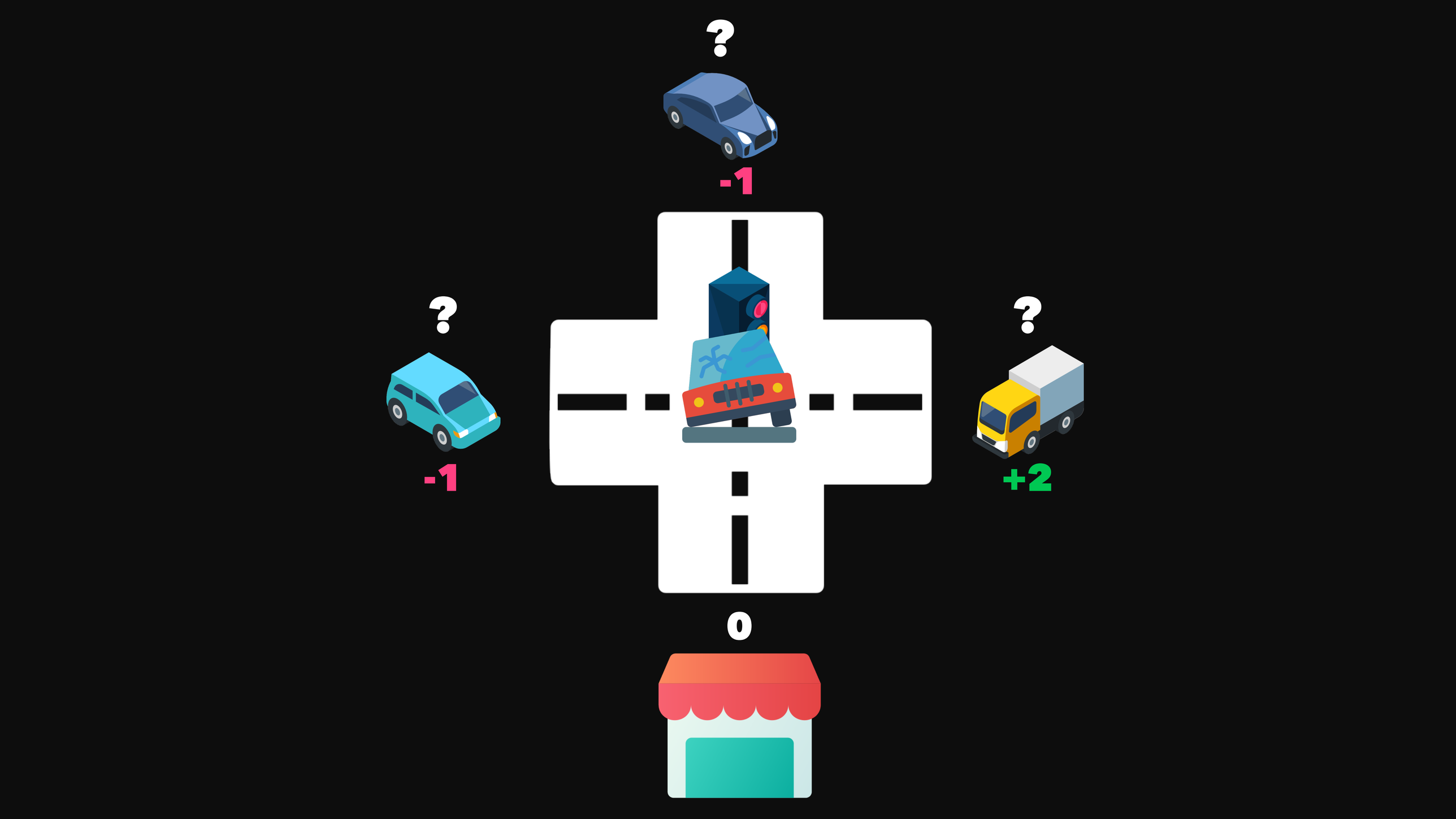
In computer science **Mutual Exclusion** is a property of **Concurrency Control**, which is instituted for the purpose of preventing **Race Conditions**.

”

