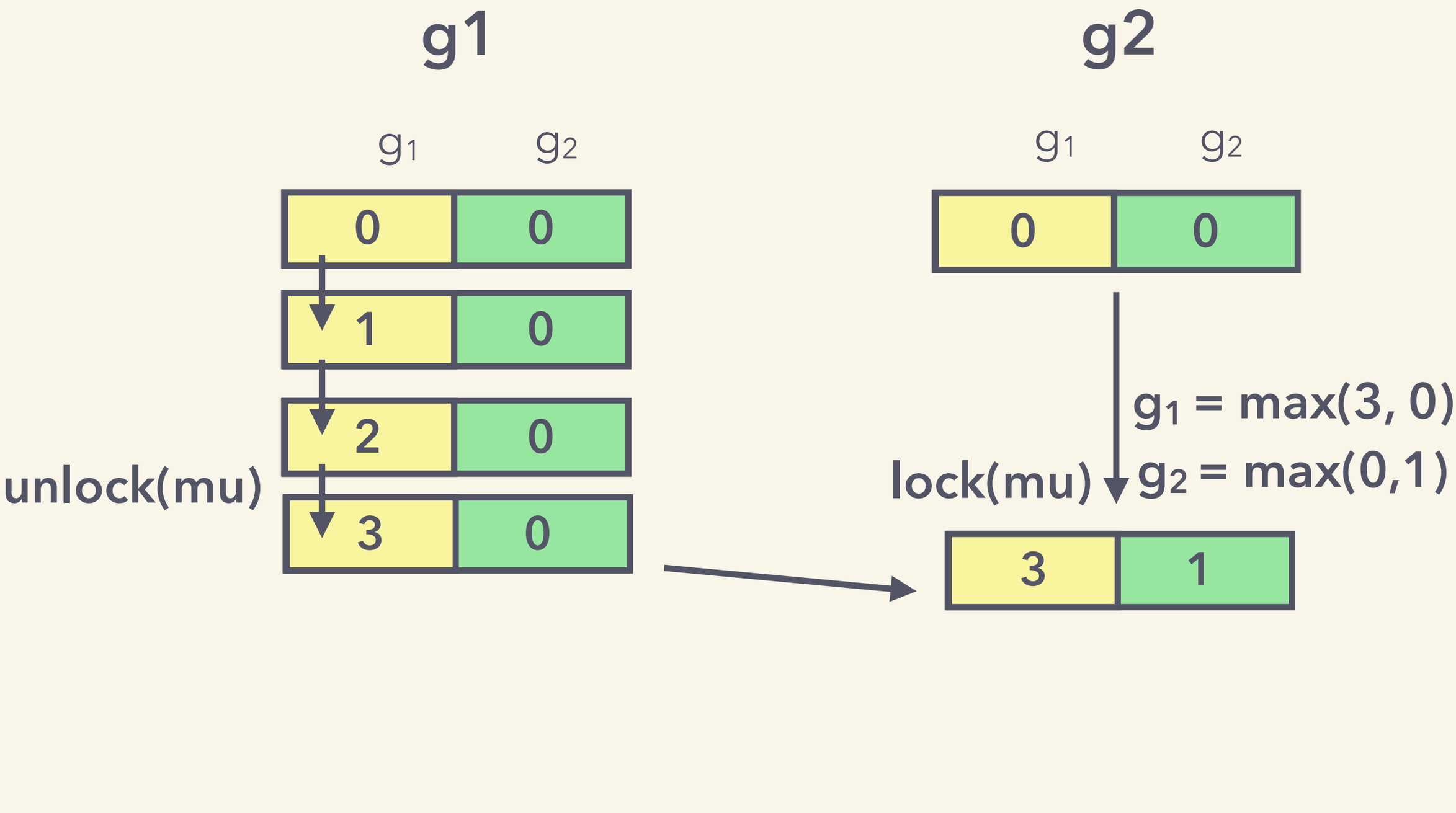
- ☑ do the access locations overlap?
- ☑ are any of the accesses a write?
- ☑ are the TIDS different?
- ☑ are they concurrent (no happens-before)?

