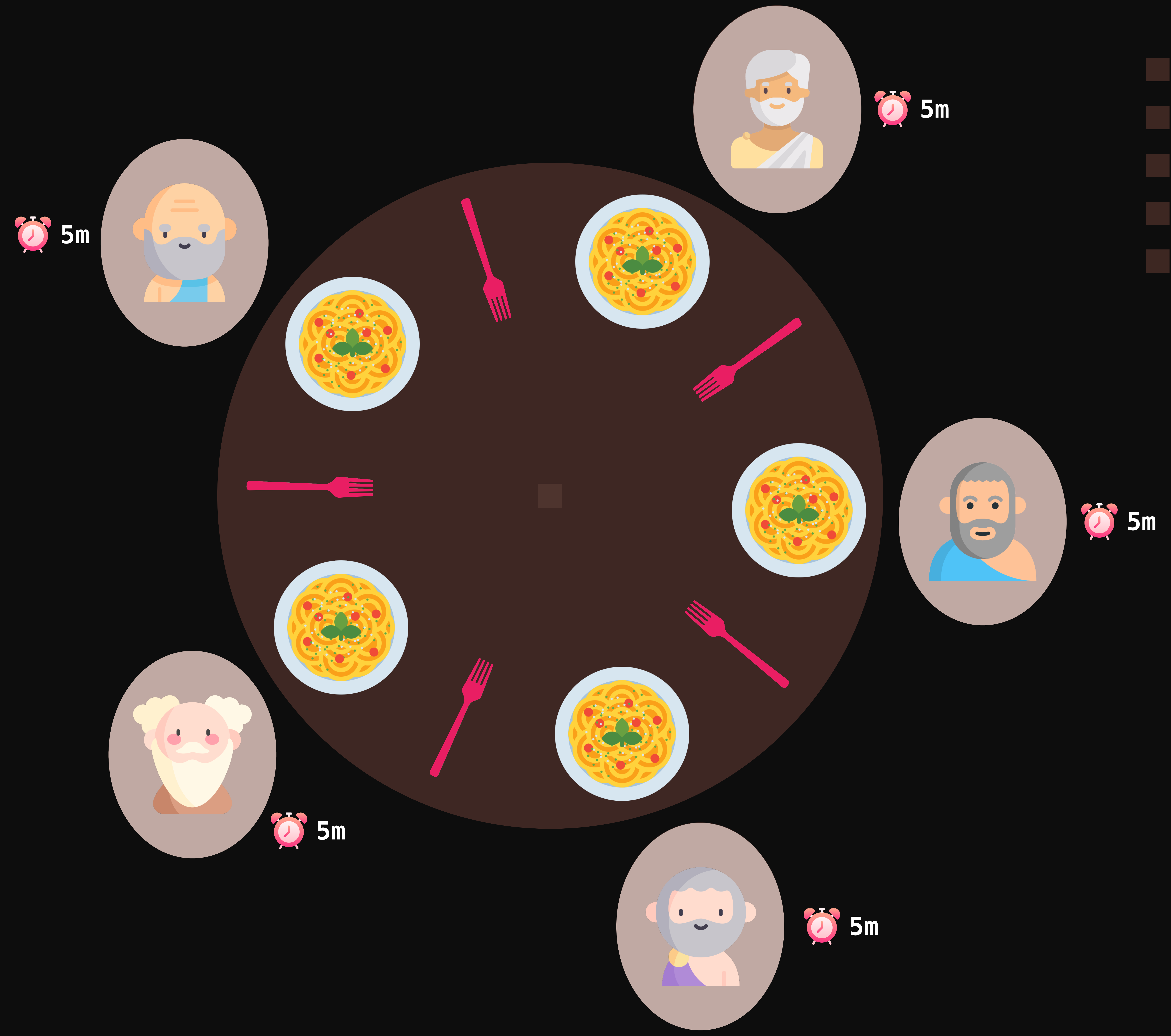


- ***Think*** *till* ***left fork*** *is available; when it is →* ***Pick it up***
- ***Think*** *till* ***right fork*** *is available; when it is →* ***Pick it up***
- ***Eat*** *for a fixed amount of time when* ***both forks*** *are held*
- ***Put*** *the* ***right fork down*** *→* ***Put*** *the* ***left fork down***
- ***Repeat*** *the process*

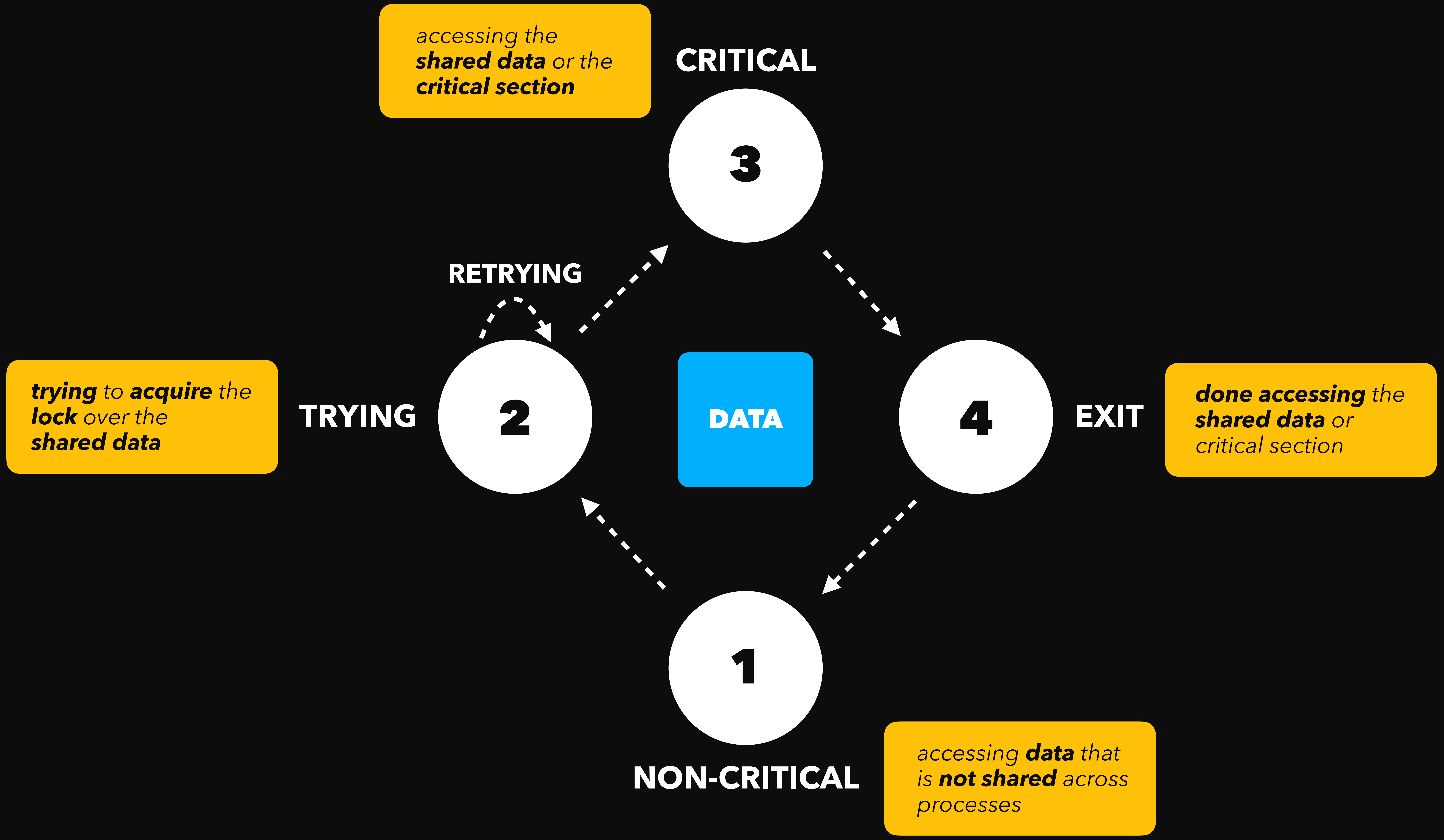
DEADLOCK

STARVATION

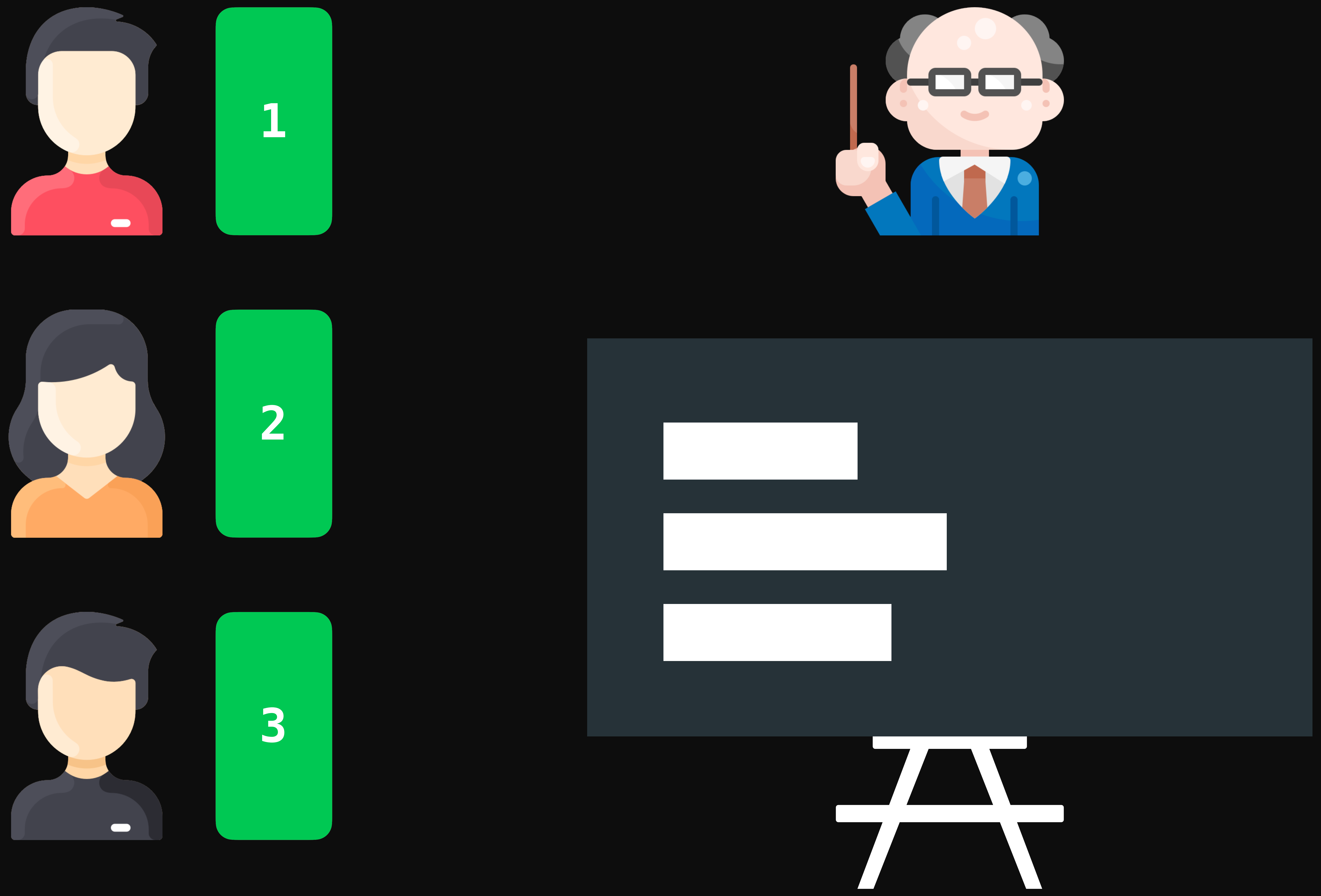
LIVELOCK

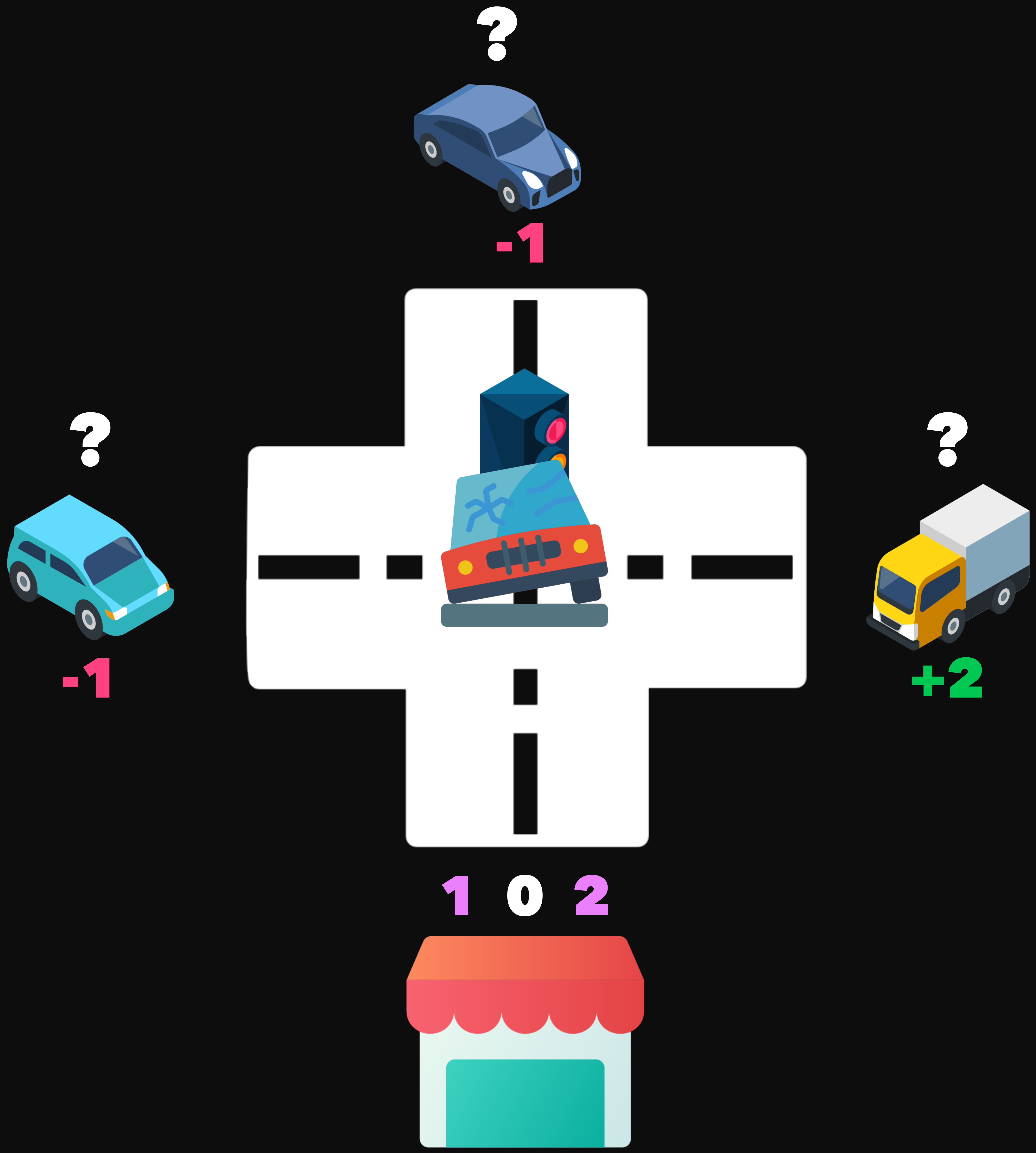
MUTUAL EXCLUSION

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


accessing the **shared data** or the **critical section**
CRITICAL
3
RETRYING
trying to **acquire** the **lock** over the **shared data**
TRYING
2
DATA
4
EXIT
done accessing the **shared data** or critical section
1
NON-CRITICAL
accessing **data** that is **not shared** across processes