1

2

3


?
-1
?
-1
?
+2
0

ORDER

RESULT

CORRECTNESS

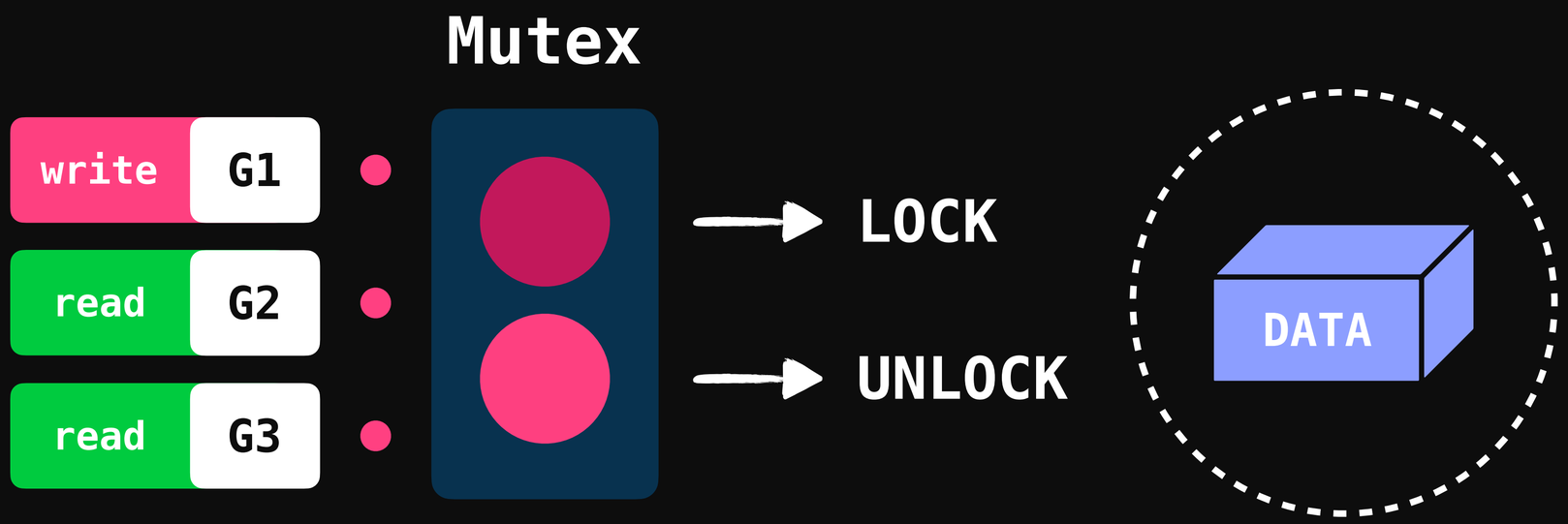

G1
G2
DATA
RACE CONDITION
read
write
write
write

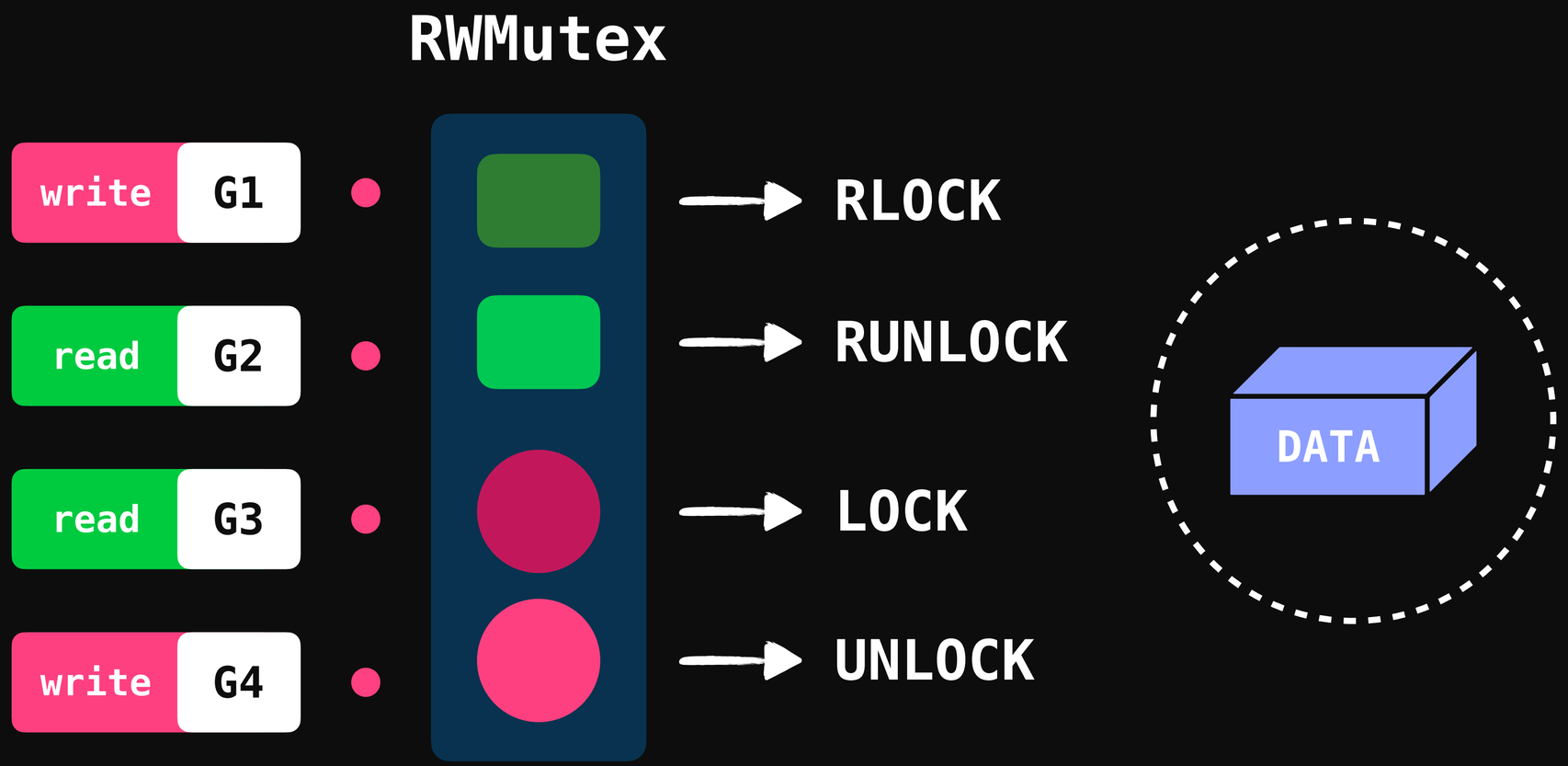
READ WRITE ONCE

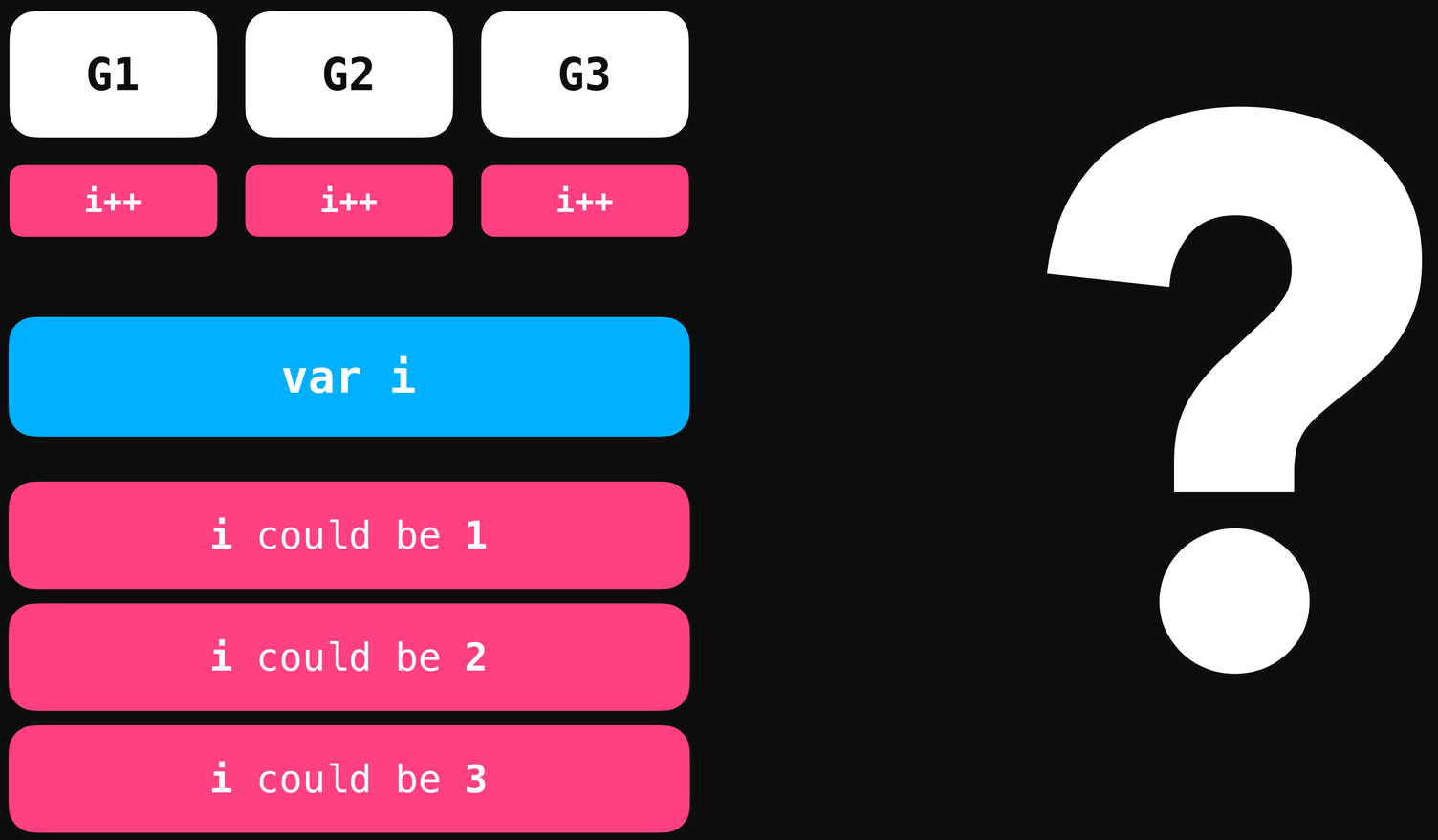
MUTEX

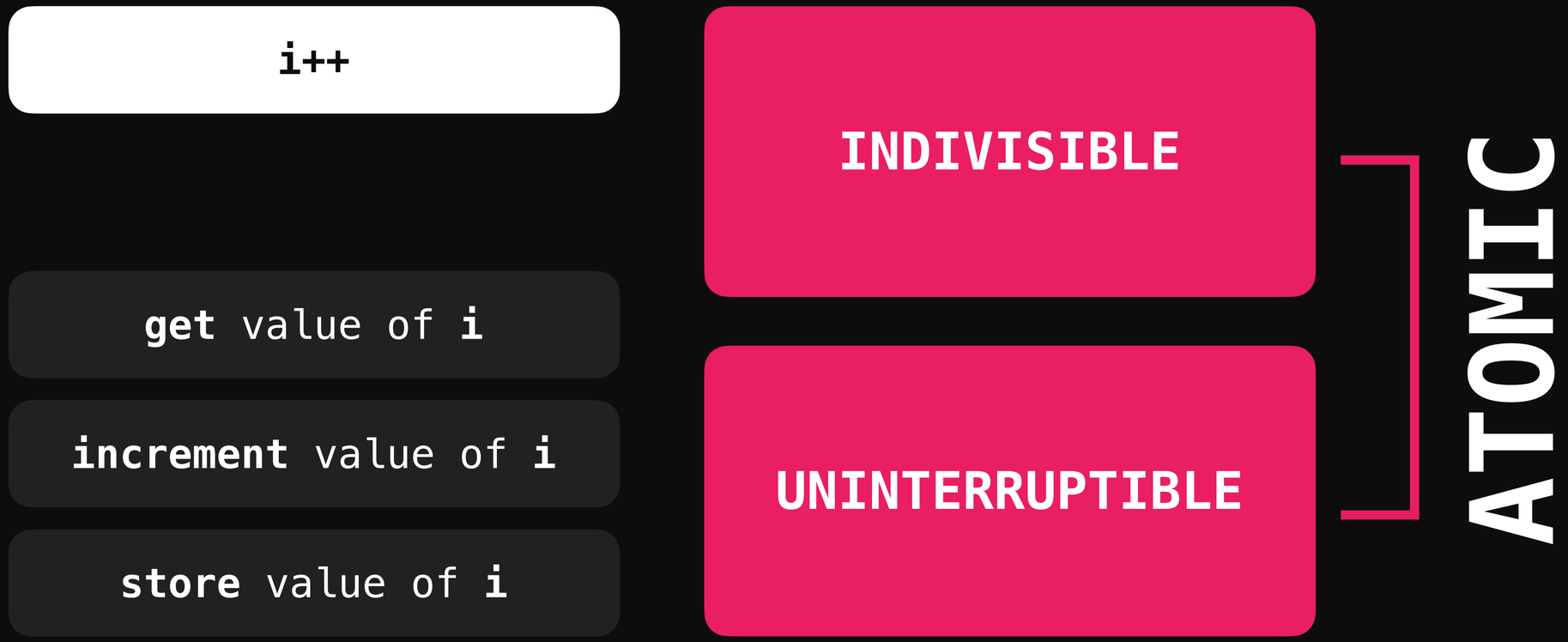
WRITE ONCE, READ MANY

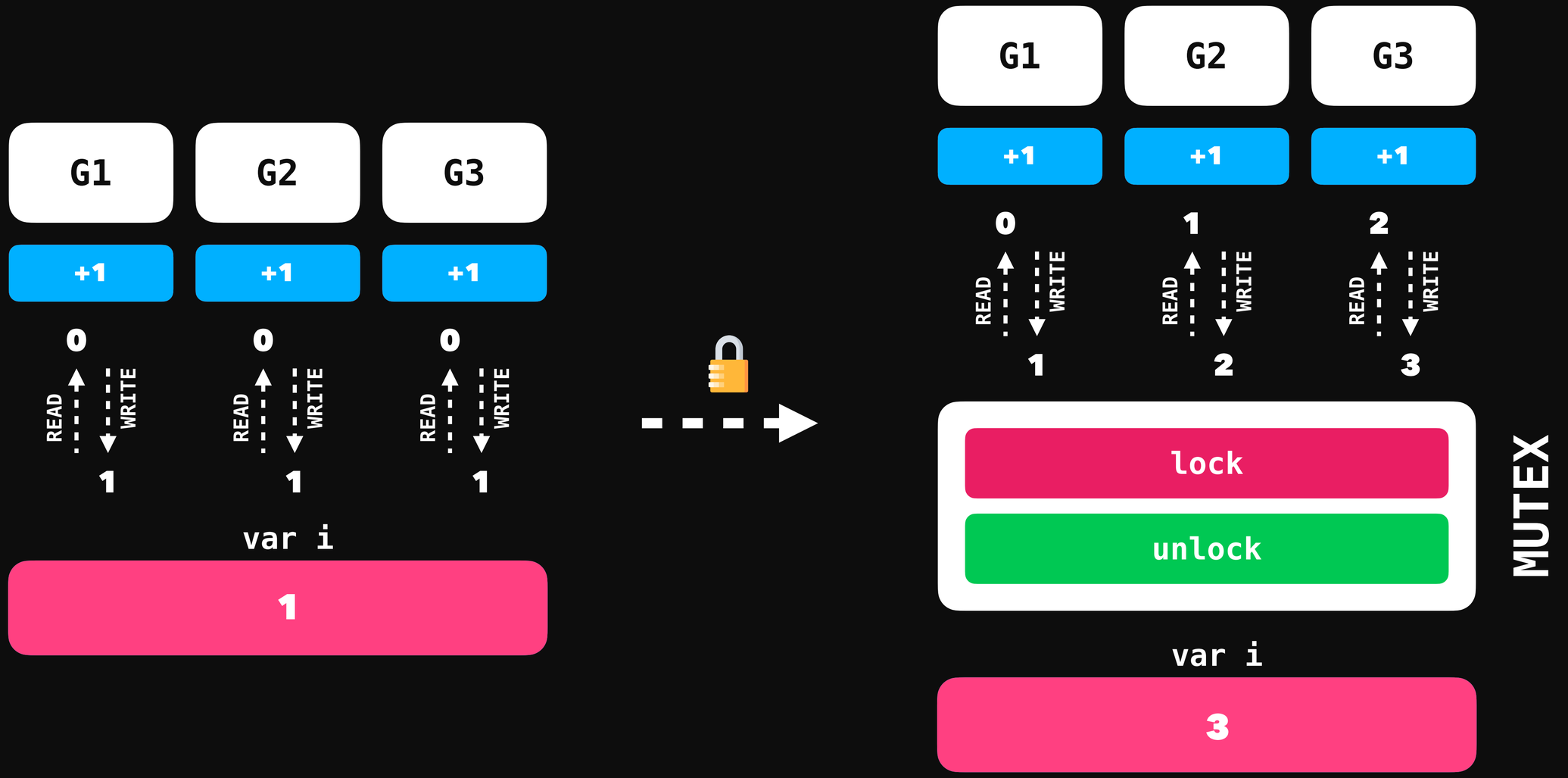
RWMUTEX