↓

RACE!

synchronization events

TSan must track synchronization events

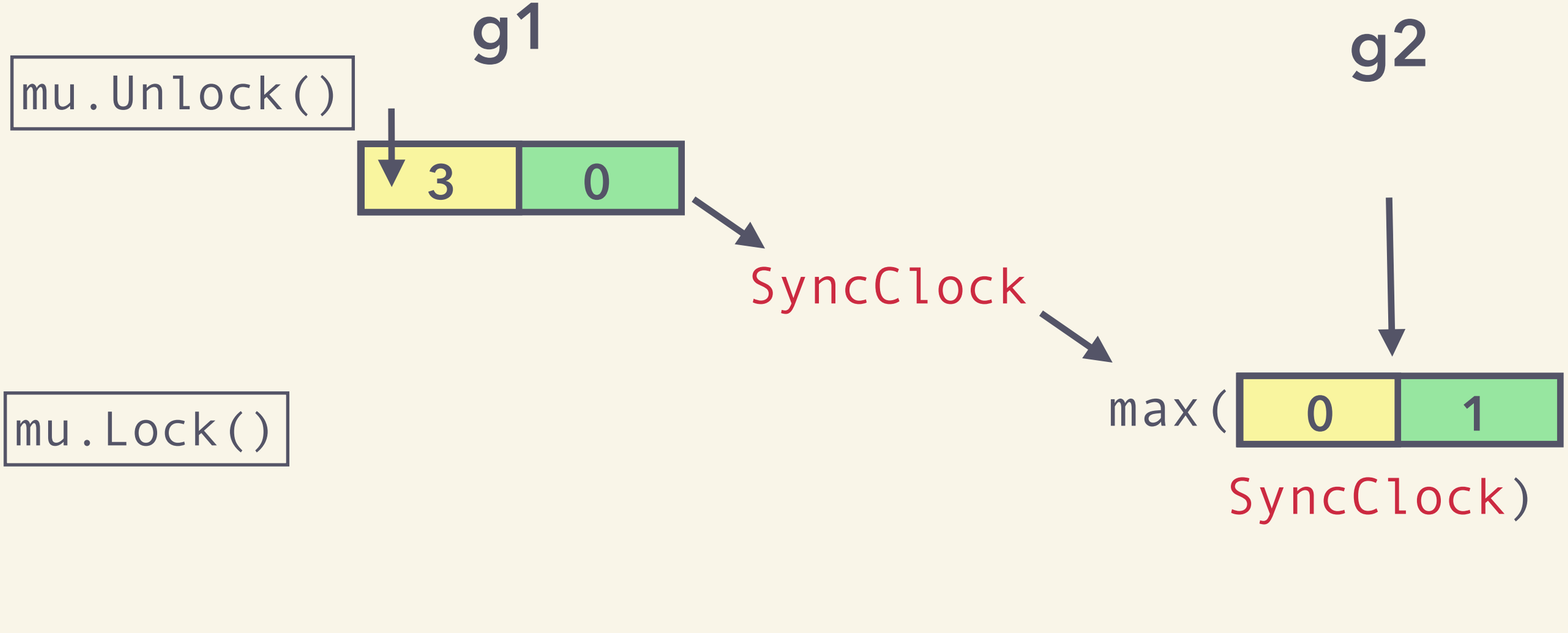


sync vars

```
mu := sync.Mutex{}
```

```
struct SyncVar {
  SyncClock clock;
}
```


stored in the **vectorclock** region.



a note (or two)...

TSan tracks file descriptors, memory allocations etc. too

TSan can track your custom sync primitives too, via dynamic annotations!

evaluation

evaluation

“is it reliable?”

no false positives
(only reports “real races”,
but can be benign)

can miss races!
depends on execution trace

“is it scalable?”

program slowdown = 5x-15x
memory usage = 5x-10x

As of August 2015,
1200+ races in Google’s codebase,
~100 in the Go stdlib,
100+ in Chromium,
+ LLVM, GCC, OpenSSL, WebRTC, Firefox