LOCKS	MUTEX
READERS–WRITER LOCKS	RWMUTEX
RECURSIVE LOCKS	UNAVAILABLE
SEMAPHORES	INTERNAL
MONITORS	INTERNAL

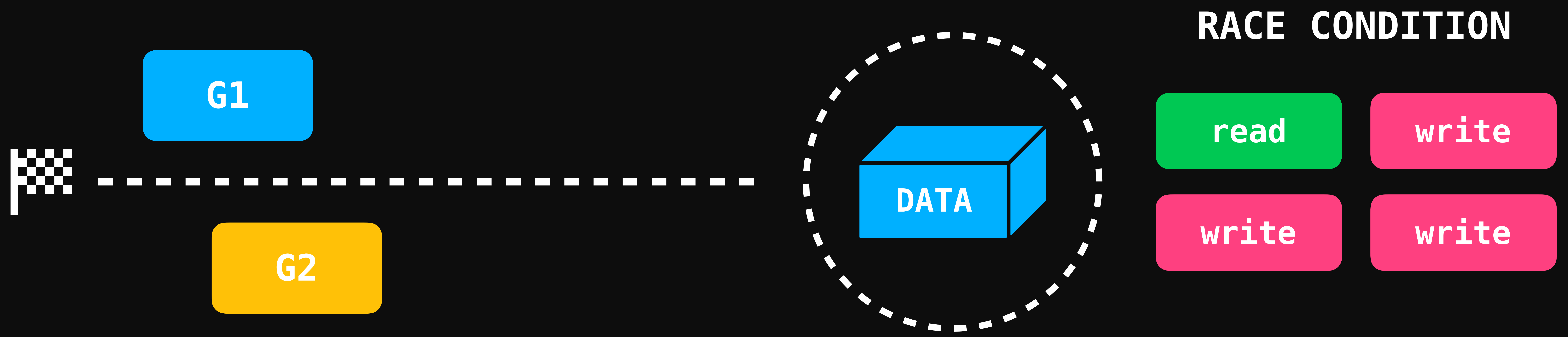

1
2
3


?
-1
?
-1
?
+2
1 0 2

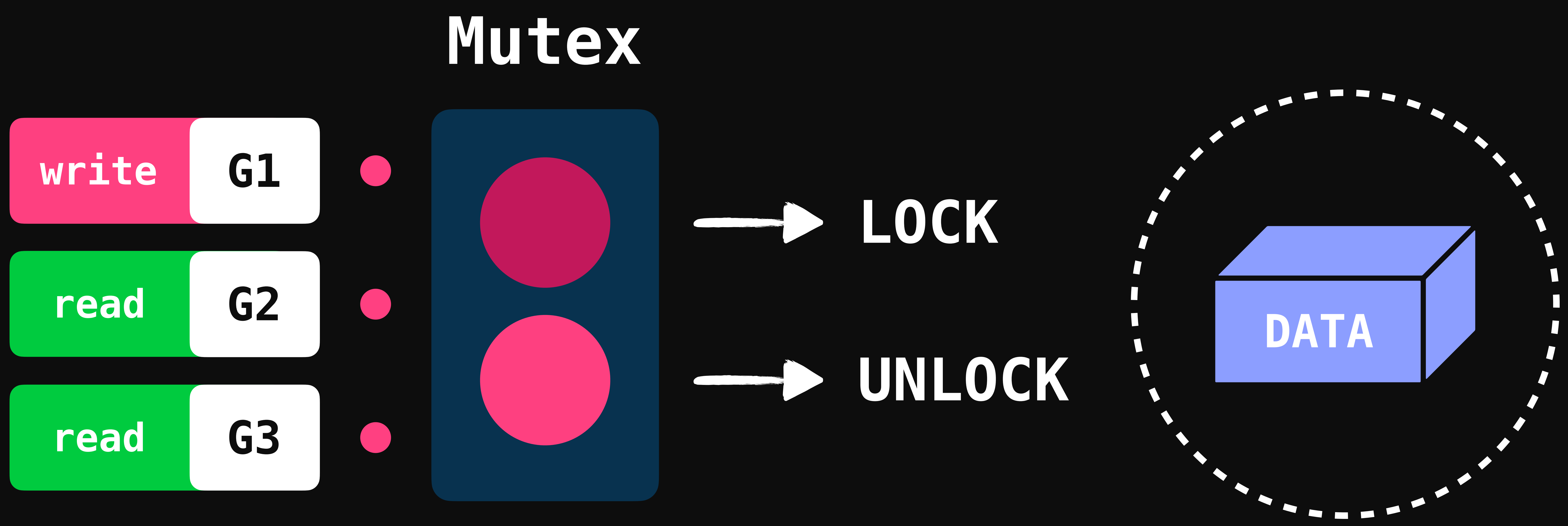
ORDER

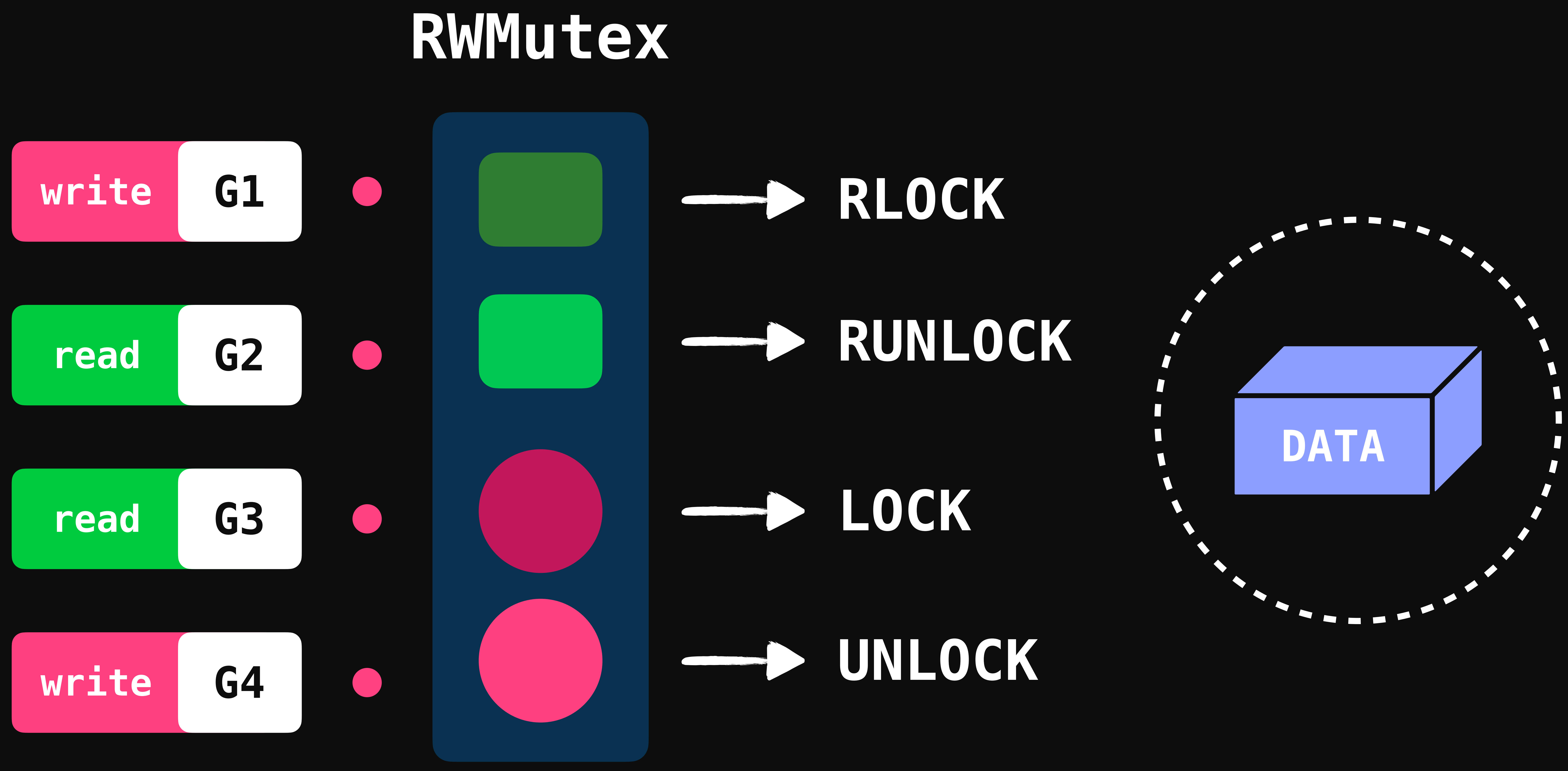
RESULT

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**

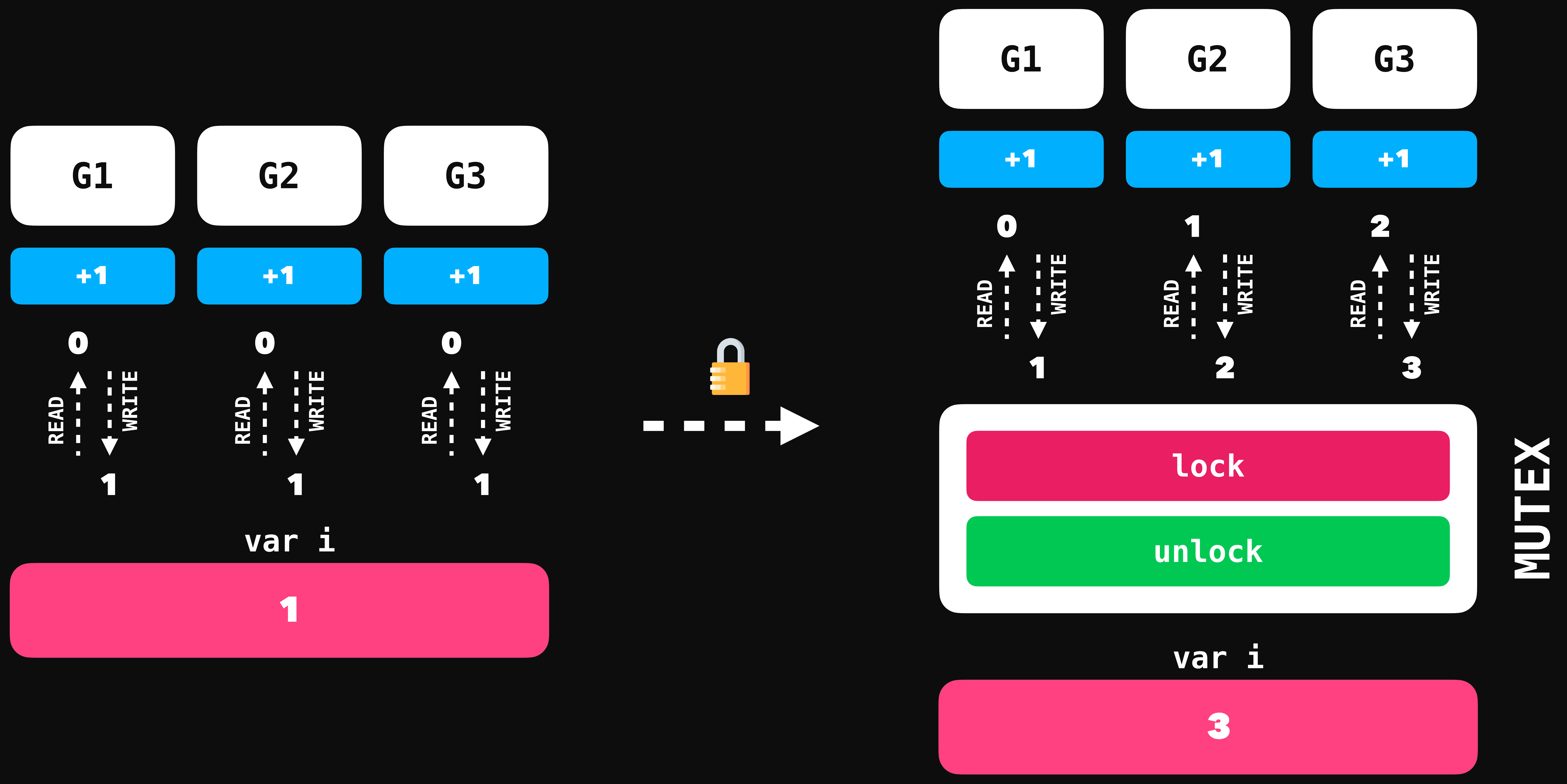
increment value of **i**

store value of **i**

INDIVISIBLE

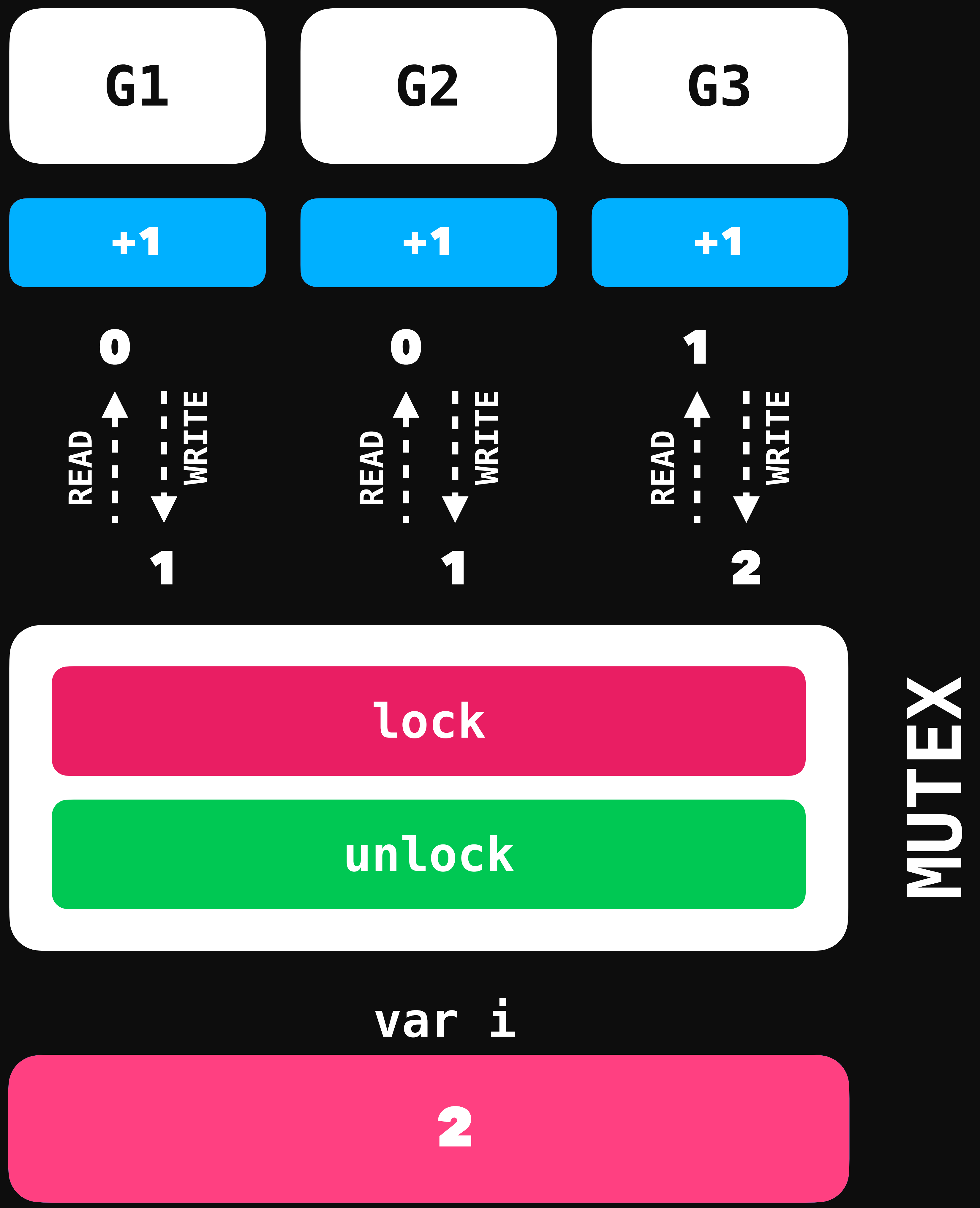
UNINTERRUPTIBLE

ATOMIC


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

Atomic	Not Atomic	Atomic
READ	WRITE	INCREMENT

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

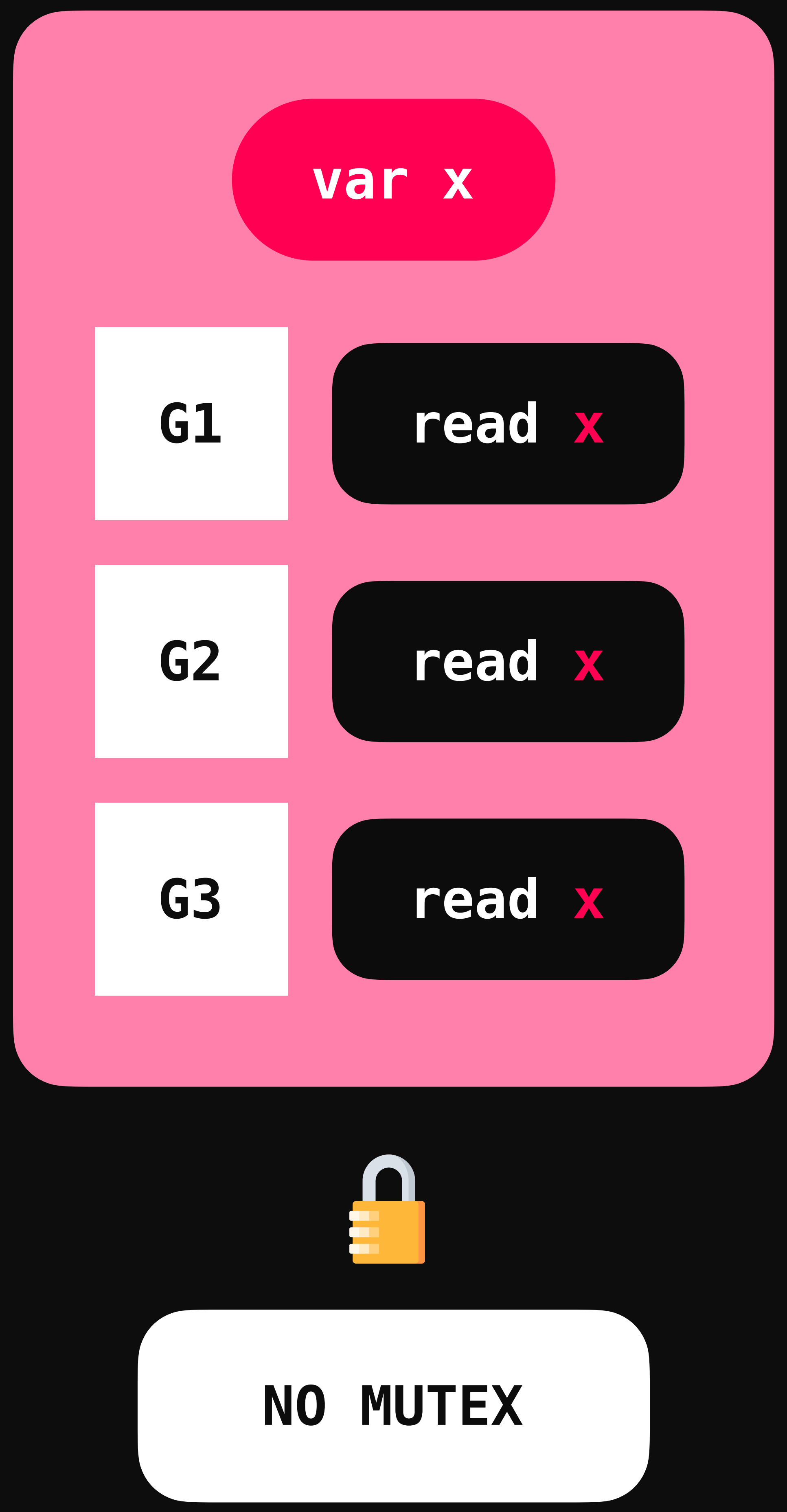