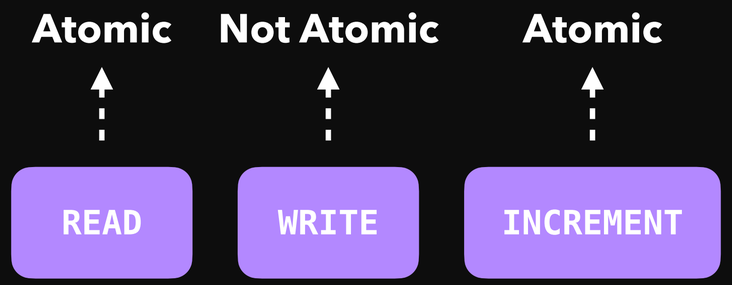

Mutex
write
G1
read
G2
read
G3
LOCK
UNLOCK
DATA


RWMutex
write
G1
read
G2
read
G3
write
G4
RLOCK
RUNLOCK
LOCK
UNLOCK
DATA

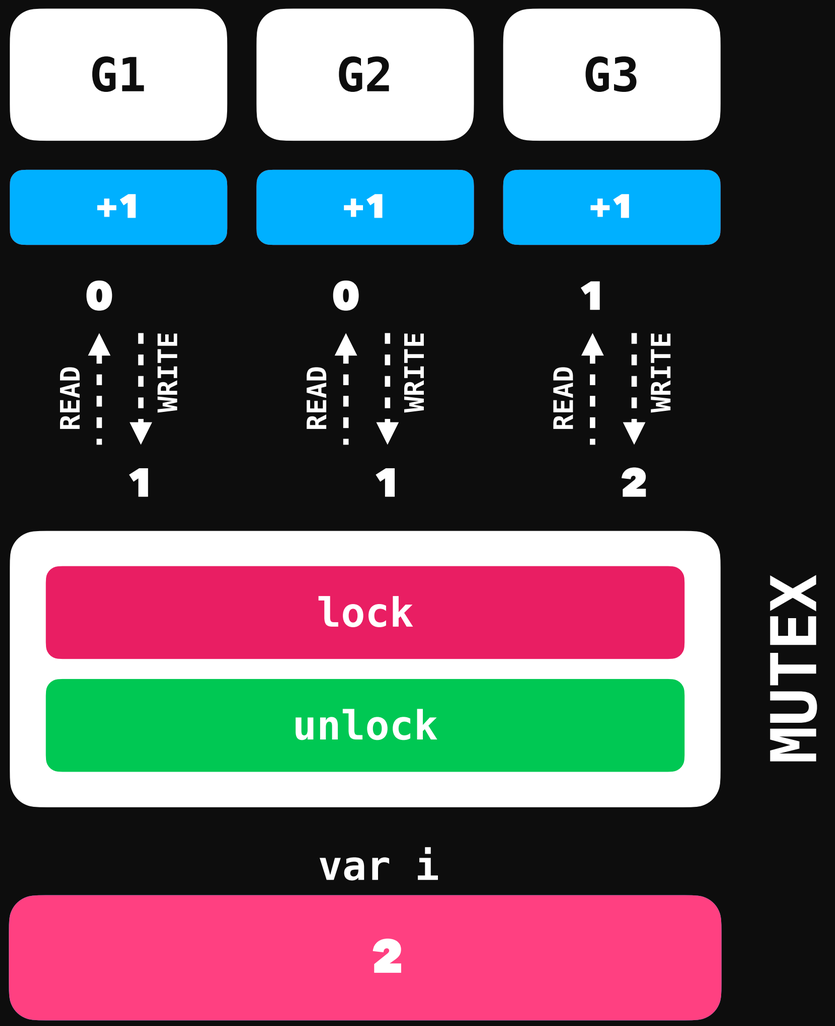

G1
G2
G3
i++
i++
i++
var i
i could be 1
i could be 2
i could be 3


i++
get value of i
increment value of i
store value of i
INDIVISIBLE
UNINTERRUPTIBLE
ATOMIC