data race detection

with

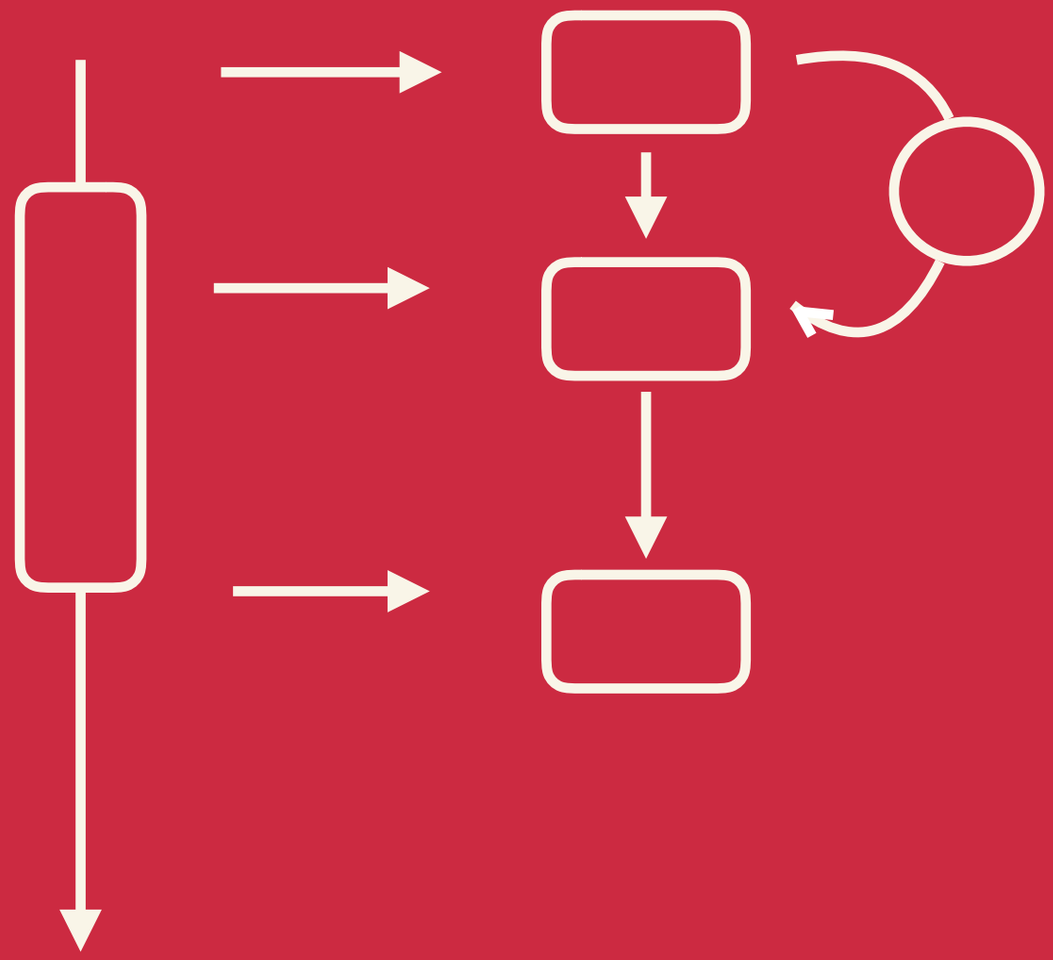
go run -race =

gc compiler instrumentation +

TSan runtime library for

happens-before using

vector clocks



@kavya719

alternatives

I. Static detectors

analyze the program's source code.

- typically have to augment the source with race annotations (-)
- single detection pass sufficient to determine all possible races (+)
- too many false positives to be practical (-)

II. Lockset-based dynamic detectors

uses an algorithm based on locks held

- more performant than pure happens-before (+)
- may not recognize synchronization via non-locks, like channels (would report as races) (-)

III. Hybrid dynamic detectors
combines happens-before + locksets.
(TSan v1, but it was hella unscalable)

- “best of both worlds” (+)
- false positives (-)
- complicated to implement (-)

requirements

I. Go specifics

v1.1+
gc compiler
gccgo does not support as per:
https://gcc.gnu.org/ml/gcc-patches/2014-12/msg01828.html
x86_64 required
Linux, OSX, Windows

II. TSan specifics

LLVM Clang 3.2, gcc 4.8
x86_64
requires ASLR, so compile/ ld with -fPIE, -pie
maps (using mmap but does not reserve) virtual address space;
tools like top/ ulimit may not work as expected.

fun facts

TSan

maps (by mmap but does not reserve) tons of virtual address space; tools like top/ ulimit may not work as expected.

need: gdb -ex 'set disable-randomization off' --args ./a.out due to ASLR requirement.

Deadlock detection?
Kernel TSan?

a fun concurrency example

goroutine 1

```
obj.UpdateMe()
mu.Lock()
flag = true
mu.Unlock()
```

goroutine 2

```
mu.Lock()
var f bool = flag
mu.Unlock ()
if (f) {
  obj.UpdateMe()
}
```