var x

G1 — read x

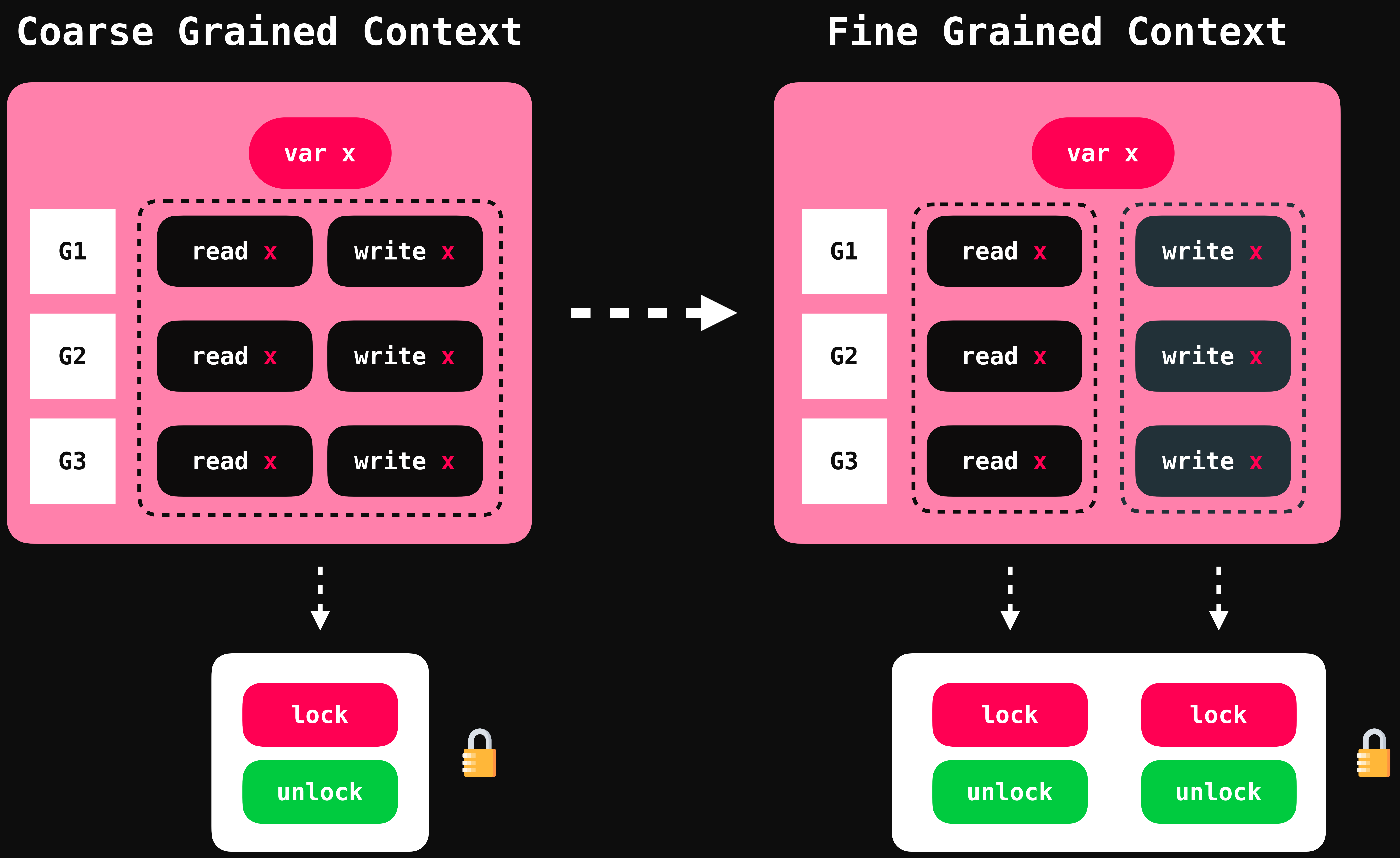
Main G — write x

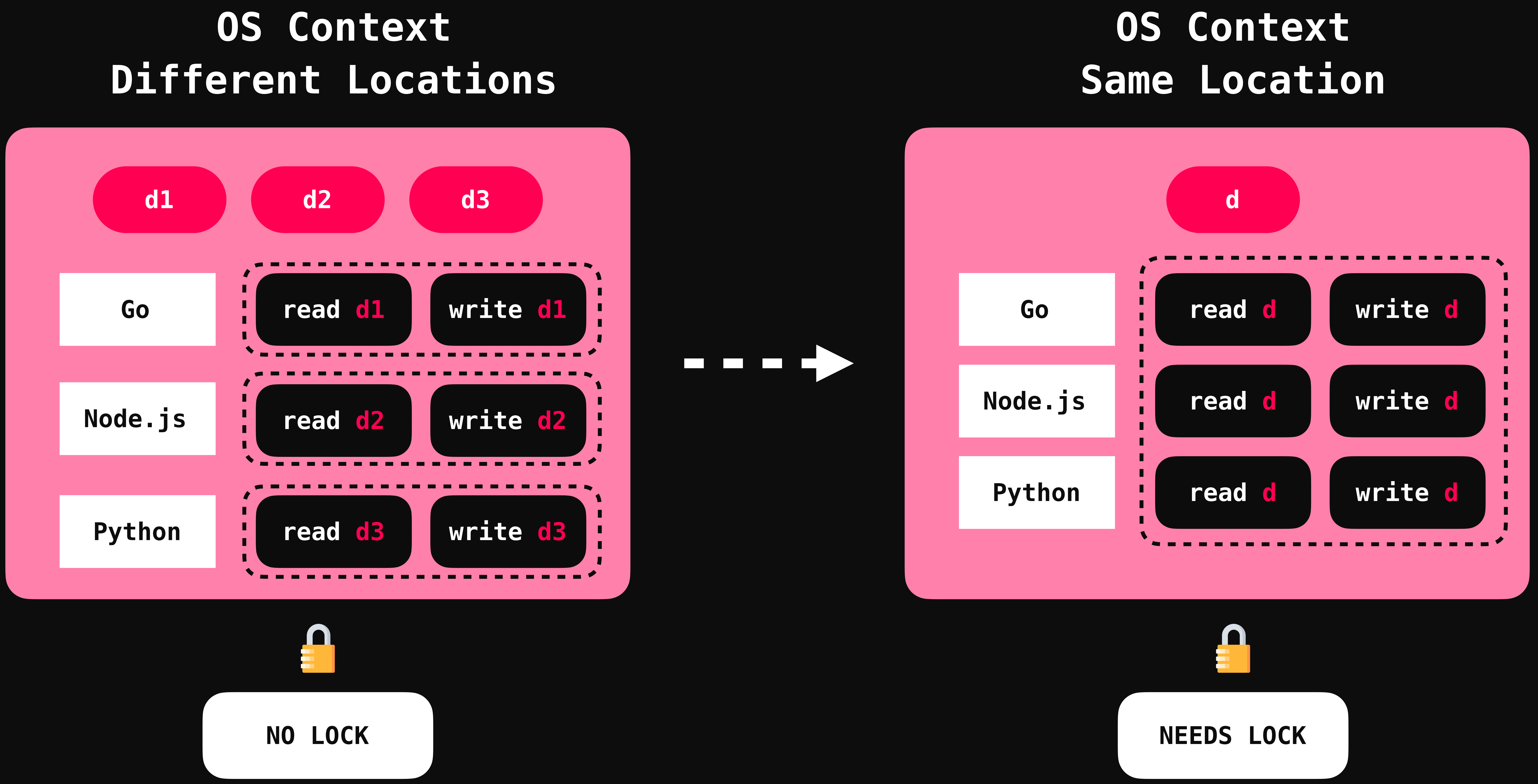
NEEDS MUTEX

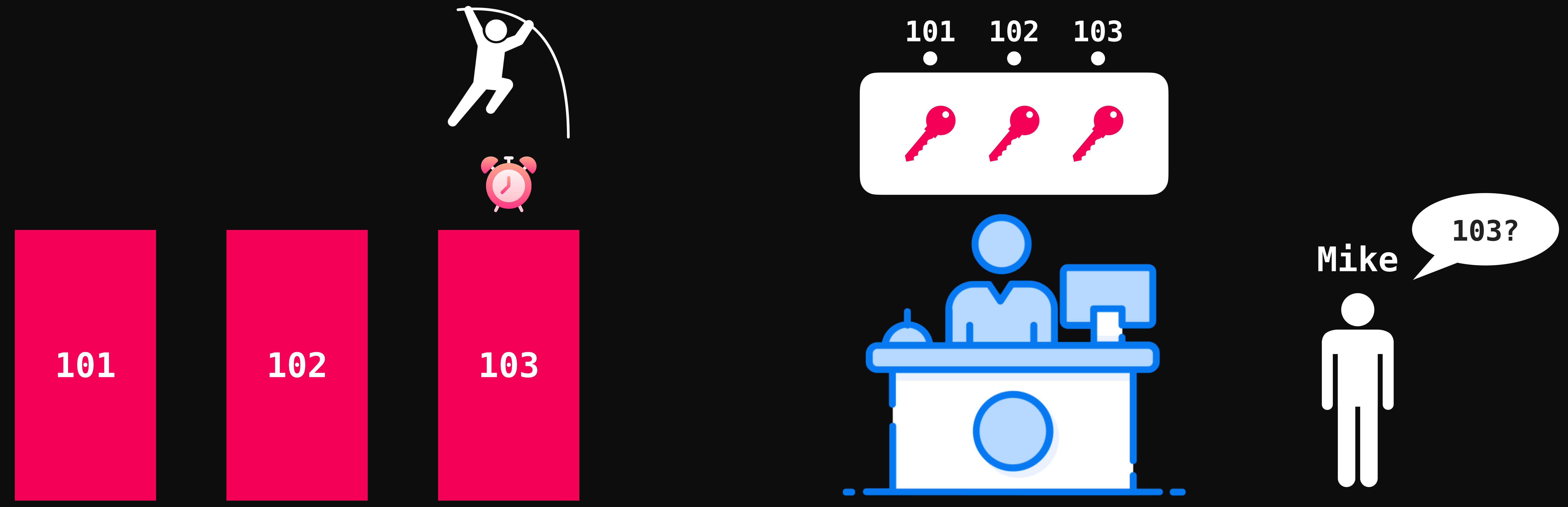
Multiple Go Routines
Read Only Context

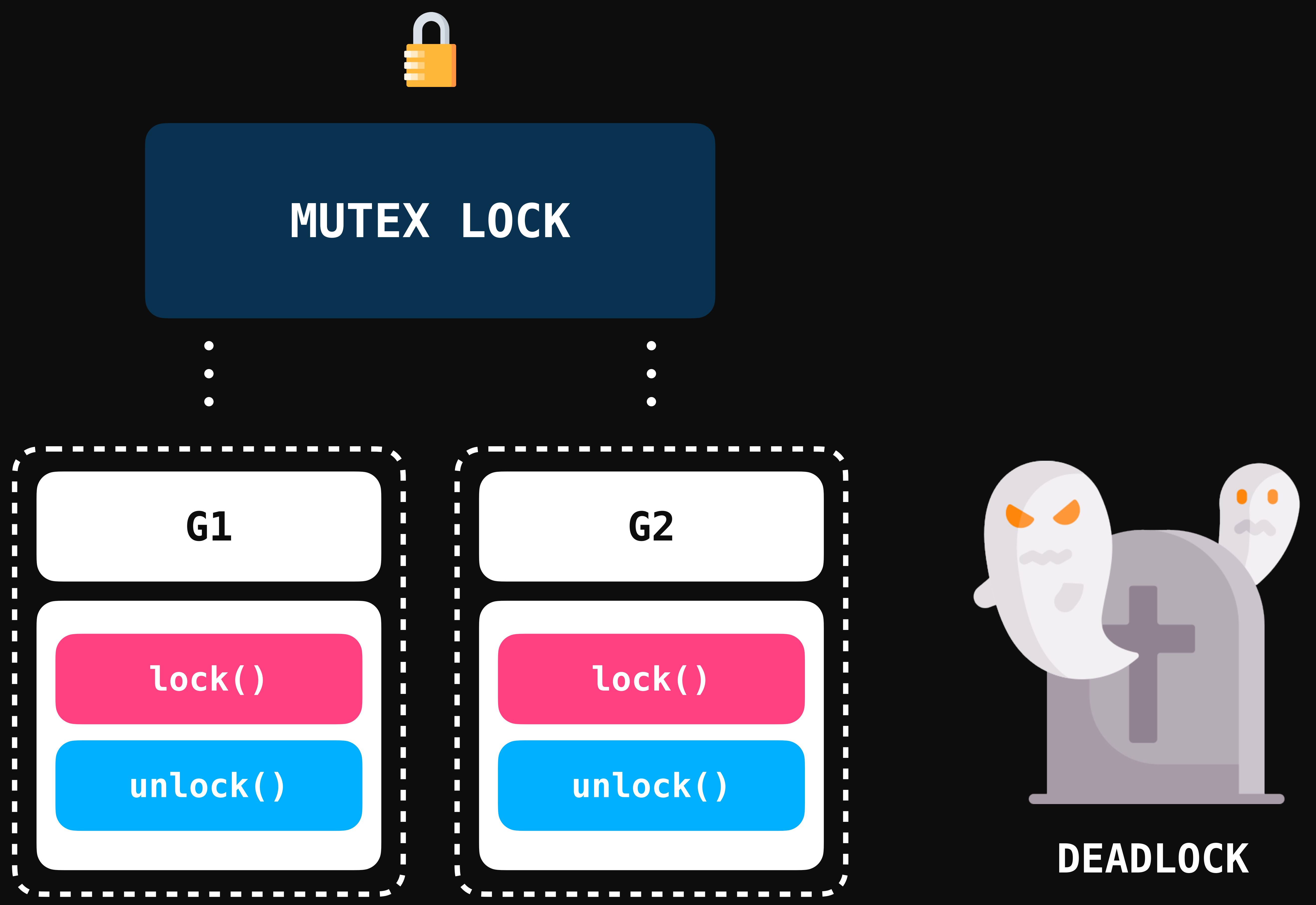



Multiple Go Routines
Read Write Context
var x
G1
read x
write x
G2
read x
write x
G3
read x
write x
NEEDS MUTEX


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


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


101
102
103
101
102
103
Mike
103?

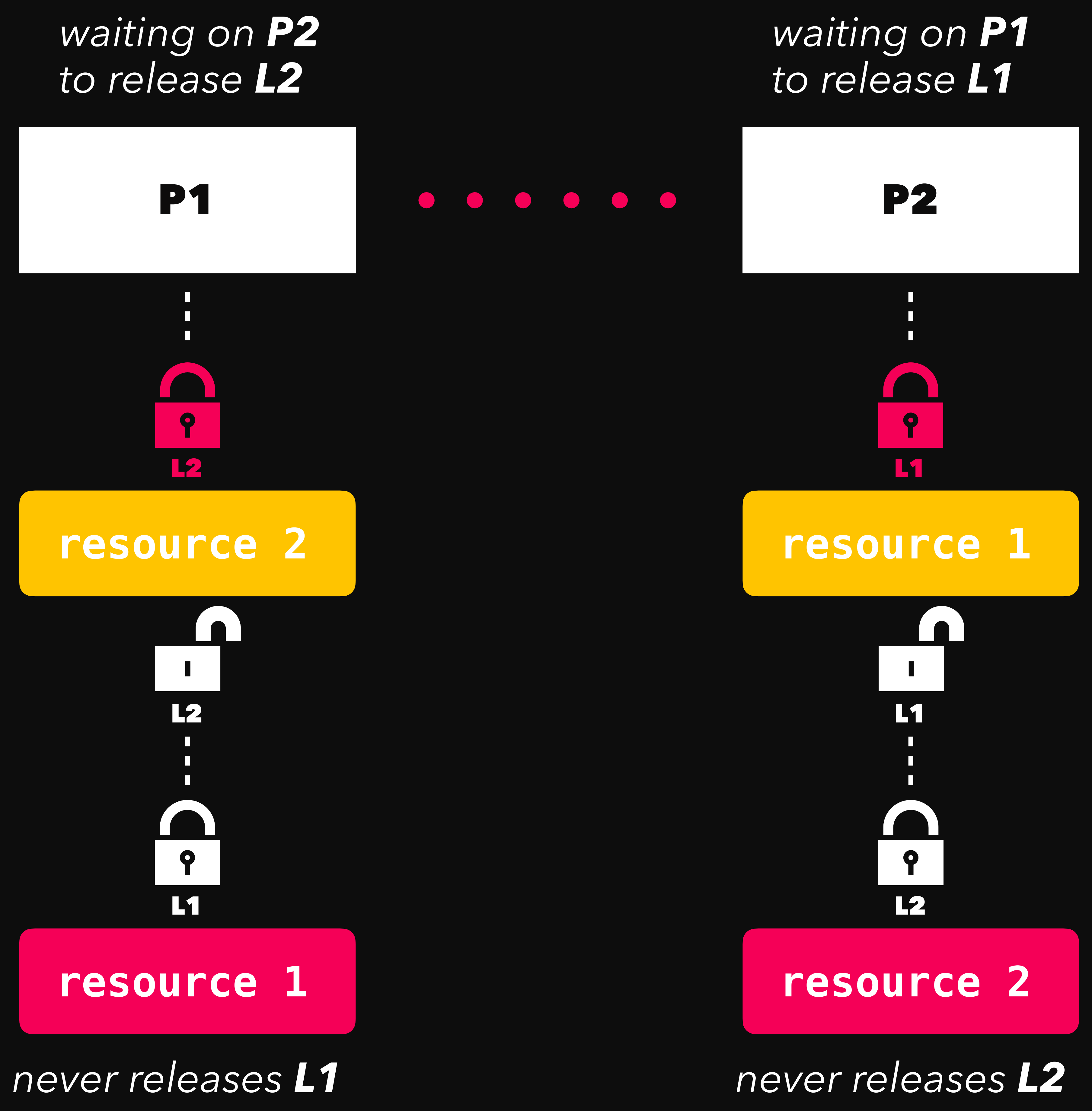

MUTEX LOCK
G1
lock()
unlock()
G2
lock()
unlock()
DEADLOCK

COFFMAN CONDITIONS

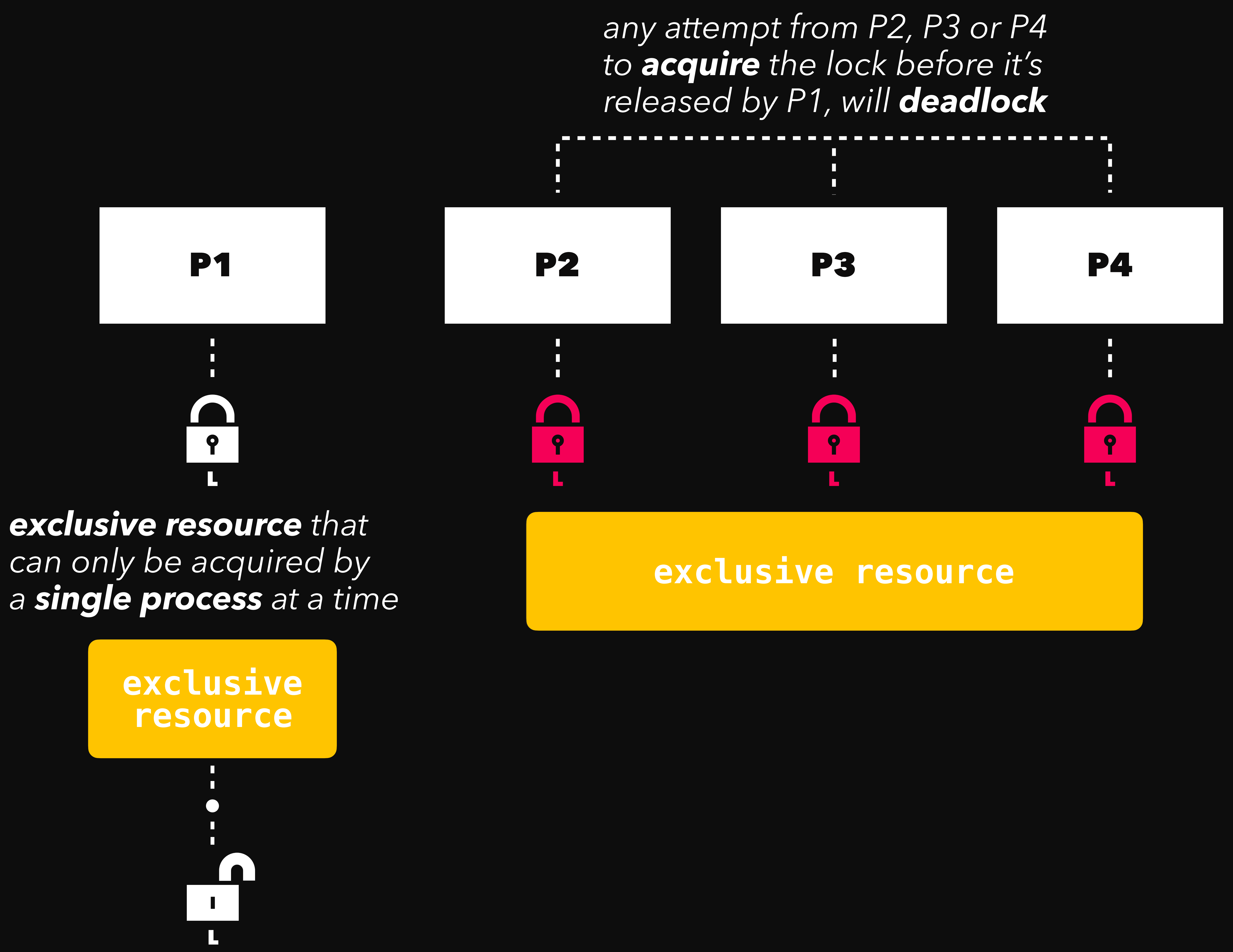
WHAT IS A DEADLOCK?