G1
G2
G3
+1
+1
+1
0
0
0
READ
WRITE
READ
WRITE
READ
WRITE
1
1
1
var i
1
G1
G2
G3
+1
+1
+1
0
1
2
READ
WRITE
READ
WRITE
READ
WRITE
1
2
3
lock
unlock
MUTEX
var i
3

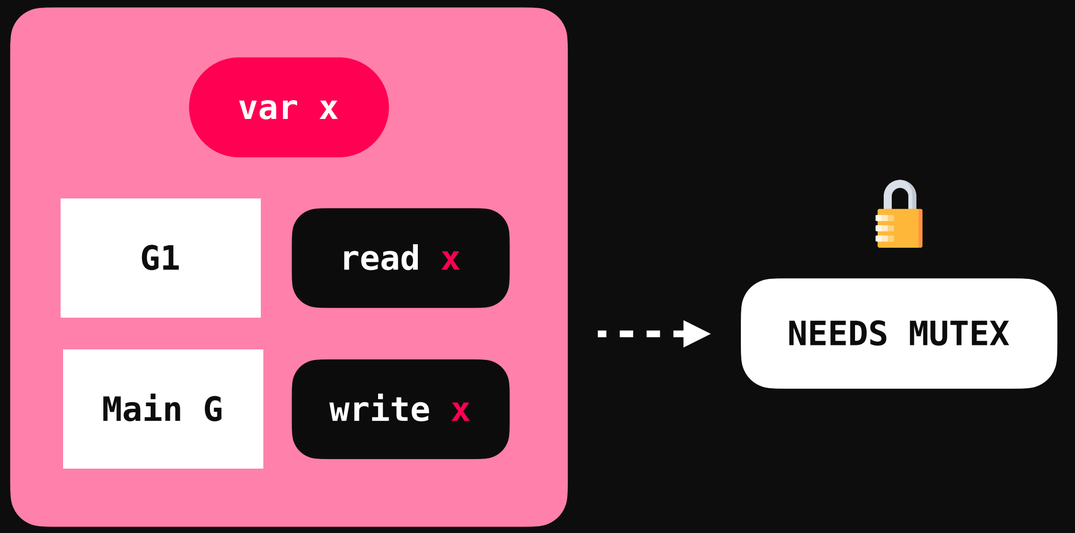


IS THAT SO?

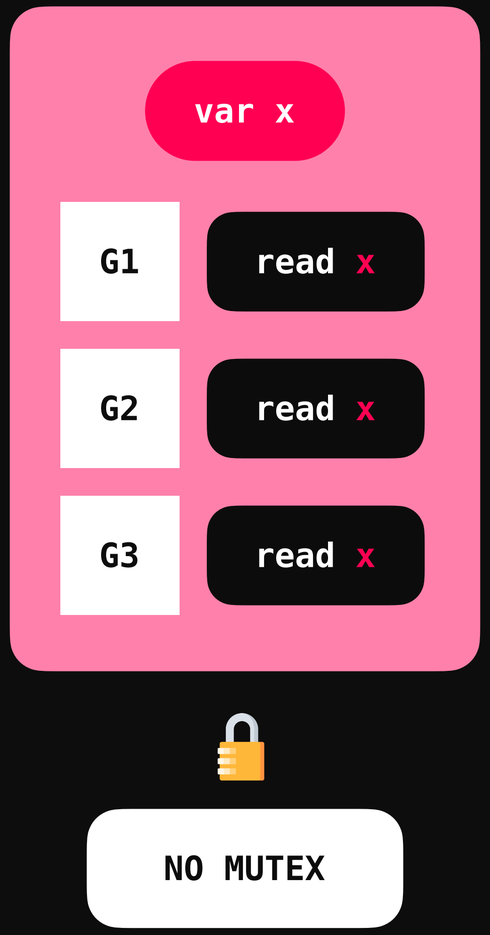

G1
G2
G3
+1
+1
+1
0
0
1
READ
WRITE
READ
WRITE
READ
WRITE
1
1
2
lock
unlock
MUTEX
var i
2