*A **deadlock** is a state in which each member of the group **waits** for another member, including itself, to take action, such as sending a message or more commonly **releasing** a **lock**.*

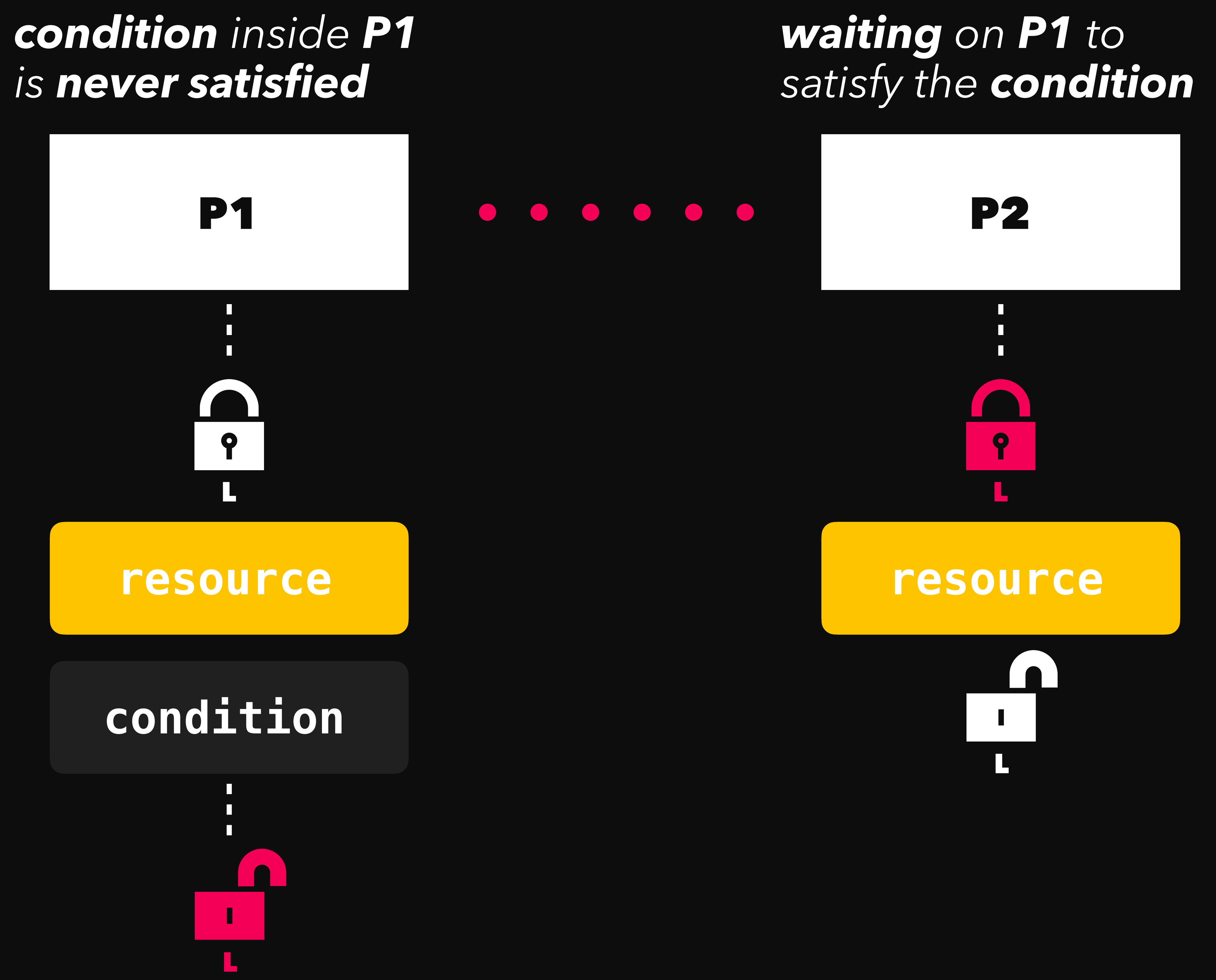
1 CIRCULAR WAIT



2 MUTUAL EXCLUSION



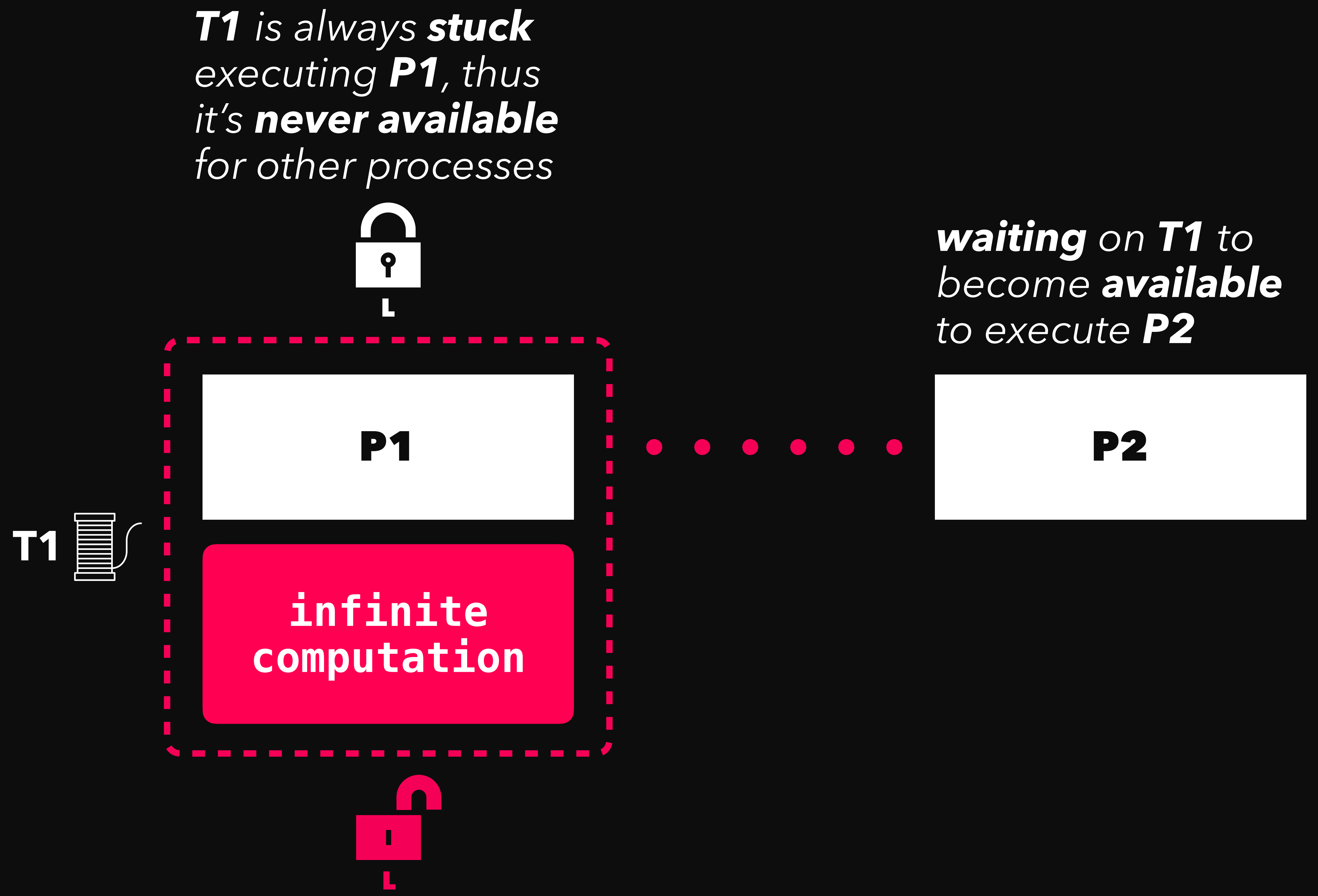
3 HOLD AND WAIT



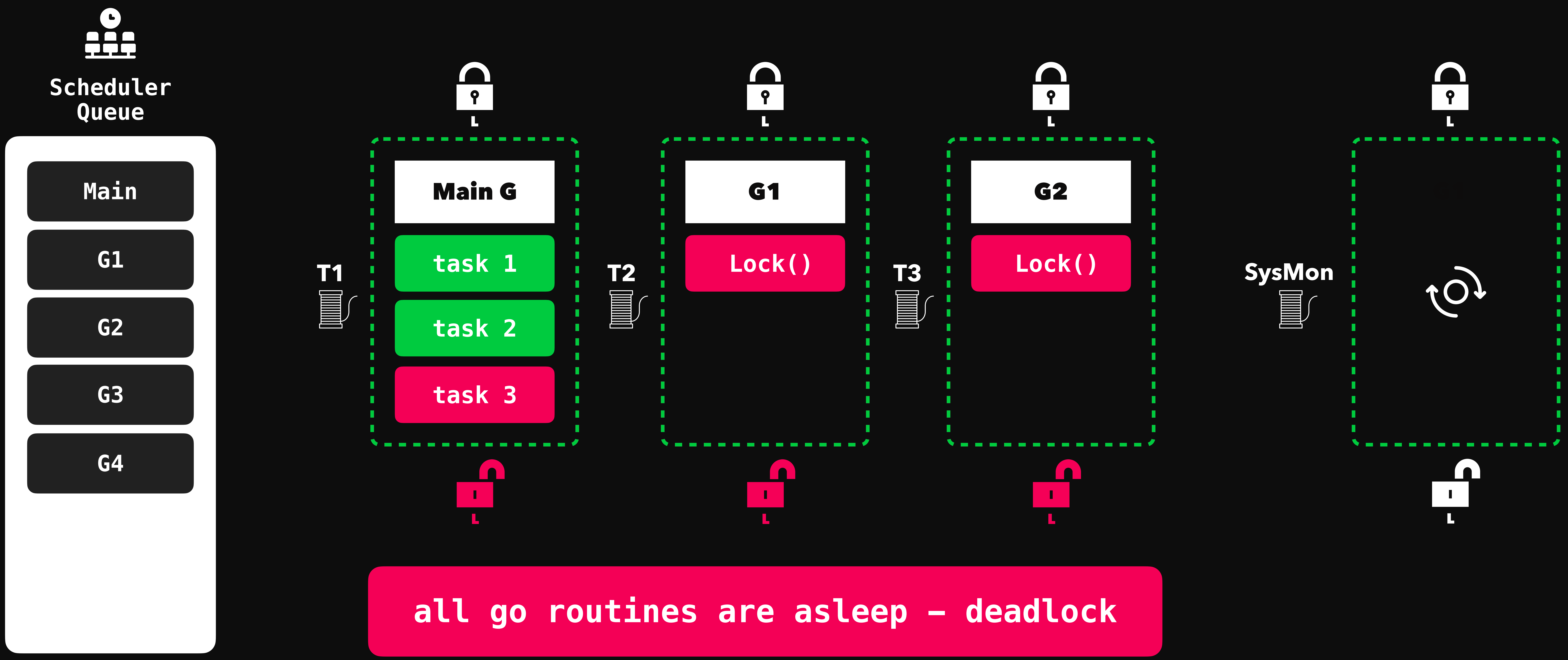
WHAT IS PREEMPTION?

> ***Preemption*** *is the act of temporarily* ***interrupting*** *an* ***executing task****, with the intention of* ***resuming*** *it at a* ***later time****.*

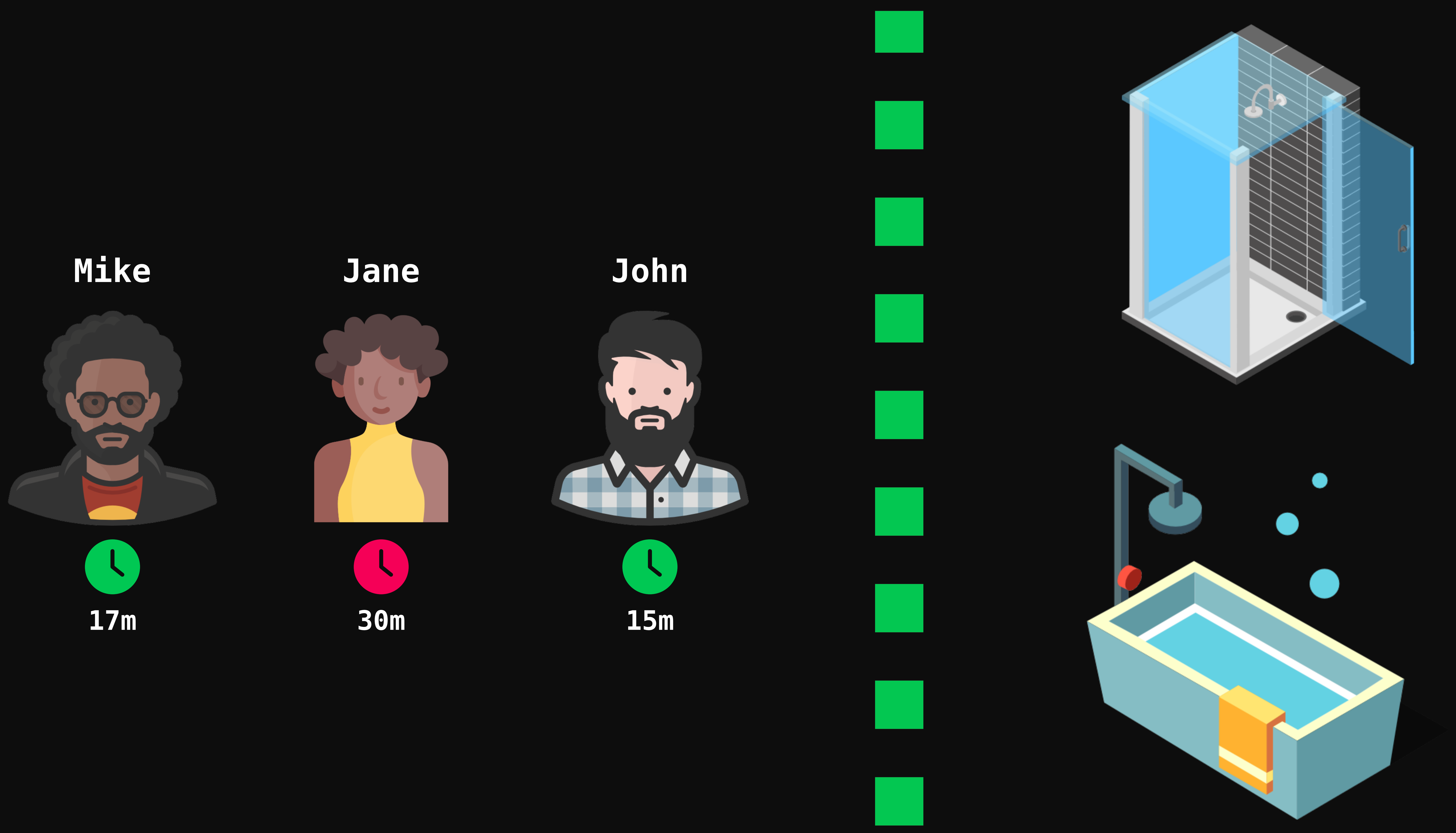
4 NO PREEMPTION

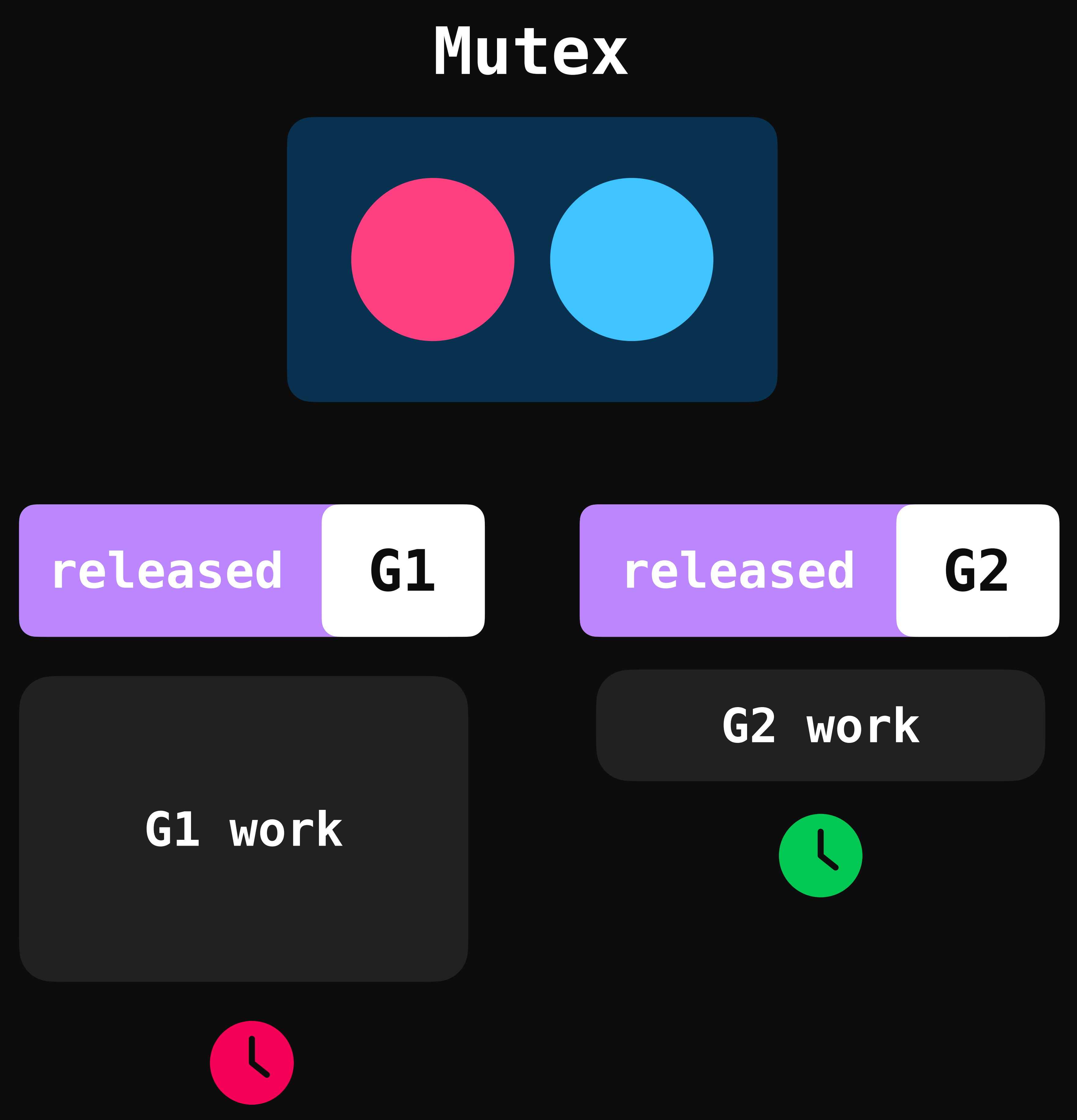


CHECK DEAD



Scheduler
Queue
Main
G1
G2
G3
G4
T1
Main G
task 1
task 2
task 3
T2
G1
Lock()
T3
G2
Lock()
SysMon
all go routines are asleep - deadlock

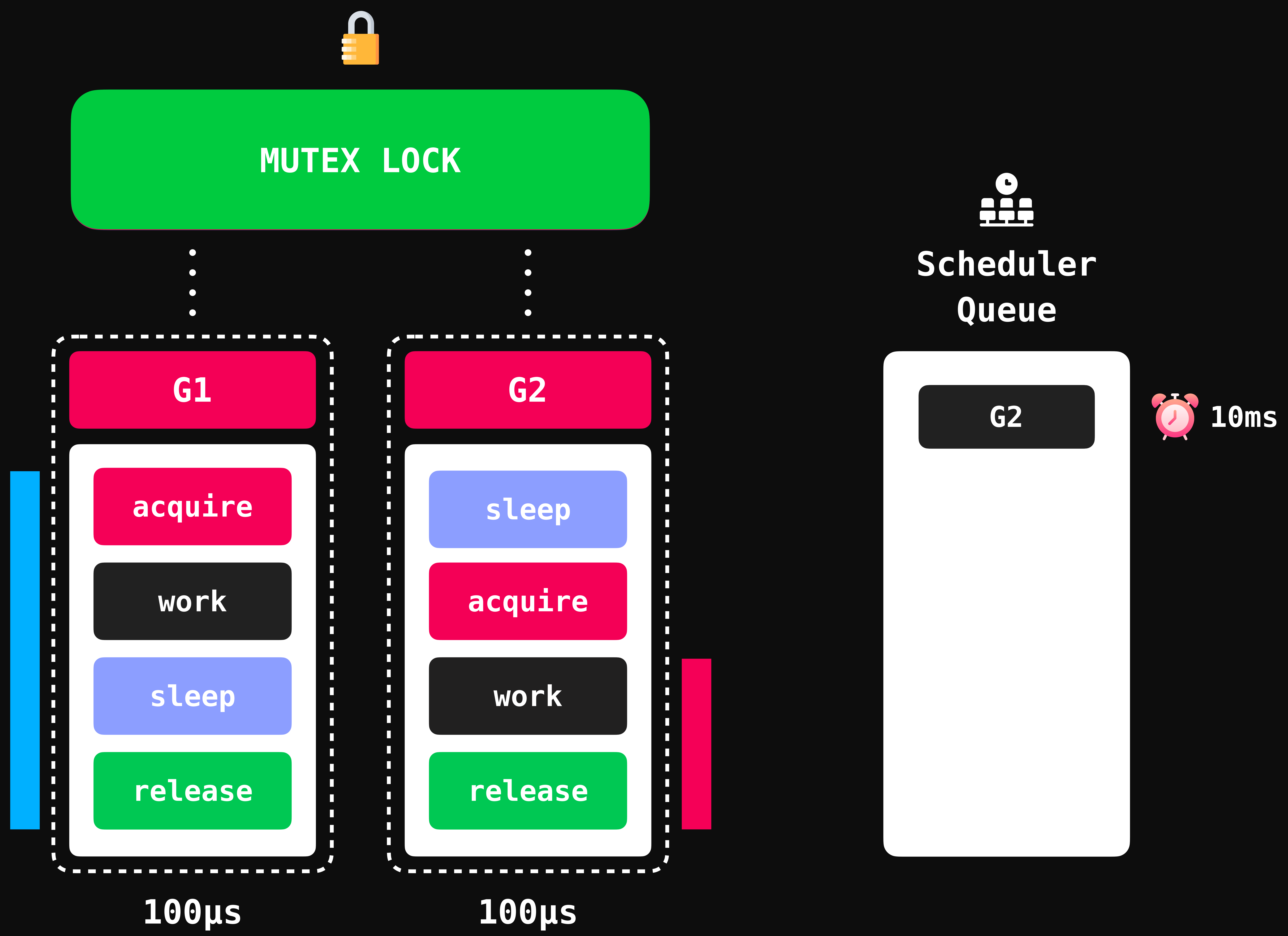

Mike
Jane
John
17m
30m
15m

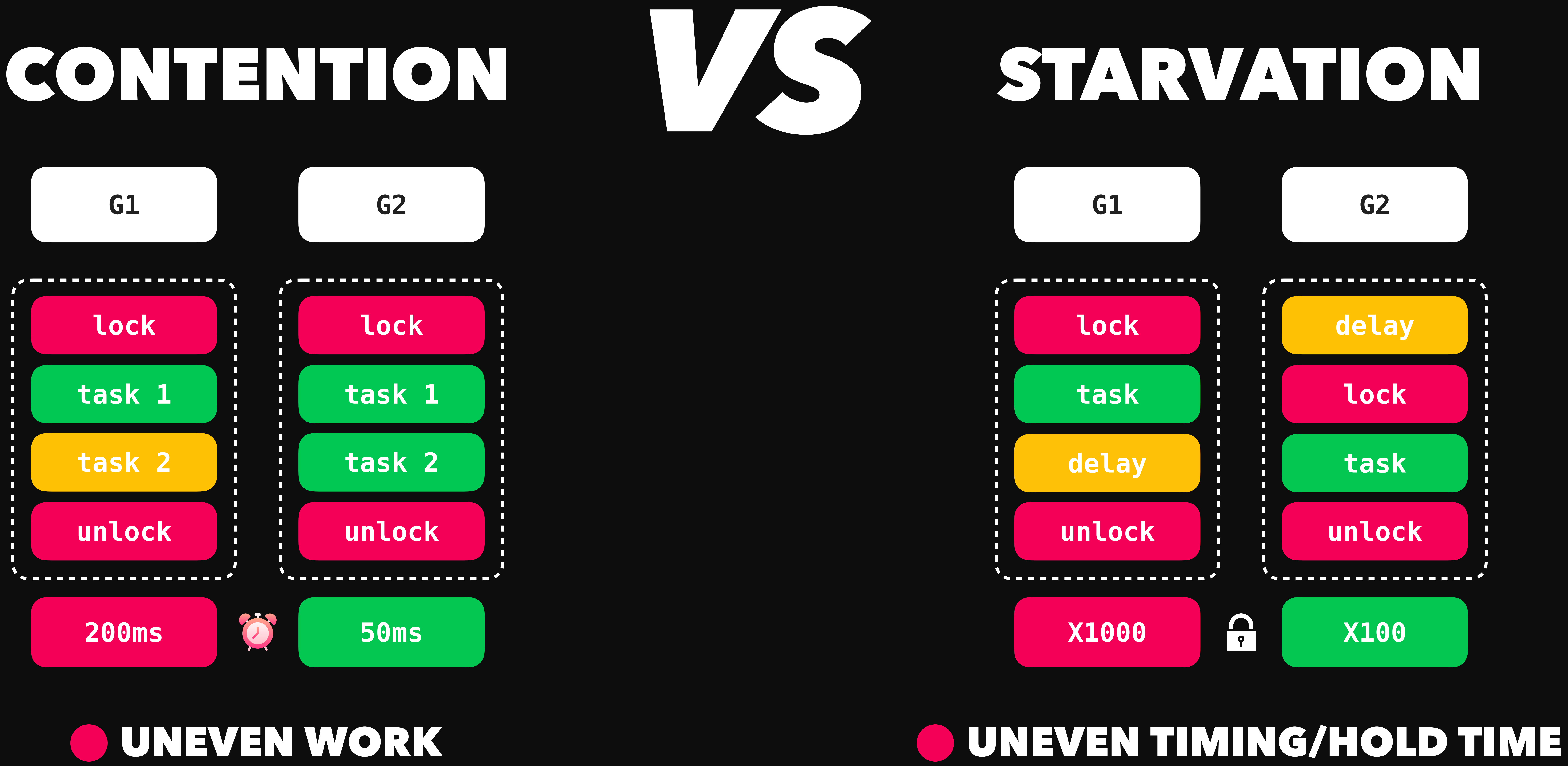

Mutex
released
G1
released
G2
G1 work
G2 work

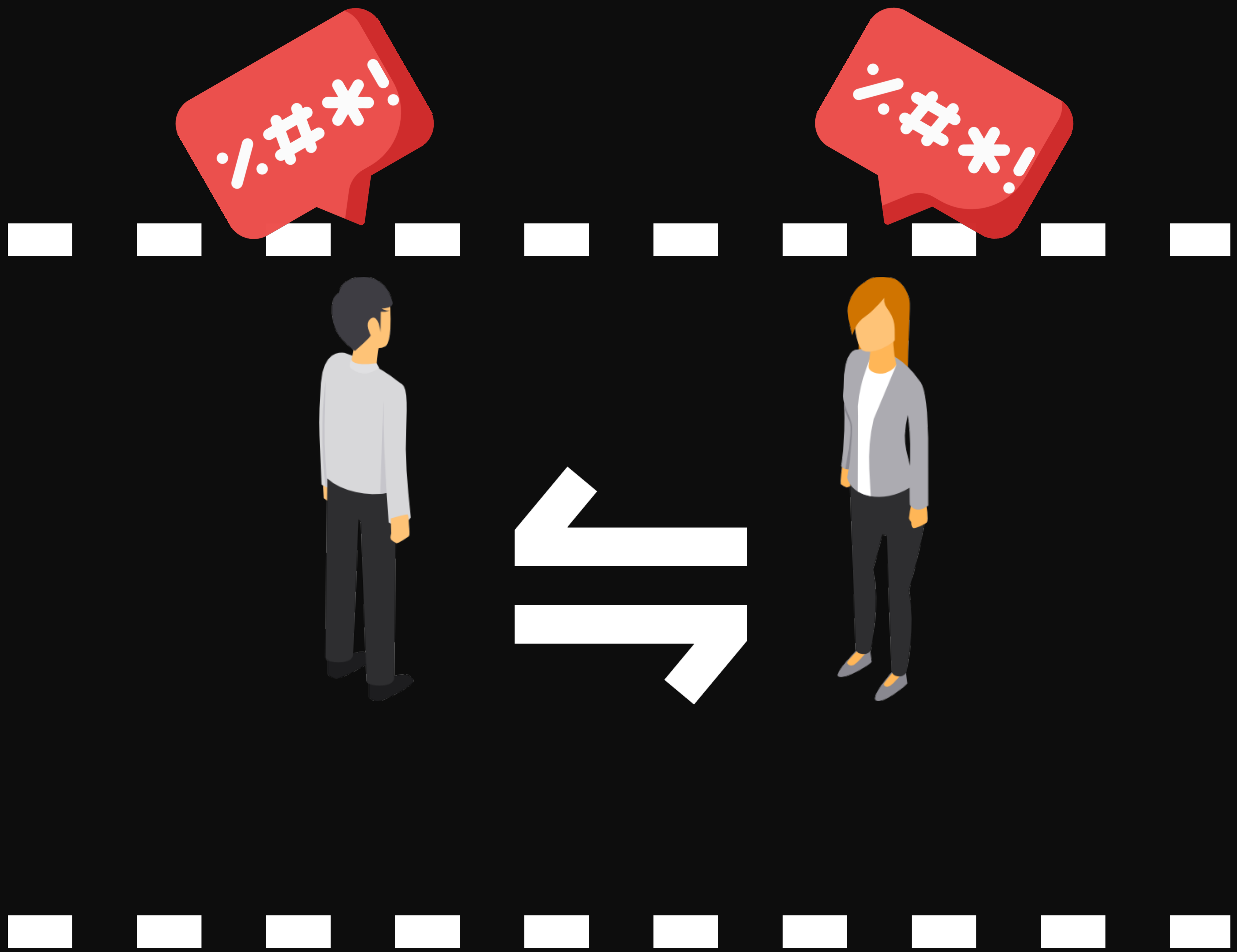
3

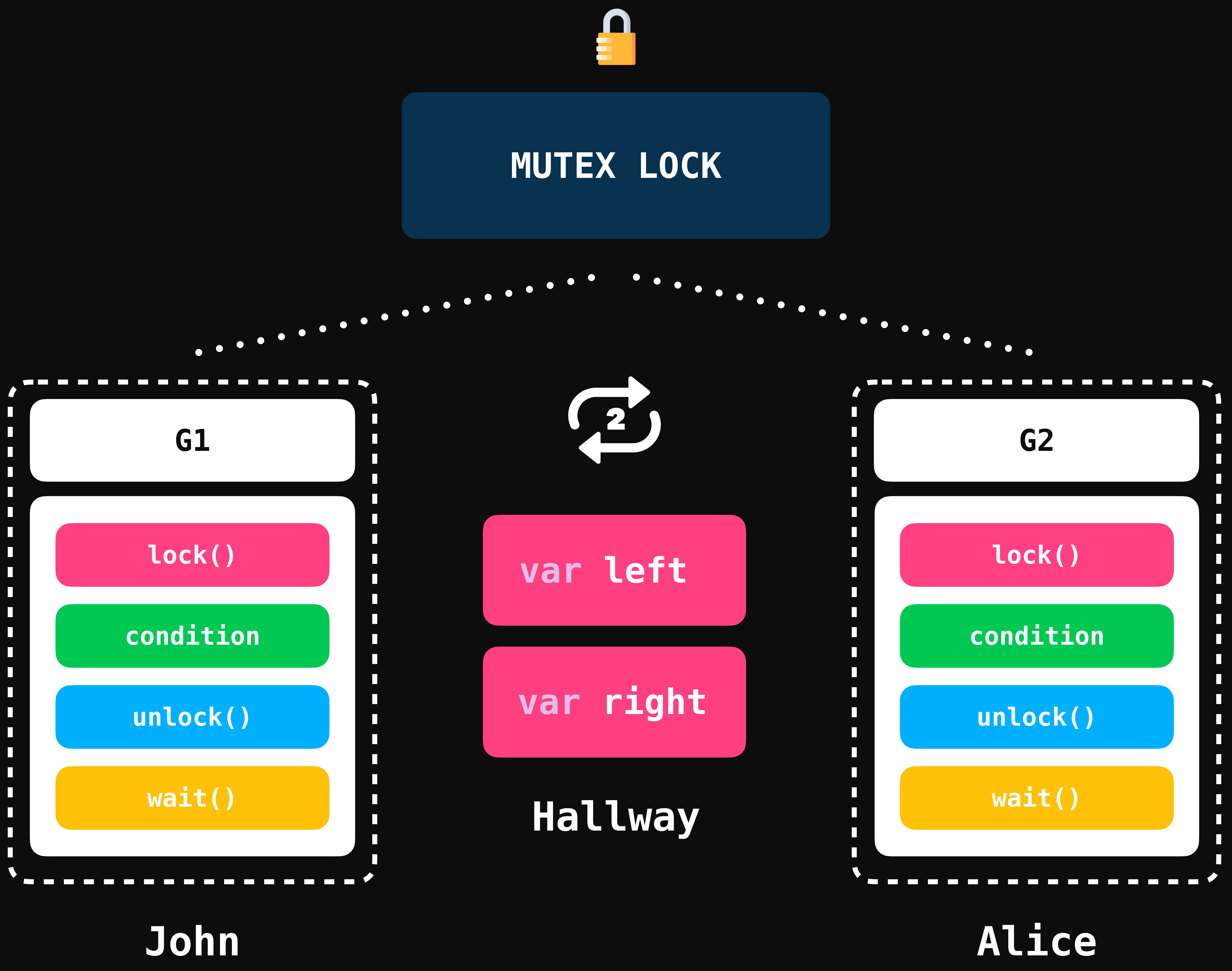
REFILL

1


MUTEX LOCK
G1
acquire
work
sleep
release
100µs
G2
sleep
acquire
work
release
100µs
Scheduler Queue
G2
10ms