Single Go Routine Context

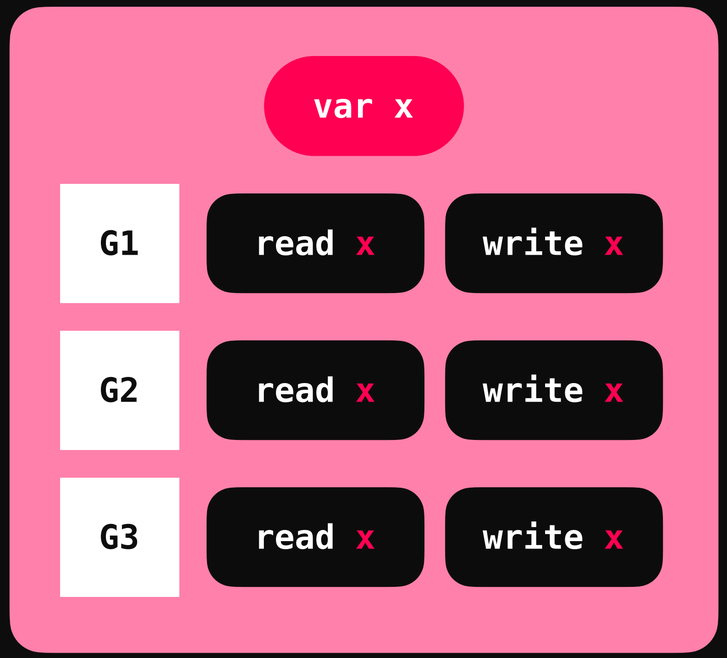
Main Go Routine Context



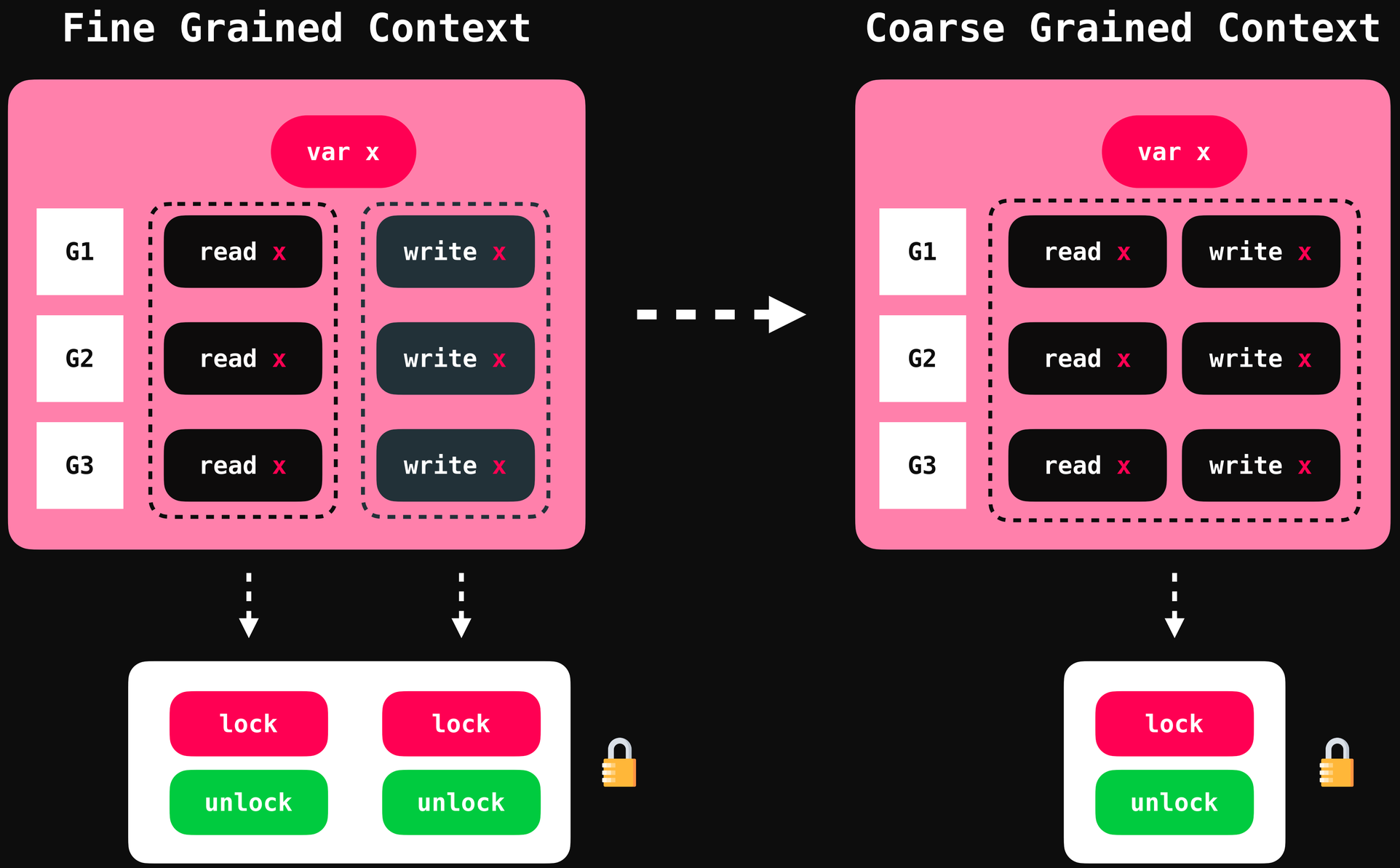
Multiple Go Routines
Read Only Context

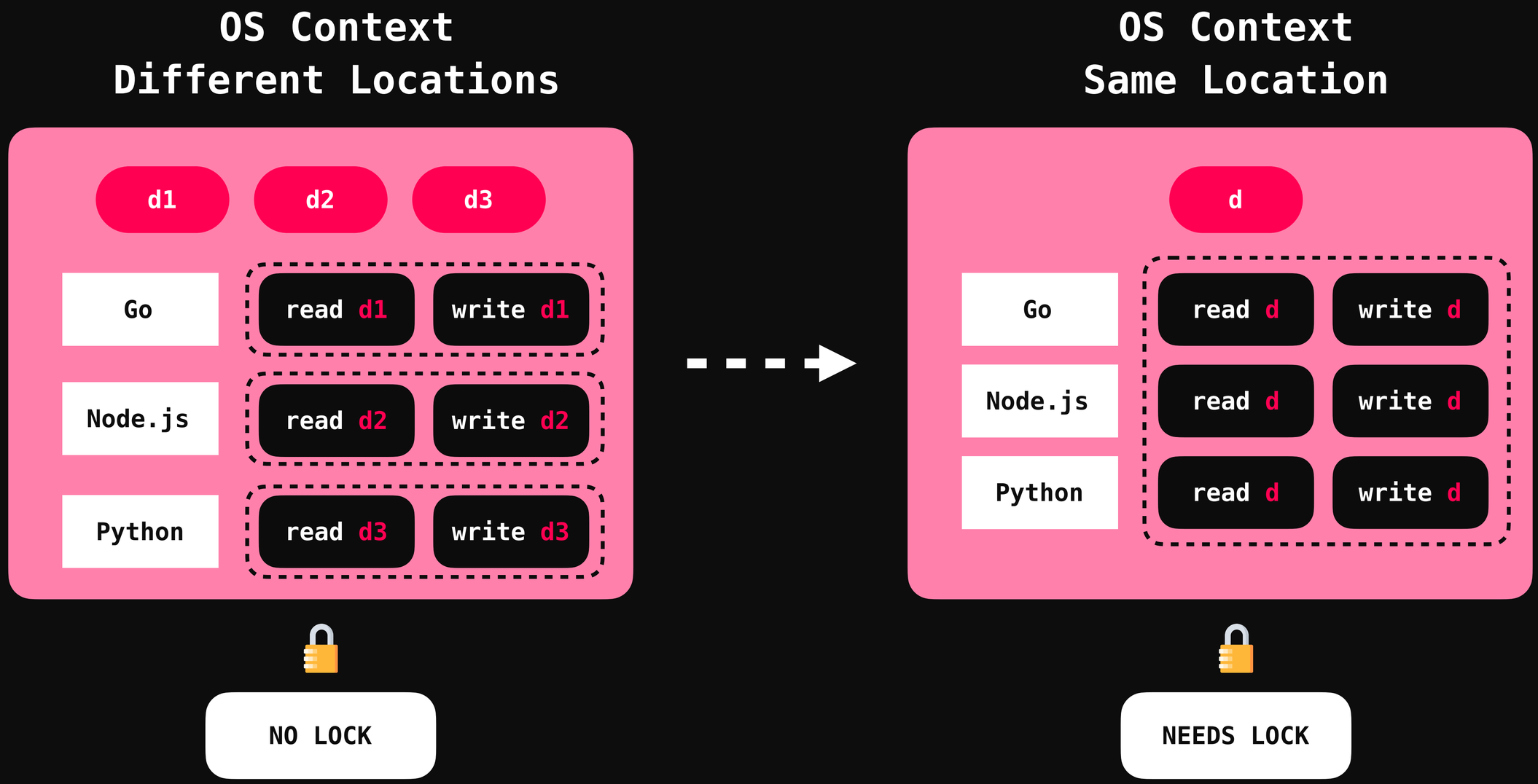


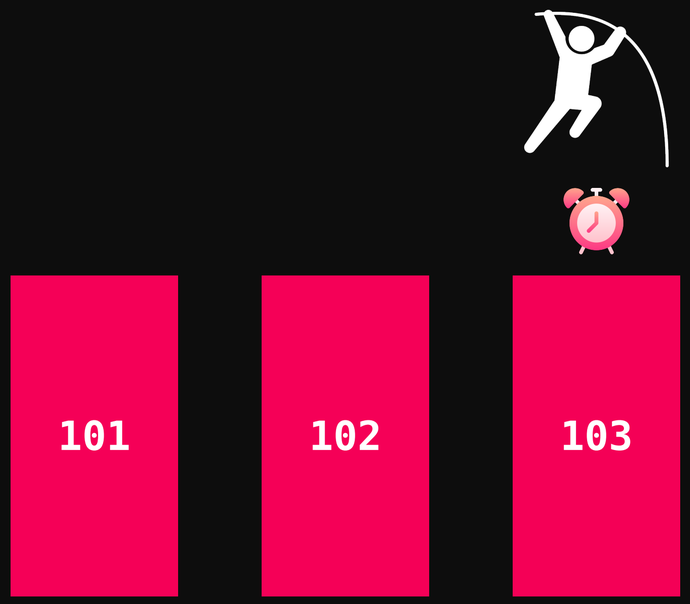
Multiple Go Routines Read Write Context

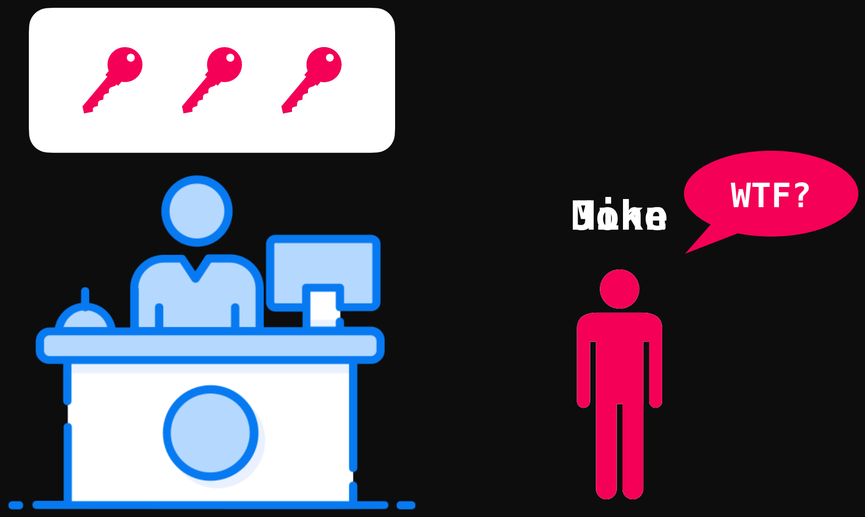


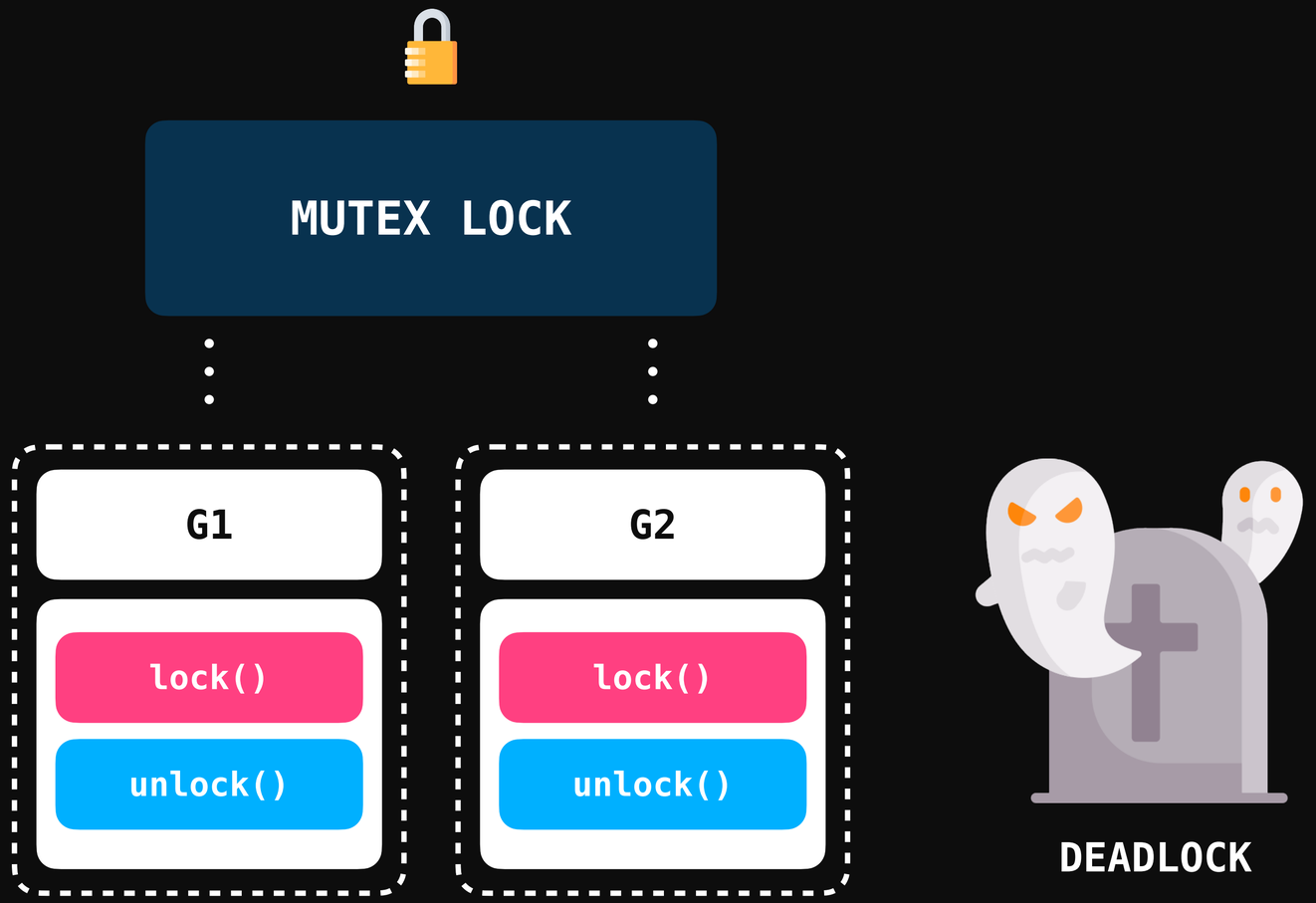
NEEDS MUTEX


Fine Grained Context
var x
G1
read x
write x
G2
read x
write x
G3
read x
write x
lock
lock
unlock
unlock
Coarse Grained Context
var x
G1
read x
write x
G2
read x
write x
G3
read x
write x
lock
unlock


OS Context
Different Locations
d1
d2
d3
Go
Node.js
Python
read d1
write d1
read d2
write d2
read d3
write d3
NO LOCK
OS Context
Same Location
d
Go
Node.js
Python
read d
write d
read d
write d
read d
write d
NEEDS LOCK


101
102
103


WTF?