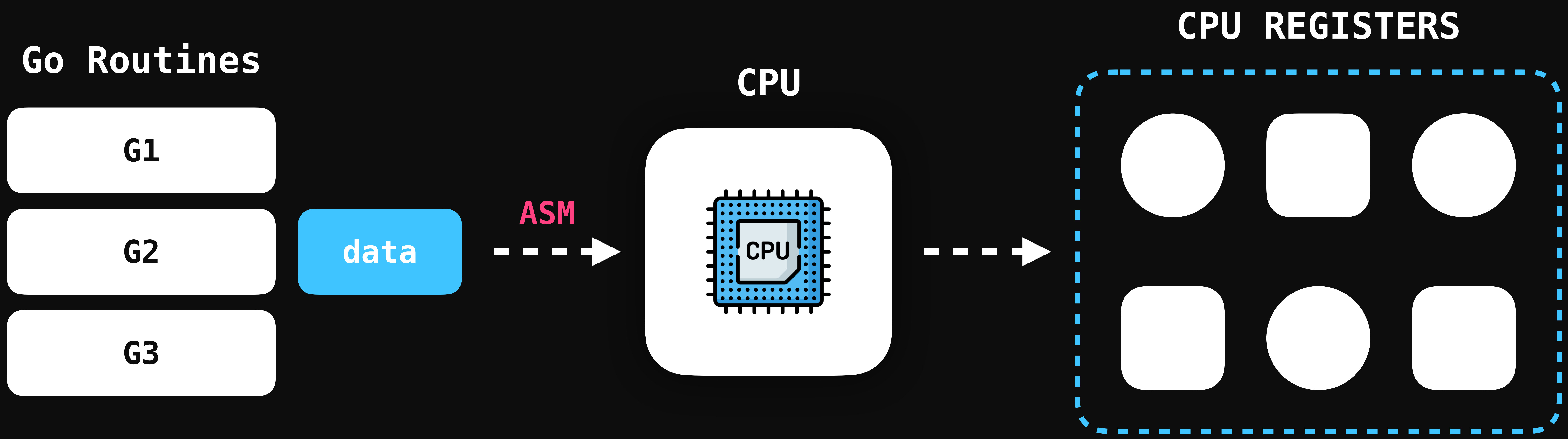

CONTENTION
VS
STARVATION
G1
G2
lock
task 1
task 2
unlock
lock
task 1
task 2
unlock
200ms
50ms
UNEVEN WORK
G1
G2
lock
task
delay
unlock
delay
lock
task
unlock
X1000
X100
UNEVEN TIMING/HOLD TIME

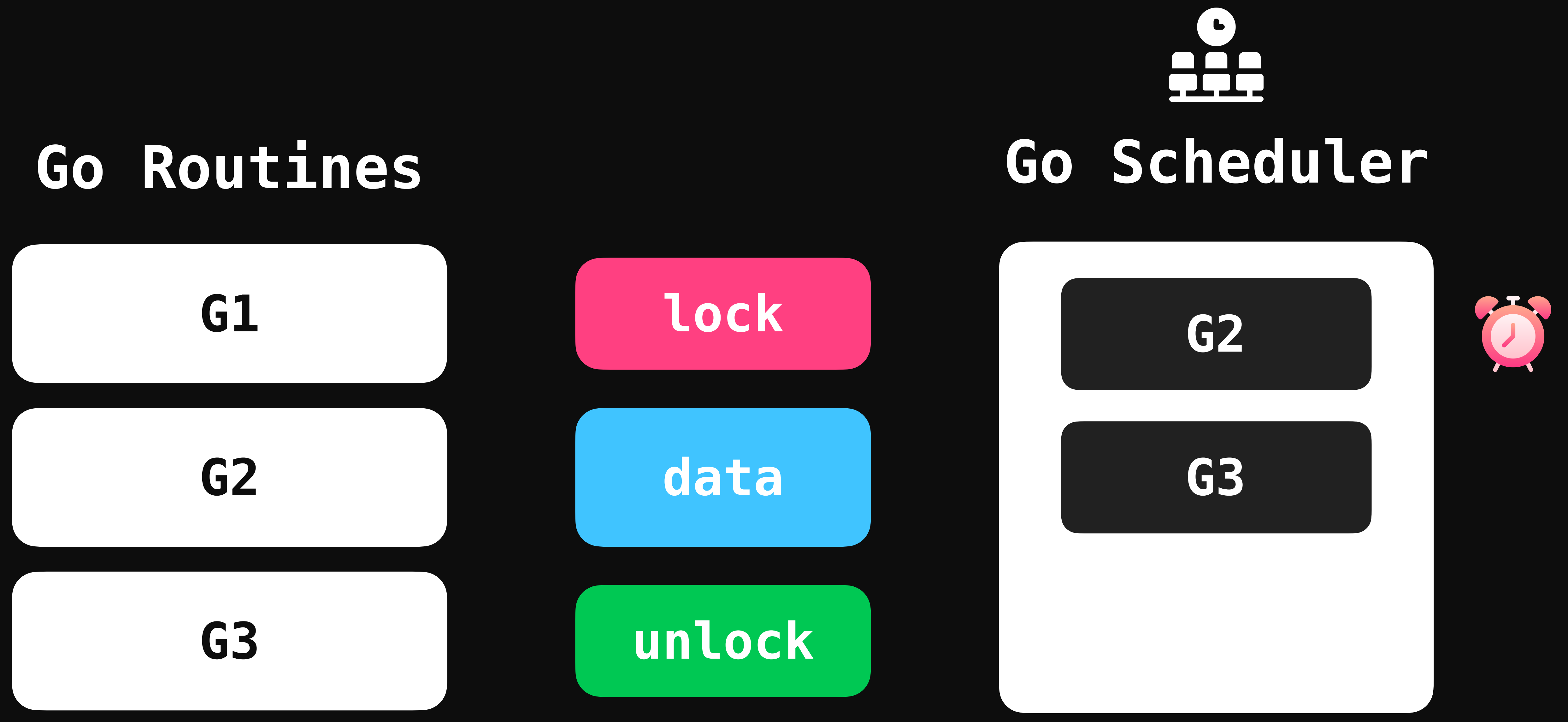



MUTEX LOCK
G1
lock()
condition
unlock()
wait()
John
2
var left
var right
Hallway
G2
lock()
condition
unlock()
wait()
Alice

sync.Locker

```go
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

```
Lock()
Unlock()
```

TEST & SET

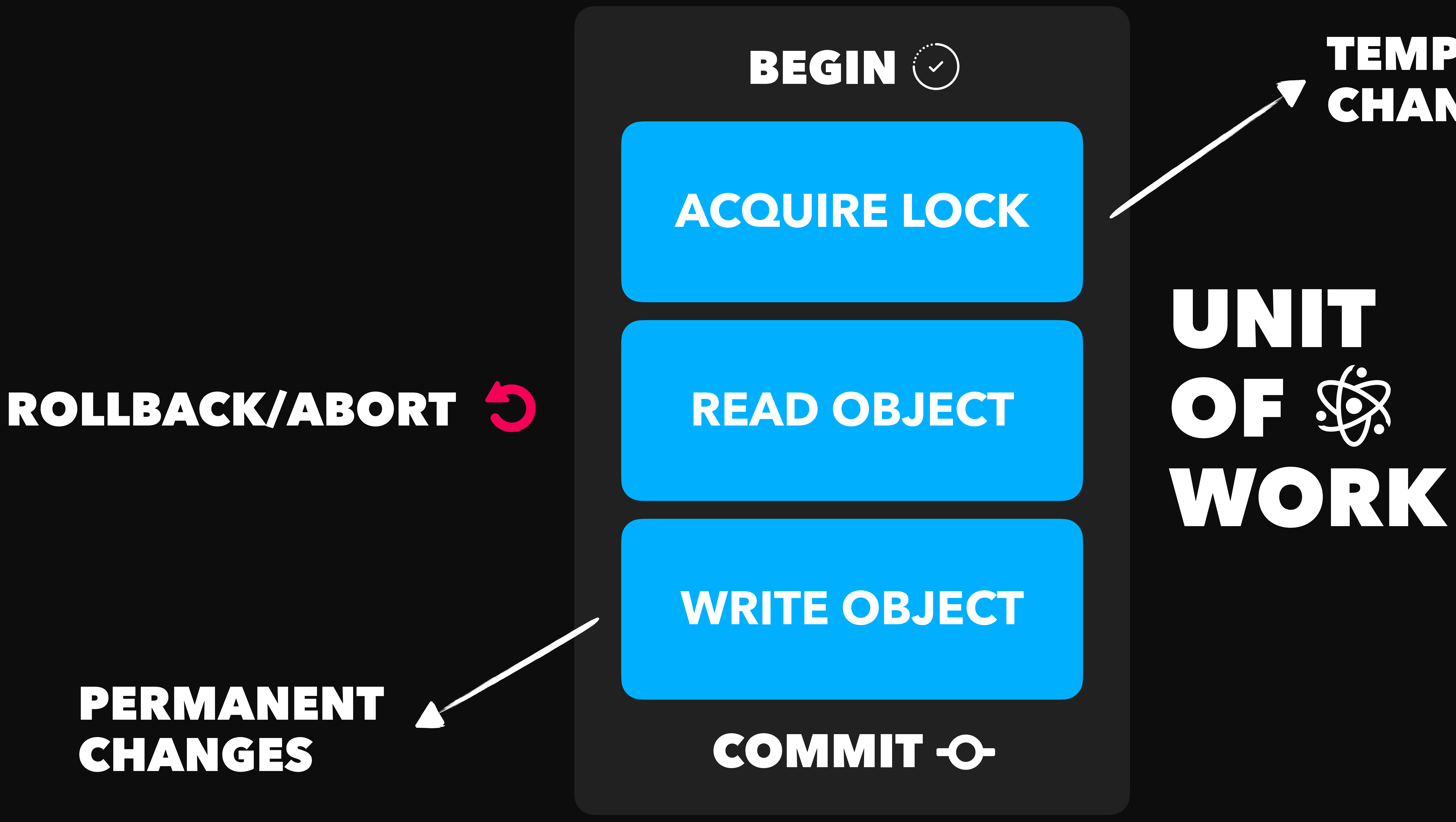
- Prefer **Atomics** over **Mutexes** for **simple data**
- Use **Mutex** for write heavy scenarios
- Use **RWMutex** for read heavy / mixed scenarios
- Prefer **Fine Grained Context** where possible (limit the context)
- **Don’t use** the mutex **longer** than you need to (**avoid extra hold time**)
- 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

Concurrency Control ensures that **correct results** for **concurrent operations** are generated, while getting those results as **quickly** as possible.

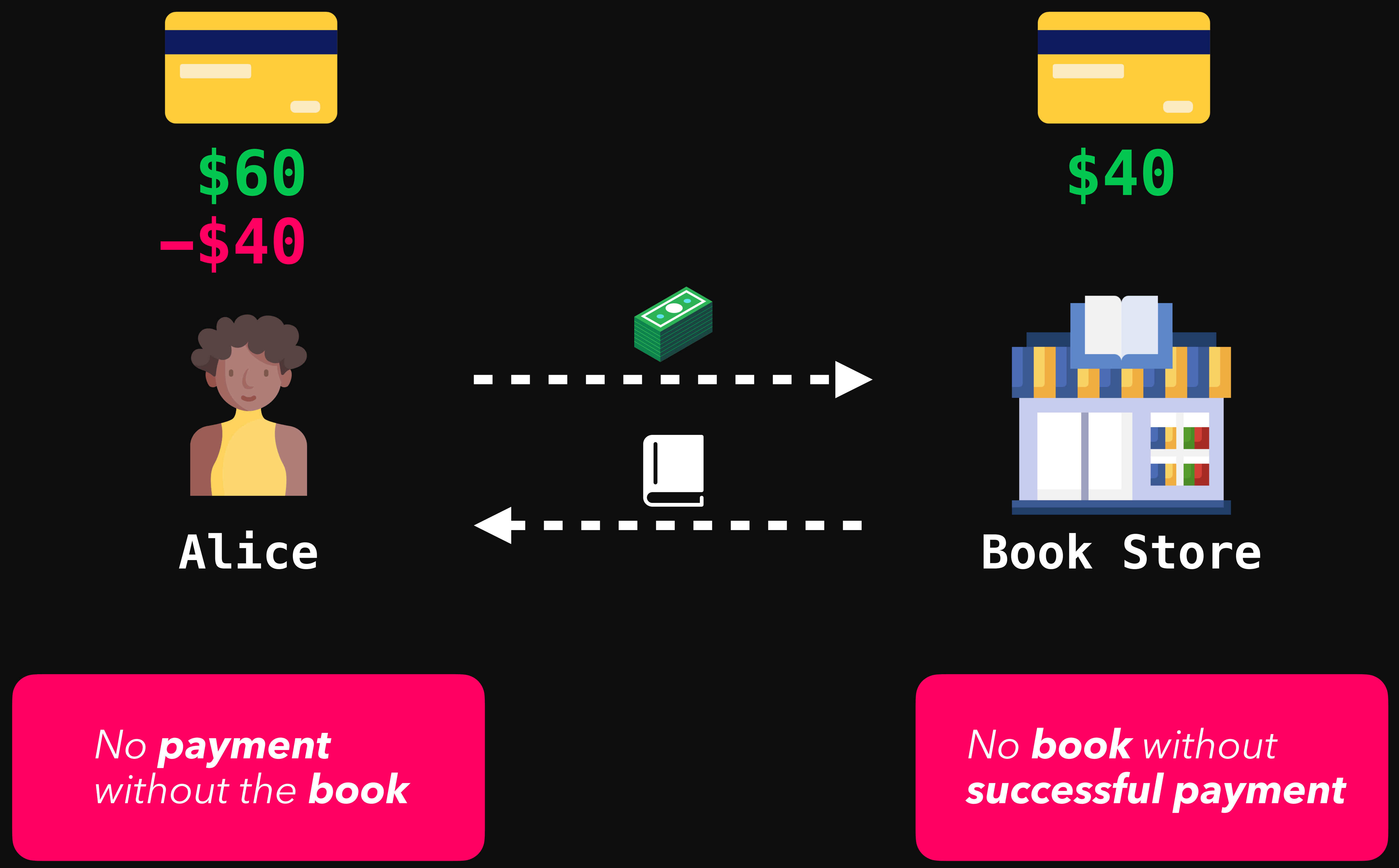
ACID

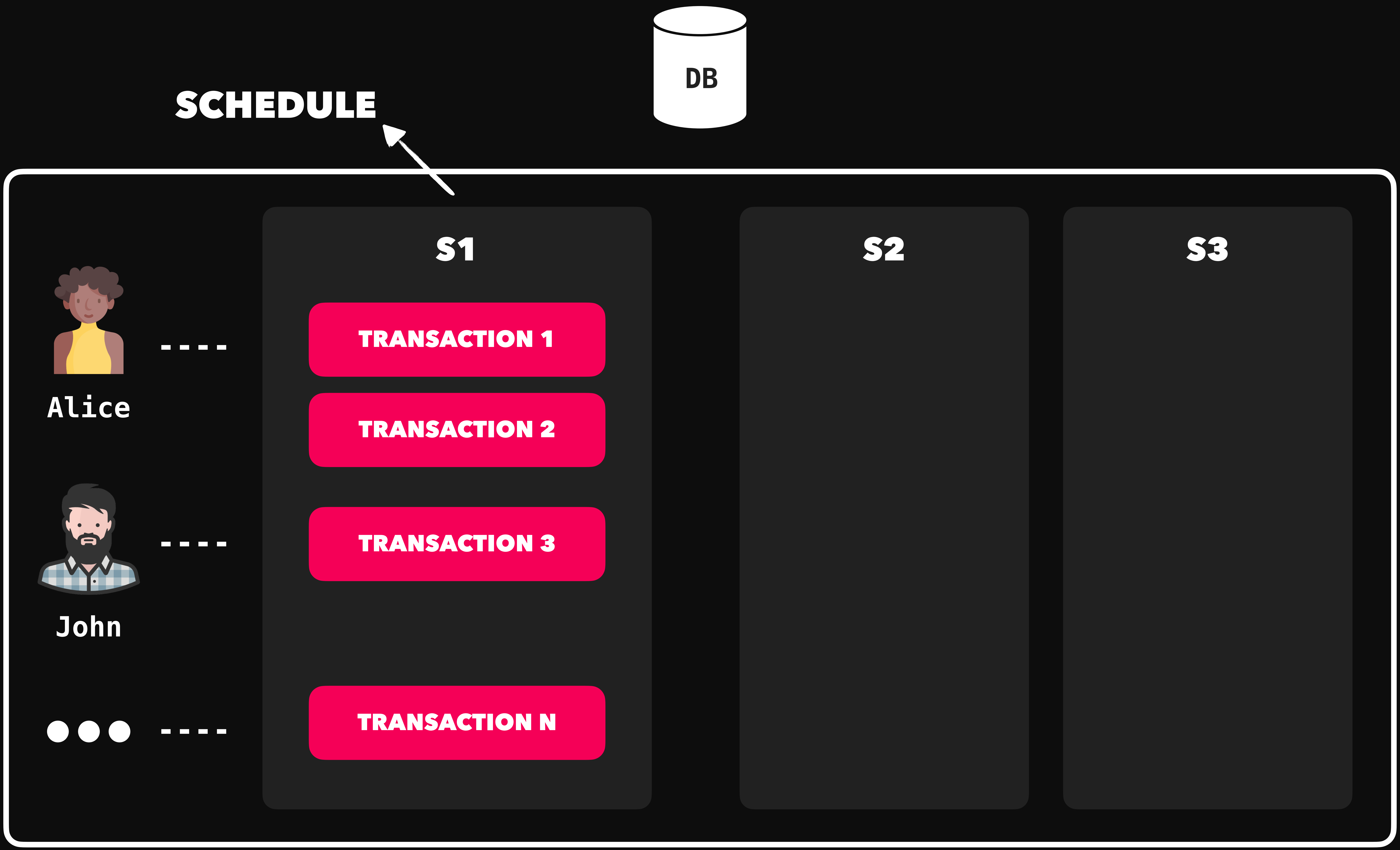
ATOMICITY	CONSISTENCY
ISOLATION	DURABILITY

TRANSACTION

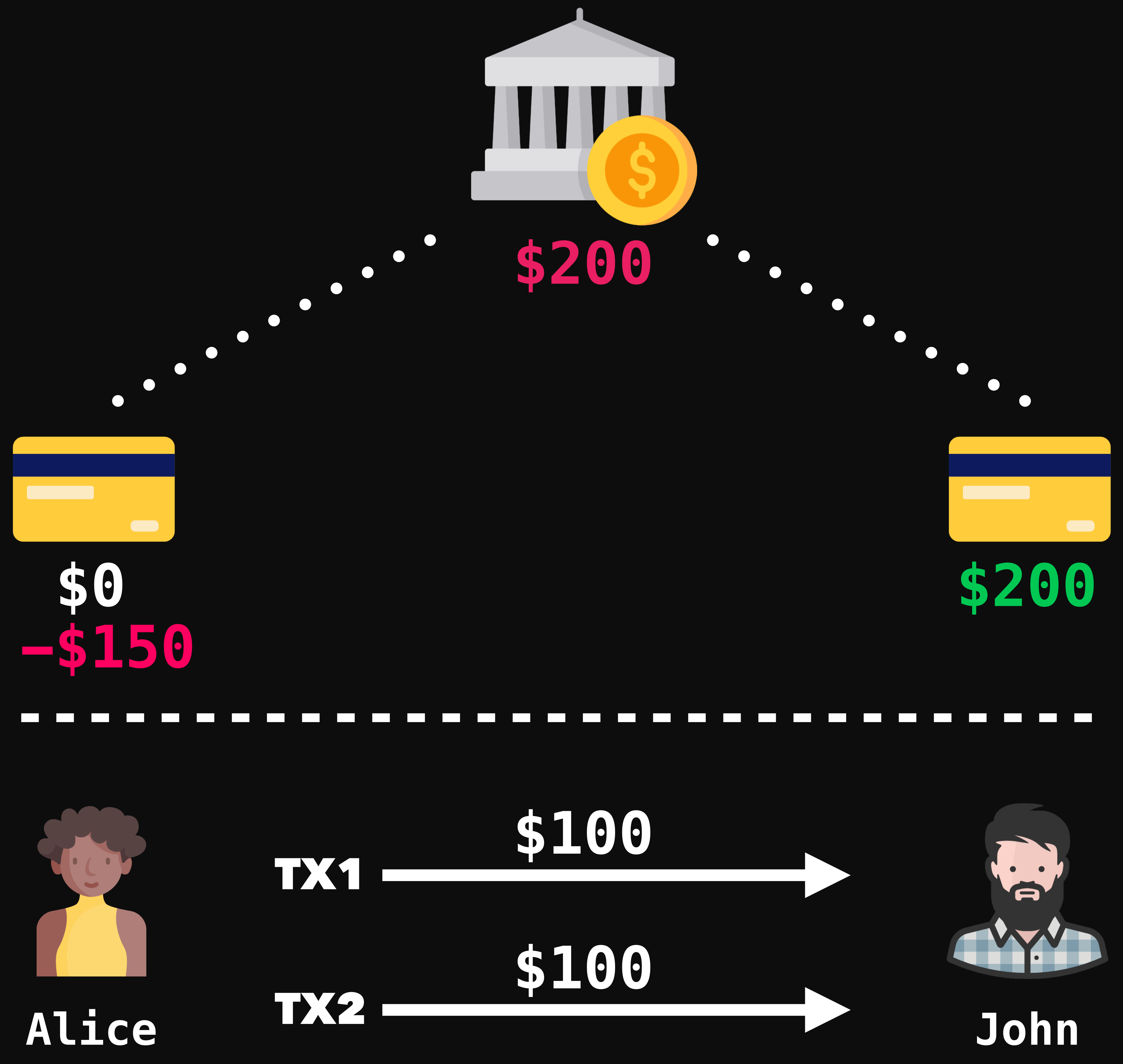


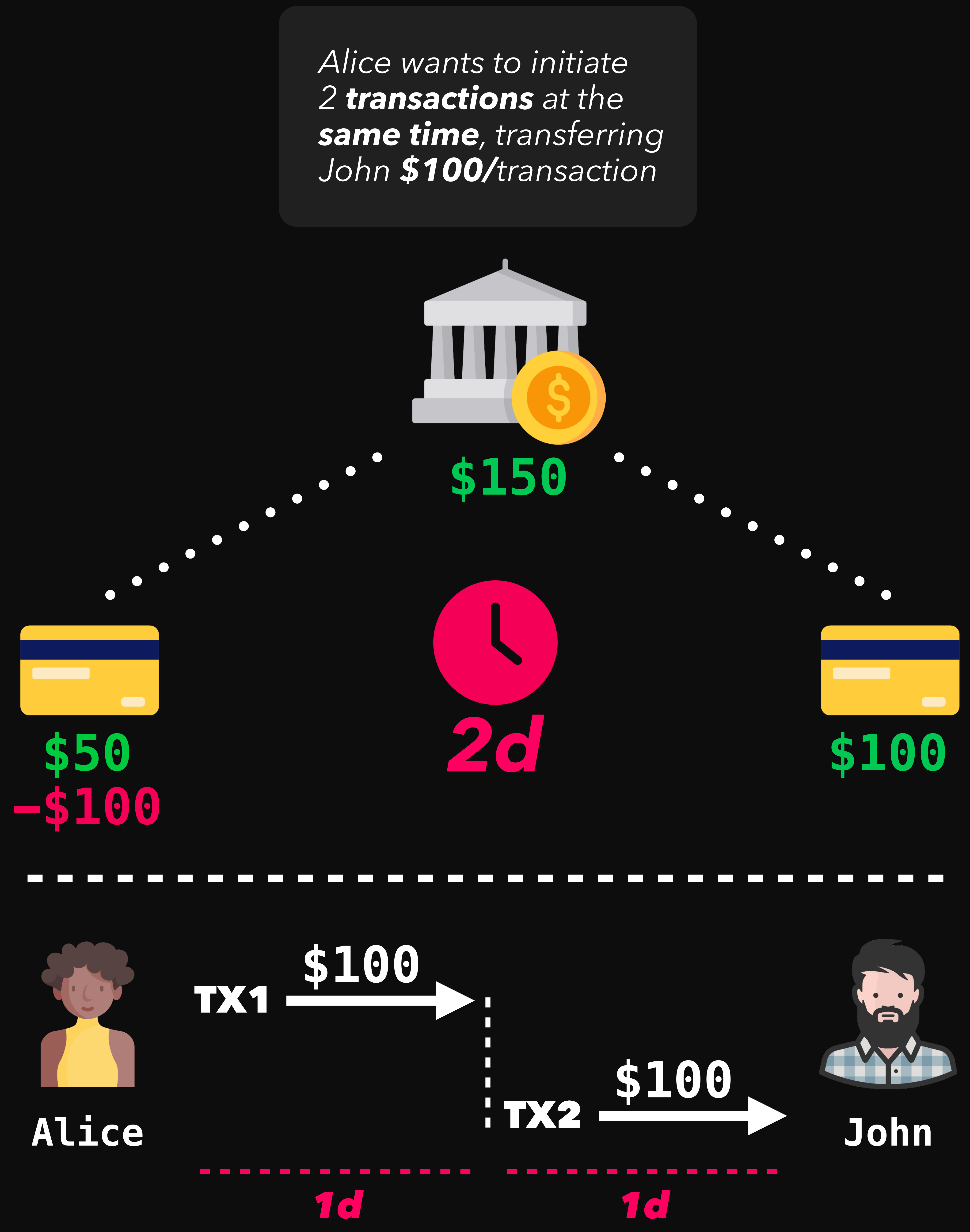
Each ***transaction MUST COMPLETE*** *or* ***FAIL*** *as a* ***UNIT****. It can't be partially complete.*


$60
-$40
$40
Alice
Book Store
No payment without the book
No book without successful payment

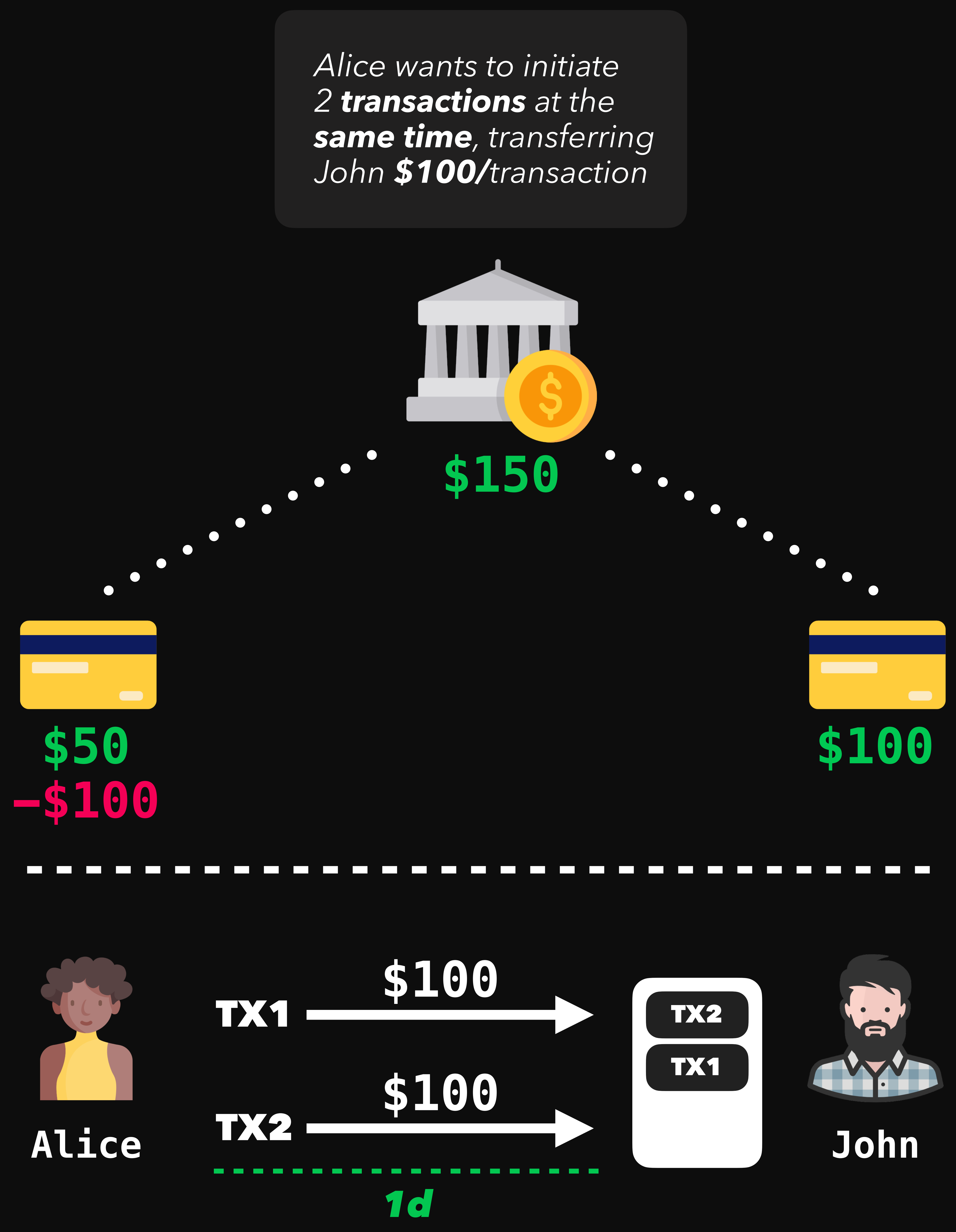

DB
SCHEDULE
S1
S2
S3
Alice
John
TRANSACTION 1
TRANSACTION 2
TRANSACTION 3
TRANSACTION N

Alice wants to initiate 2 ***transactions*** *at the* ***same time****, transferring John* ***$100/****transaction*





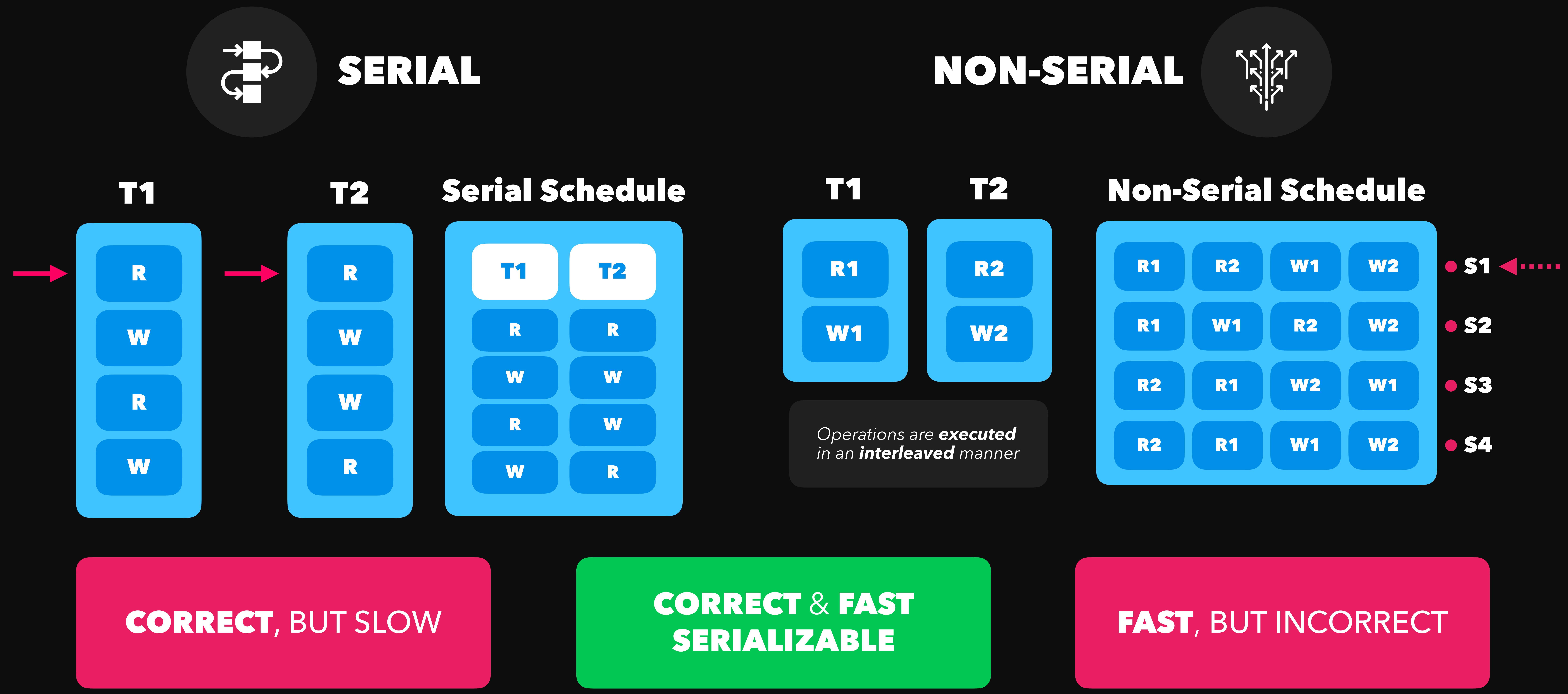
Alice wants to initiate 2 **transactions** at the **same time**, transferring John **$100/**transaction
$150
2d
$50
-$100
$100
Alice
TX1
$100
TX2
$100
John
1d
1d



Alice wants to initiate 2 **transactions** at the **same time**, transferring John **$100/**transaction
$150
$50
-$100
$100
TX1
$100
TX2
$100
TX2
TX1
Alice
John
1d

In **Concurrency Control** of **databases** and various **transactional applications**, a transaction schedule is **Serializable** if its **outcome** is **equal** to the outcome of its transactions **executed serially**, without overlapping in time.

SCHEDULING



SERIAL
NON-SERIAL
T1
R
W
R
W
T2
R
W
W
R
Serial Schedule
T1 T2
R R
W W
R W
W R
T1
R1
W1
T2
R2
W2
Operations are ***executed*** *in an* ***interleaved*** *manner*
Non-Serial Schedule
R1 R2 W1 W2
R1 W1 R2 W2
R2 R1 W2 W1
R2 R1 W1 W2
S1
S2
S3
S4
CORRECT, BUT SLOW
CORRECT & **FAST**
SERIALIZABLE
FAST, BUT INCORRECT

SERIALIZABILITY

RESULT EQUIVALENT

CONFLICT SERIALIZABLE

VIEW SERIALIZABLE

RESULT EQUIVALENT

X: 2
Y: 5

Initial values

S1

T1	T2
R(X)	
X=X+5	
W(X)	
R(Y)	
Y=Y+5	
W(Y)	
	R(X)
	X=X*3
	W(X)

X: 21 Y: 10