MUTEX LOCK
G1
lock()
unlock()
G2
lock()
unlock()
DEADLOCK

Mike

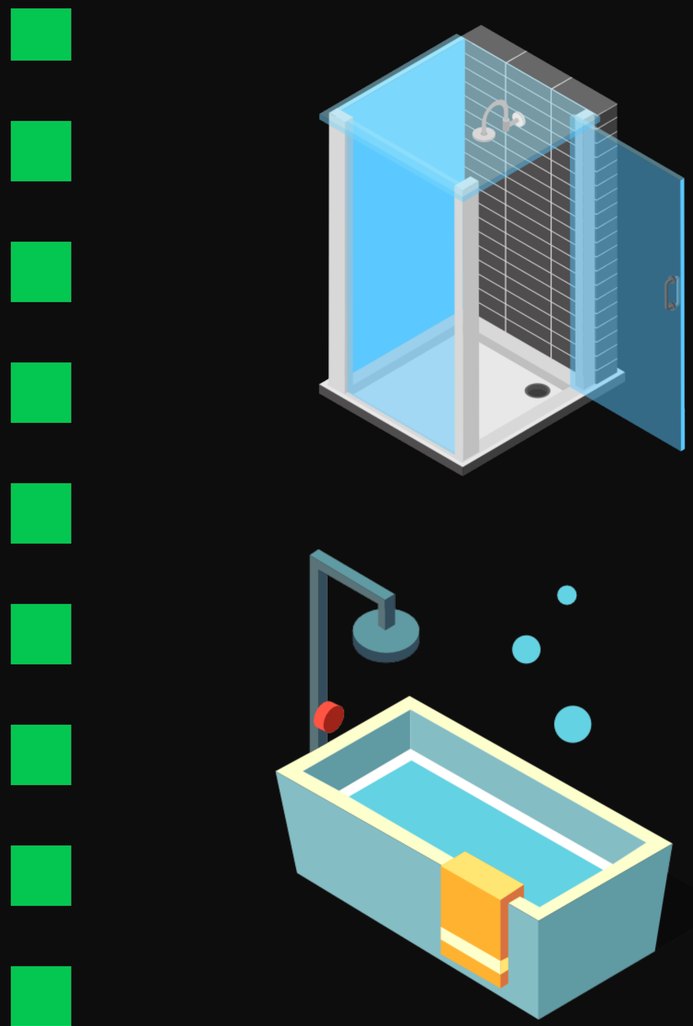
17m

Jane

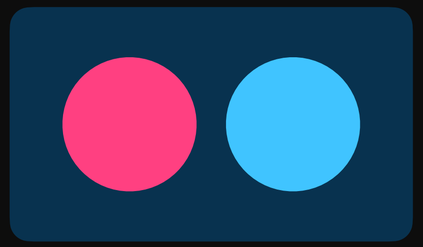
30m

John

15m



Mutex



released G1

G1 work

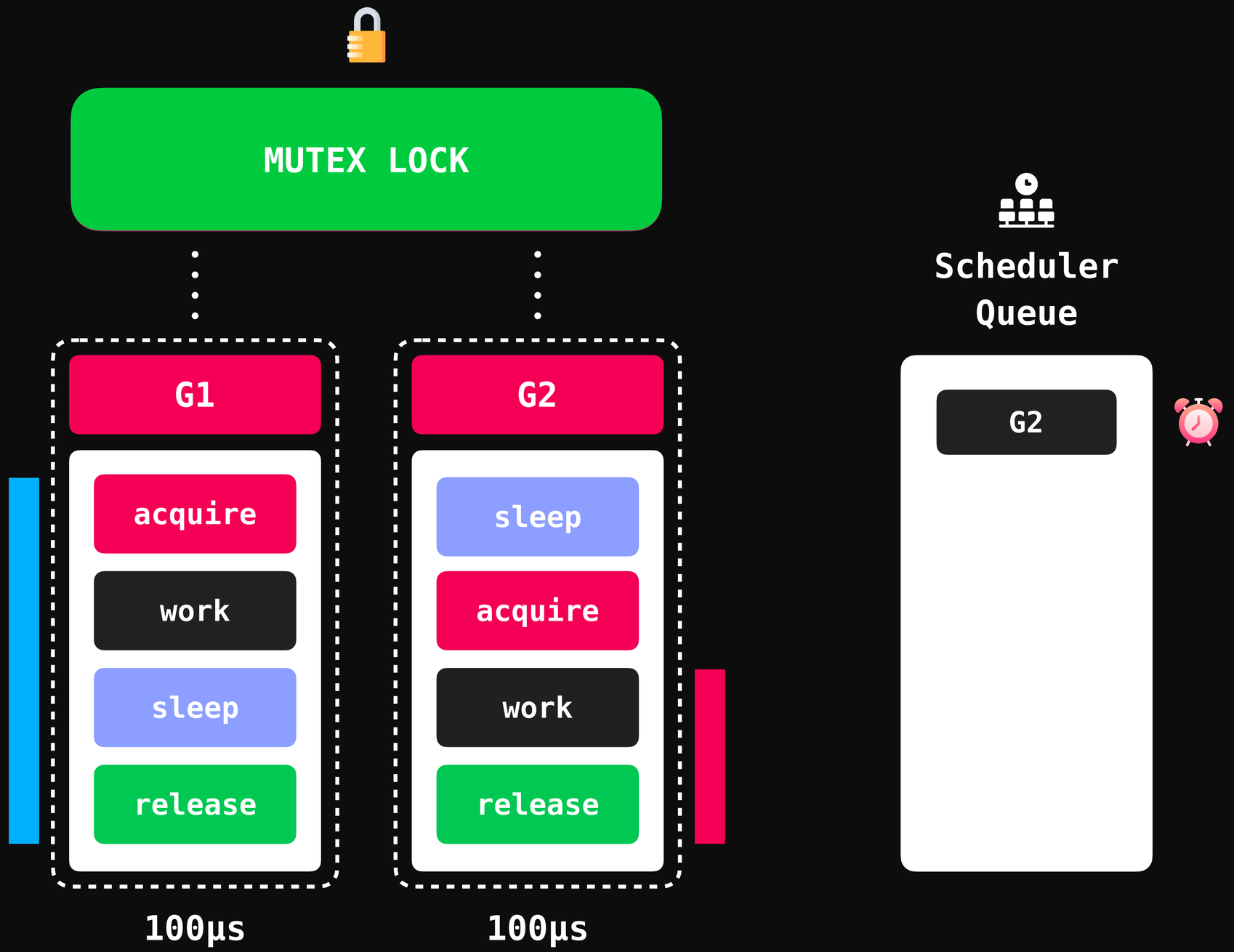
released G2

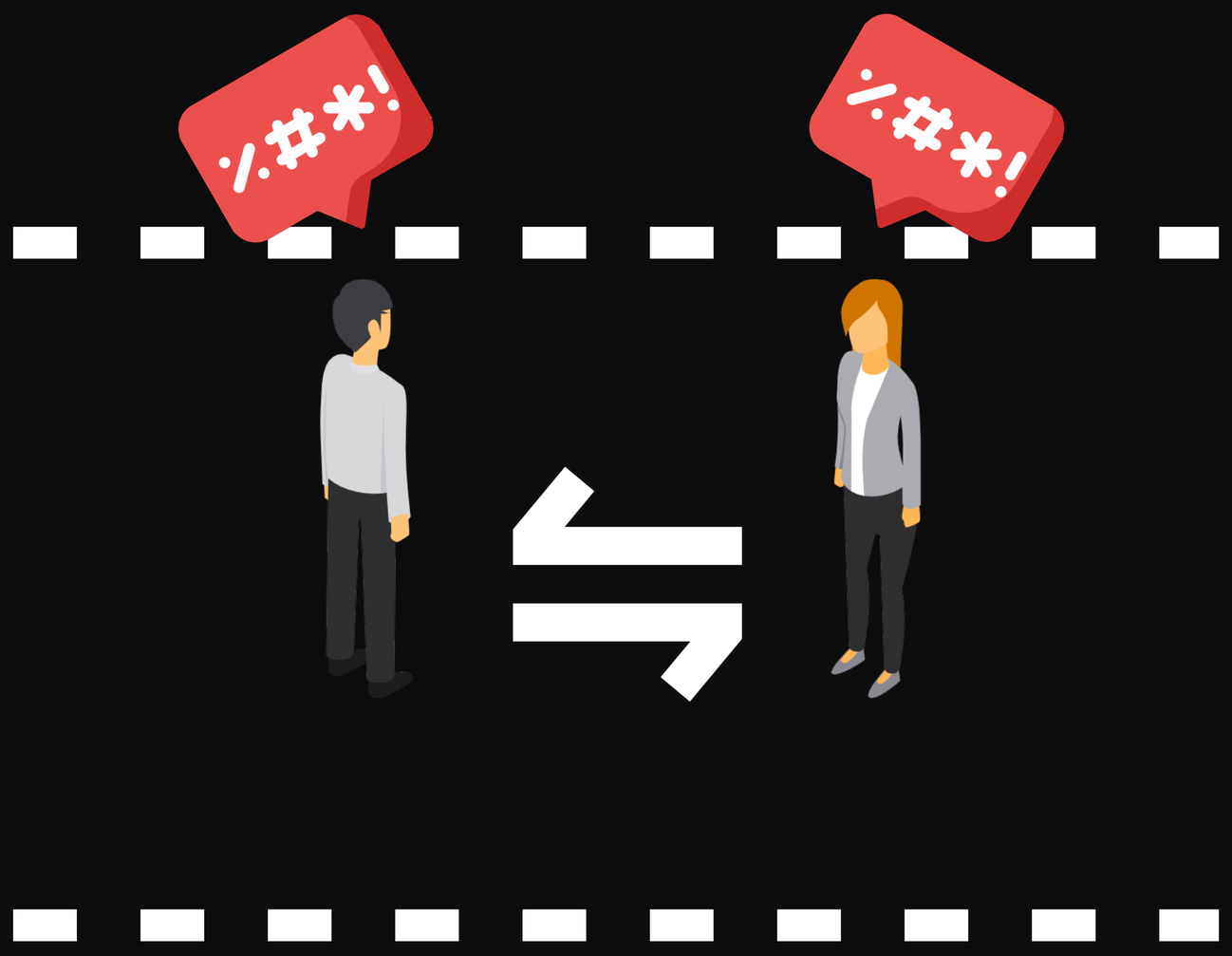
G2 work

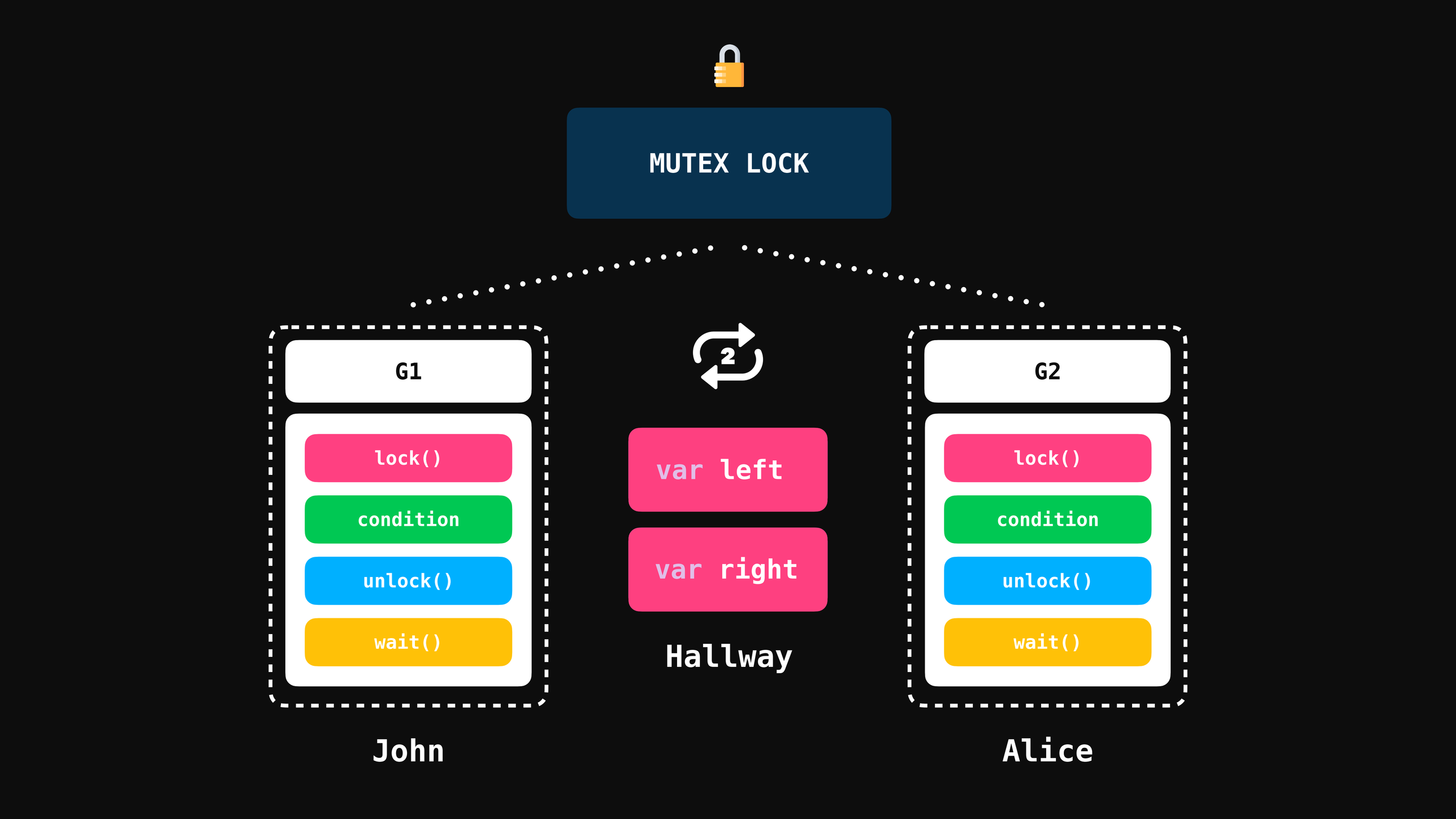
3

REFILL

1

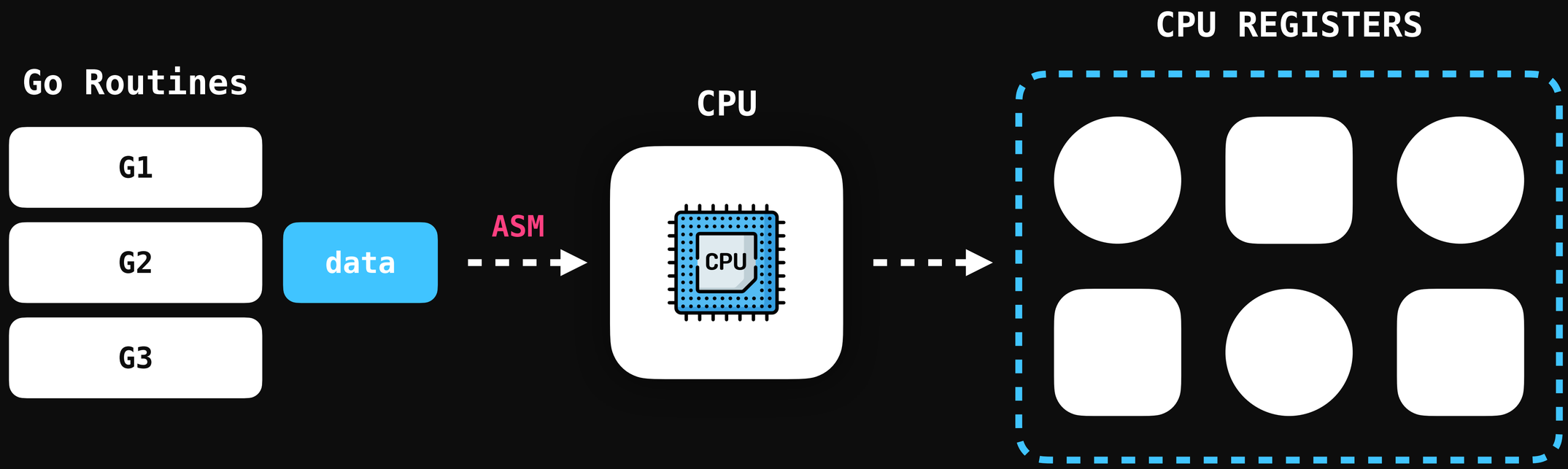

MUTEX LOCK
G1
acquire
work
sleep
release
G2
sleep
acquire
work
release
100µs
100µs
Scheduler
Queue
G2

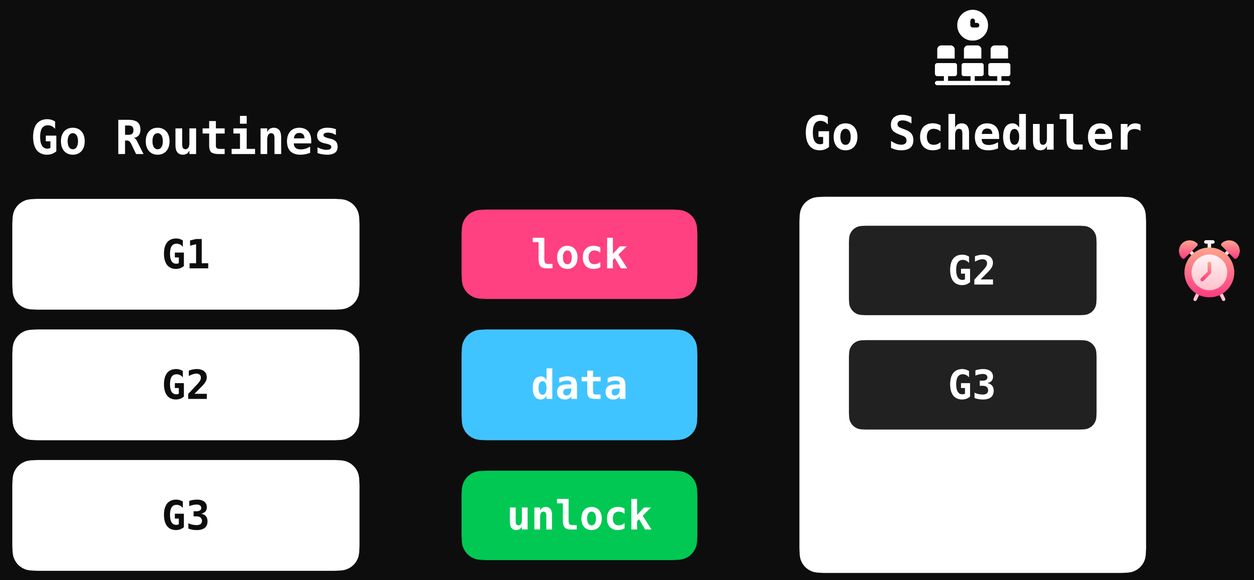



MUTEX LOCK
G1
lock()
condition
unlock()
wait()
2
var left
var right
Hallway
G2
lock()
condition
unlock()
wait()
John
Alice

sync.Locker

```
type Locker interface {
    Lock()
    Unlock()
}
```


Go Routines
G1
G2
G3
data
ASM
CPU
CPU
CPU REGISTERS


Go Routines
G1
G2
G3
lock
data
unlock
Go Scheduler
G2
G3

- Prefer **Atomics** over **Mutexes** for **simple data**
- Use **Mutex** for write heavy scenarios
- Use **RWMutex** for read heavy / mixed scenarios
- Prefer **Fine Grained Context** where possible (limit the context)
- **Don’t use** the mutex **longer** than you need to (**avoid extra hold time**)
- Use **types**, a **local mutex** and **methods** over direct mutex calls
- Avoid **Contention** by **distributing work evenly**
- Avoid **Starvation** by testing for **mutex fairness**
- Use **sync.Locker** for generic code that uses a mutex