⇔

S2

T1	T2
R(X)	
X=X+5	
W(X)	
	R(X)
	X=X*3
	W(X)
R(Y)	
Y=Y+5	
W(Y)	

X: 21 Y: 10

Initial values

X: 3
Y: 6

S1

T1	T2
R(X)	
X=X+1	
W(X)	
R(Y)	
Y=Y*2	
W(Y)	
	R(X)
	X=X*3
	W(X)

X: 12 Y: 12

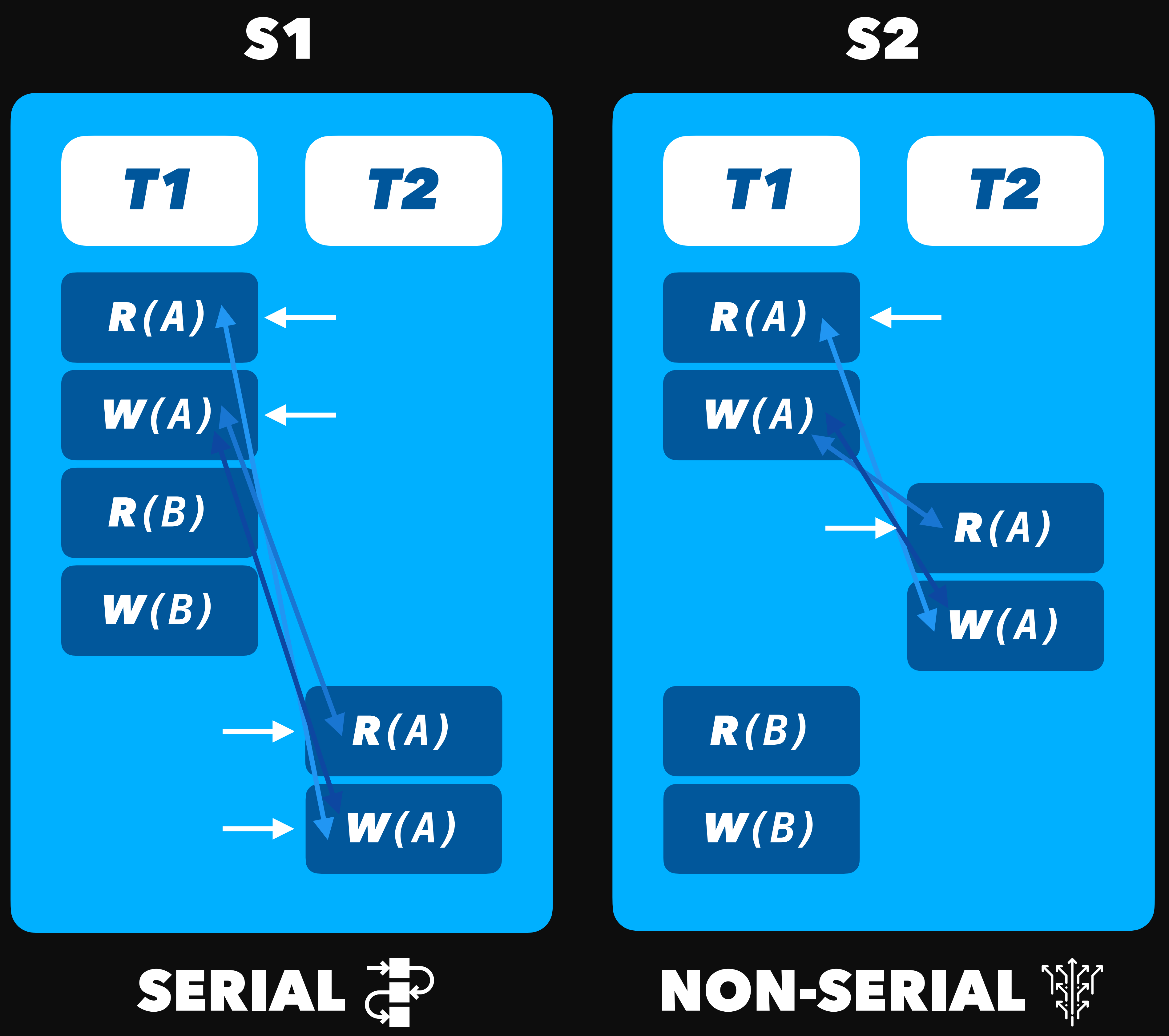
⇔

S2

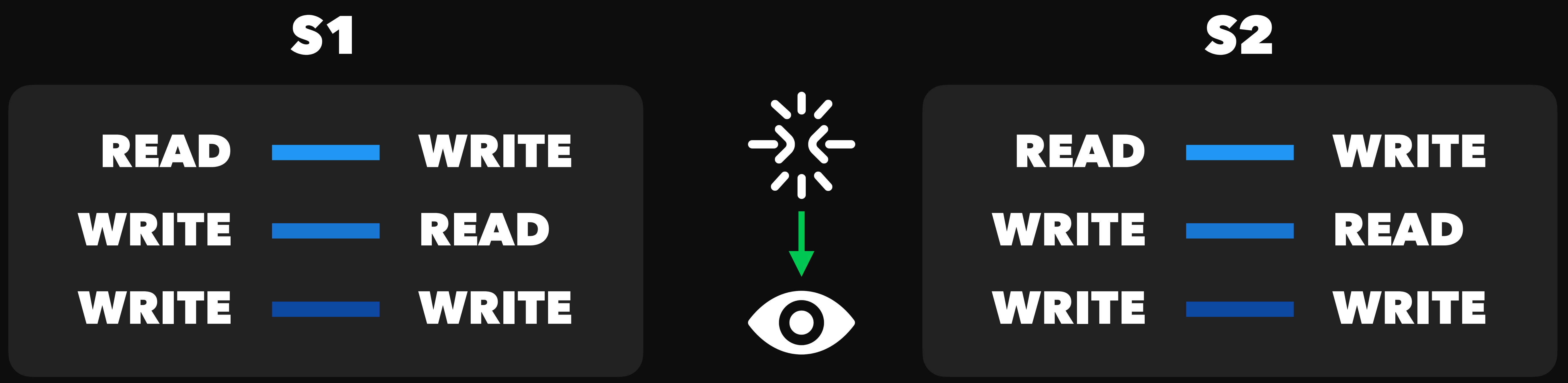
T1	T2
R(X)	
X=X+1	
W(X)	
	R(Y)
	X=Y*2
	W(Y)

X: 4 Y: 12

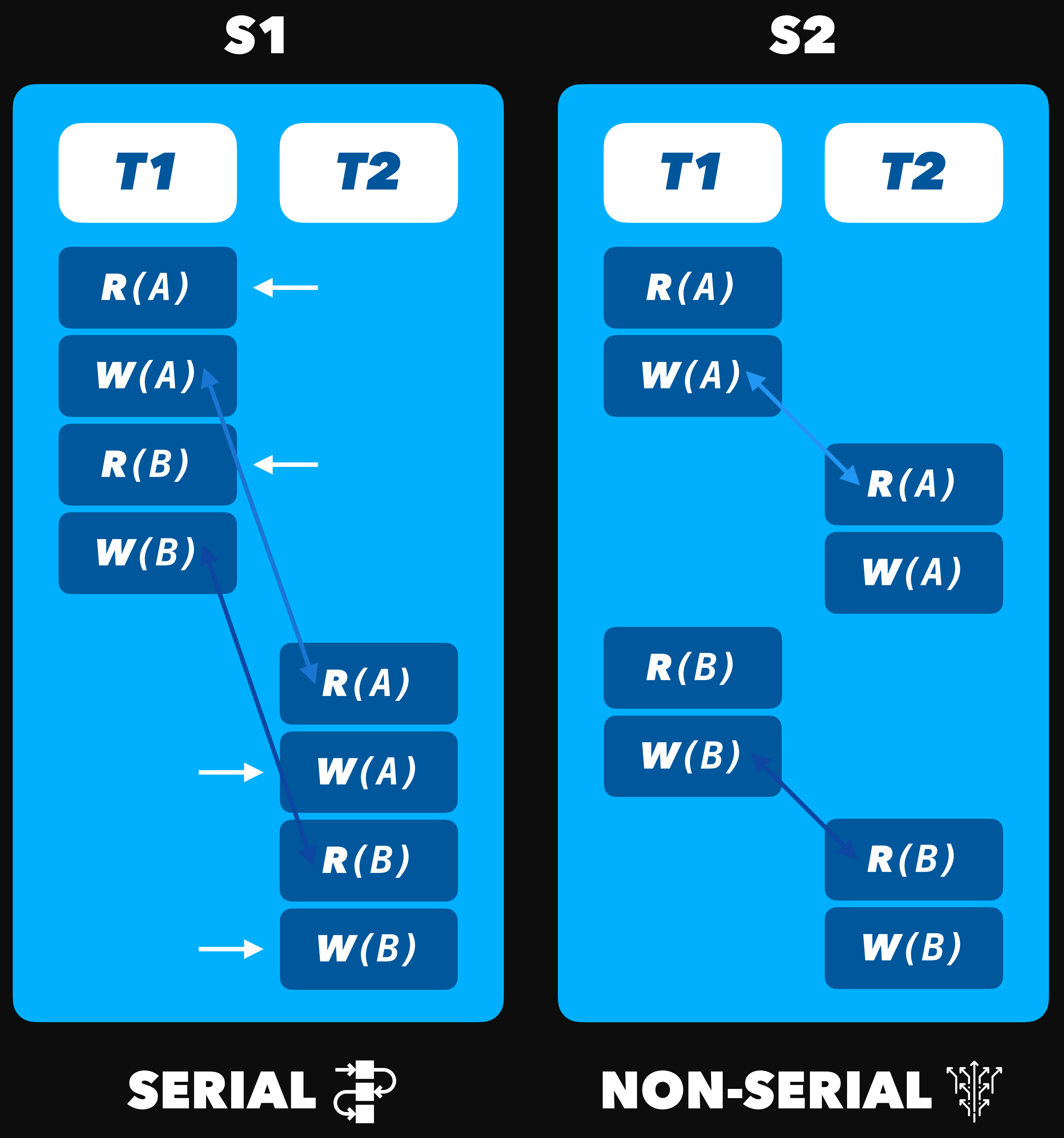
CONFLICT SERIALIZABLE



- **Conflict operations** are on the **same data item** i.e **A**, **B**
- **RW**, **WR**, **WW** conflicts MUST be on **different** transactions
- At least **1** of conflict operations MUST be a **Write**
- **2 READ** operations will **not** create a **conflict**



VIEW SERIALIZABLE



*FIRST **READ** performed by the **same transaction***

Initial operations on each **data item** are on the same transaction

Final operations on each **data item** are on the same transaction

*LAST **WRITE** performed by the **same transaction***

Number of **WR** operations MUST be the **same** on all schedules

*Maintain **Producer-Consumer/W→R** sequence*

S1

- ***Initial operation** of data item **A**: **T1***
- ***Initial operation** of data item **B**: **T1***
- ***Final operation** of data item **A**: **T2***
- ***Final operation** of data item **B**: **T2***
- *Number of **RW operations**: **2***

S2

- ***Initial operation** of data item **A**: **T1***
- ***Initial operation** of data item **B**: **T1***
- ***Final operation** of data item **A**: **T2***
- ***Final operation** of data item **B**: **T2***
- *Number of **RW operations**: **2***

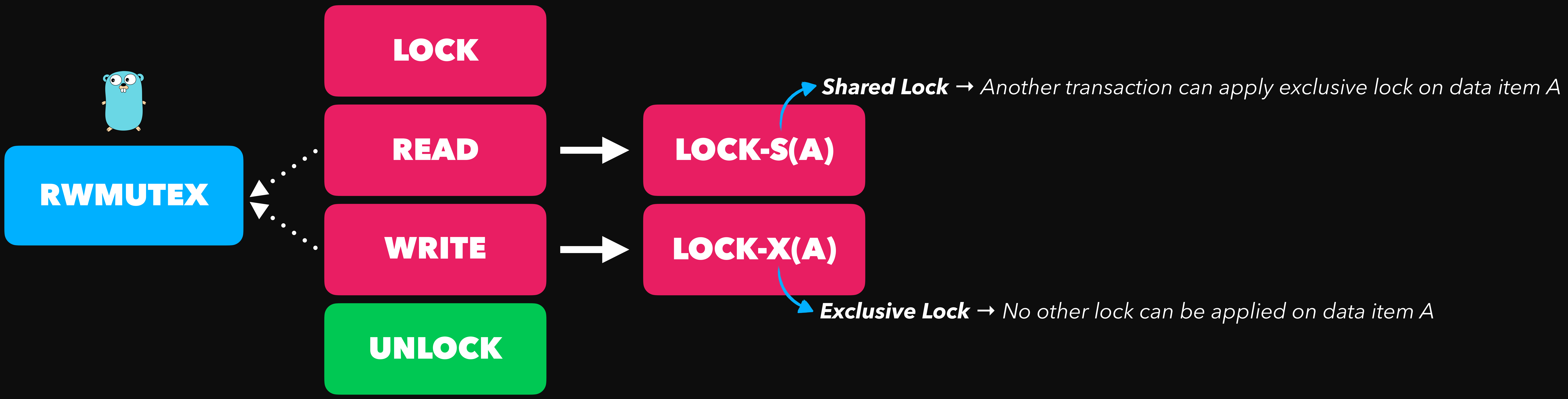
SERIALIZABILITY

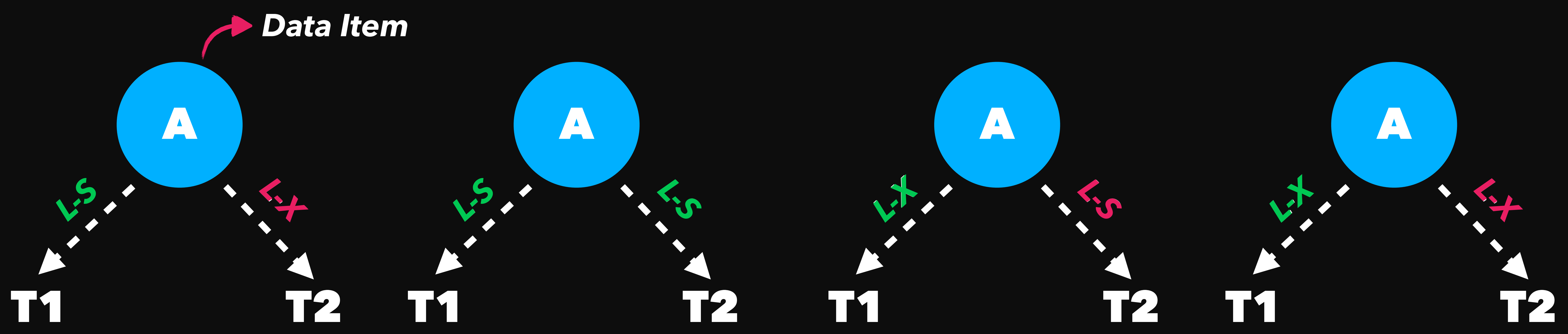
- LOCKING
- SERIALIZATION – GRAPH CHECKING
- TIMESTAMP ORDERING
- COMMITMENT ORDERING
- MULTIVERSION CONCURRENCY CONROL
- INDEX CONCURRENCY CONROL
- PRIVATE WORKSPACE MODEL

LOCKING PROTOCOLS

- SIMPLE LOCKING
- BASIC 2PL
- CONSERVATIVE 2PL
- STRICT 2PL
- RIGOROUS 2PL

SIMPLE LOCKING





TYPE	READ LOCK	WRITE LOCK
READ LOCK	✔	X
WRITE LOCK	X	X

SIMPLE LOCKING EXAMPLE

1 **A=A-100** → 2 **B=B+100** → 3 Display **A+B**

T1: **A** $\xrightarrow{100}$ **B**

- Lock-X(A)
- Read(A)
- A=A-100
- Write(A)
- Unlock(A)
- Lock-X(B)
- Read(B)
- B=B+100
- Write(B)
- Unlock(B)

A: **100**

B: **100**

A: 0

B: 100

T2: Display **A+B**

- Lock-S(A)
- Read(A)
- Unlock(A)
- Lock-S(B)
- Read(B)
- Unlock(B)
- Display(A+B)

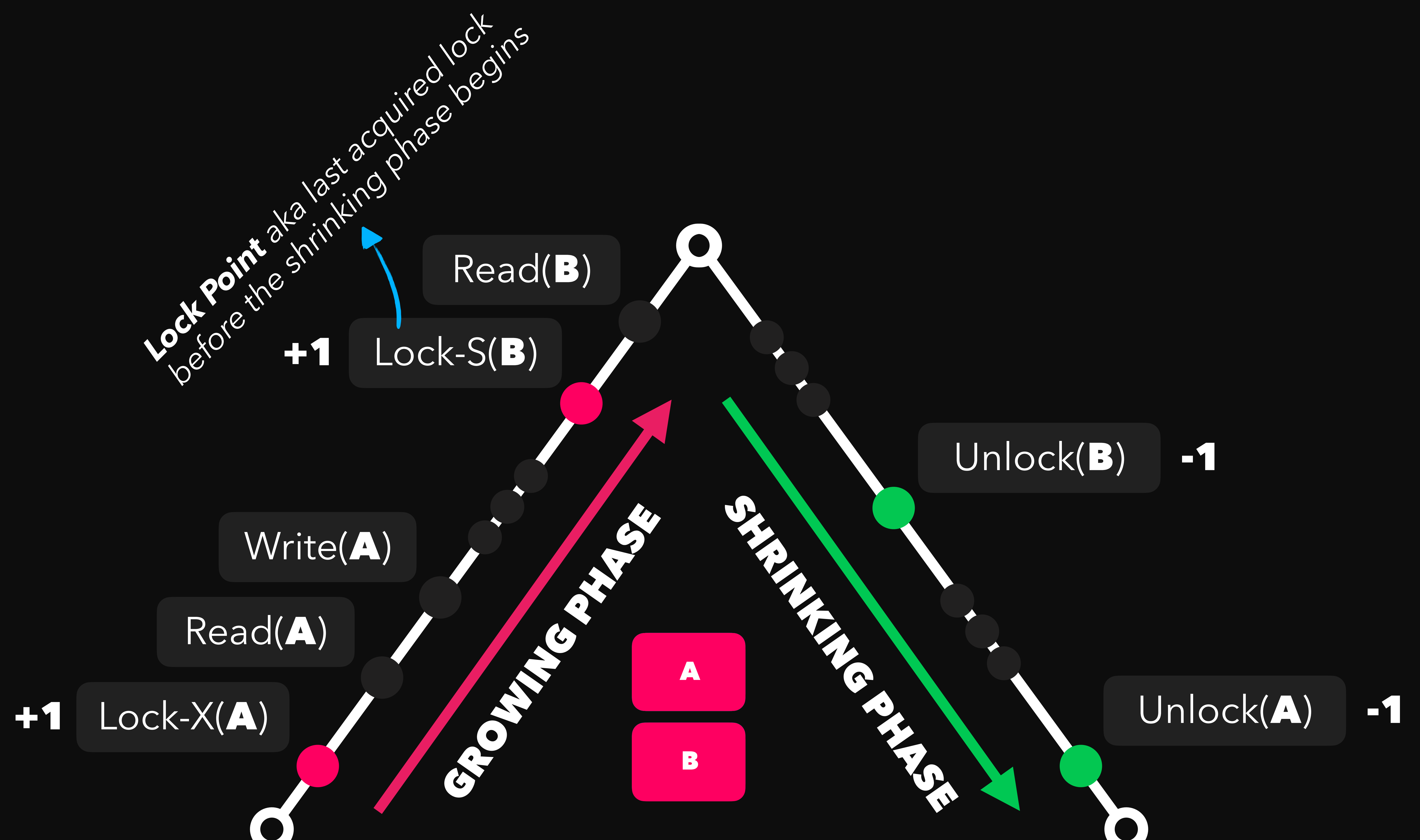
A: **100**

B: **100**

→ Display(**0+100**)

→ Display(**100+100**)

BASIC 2PL

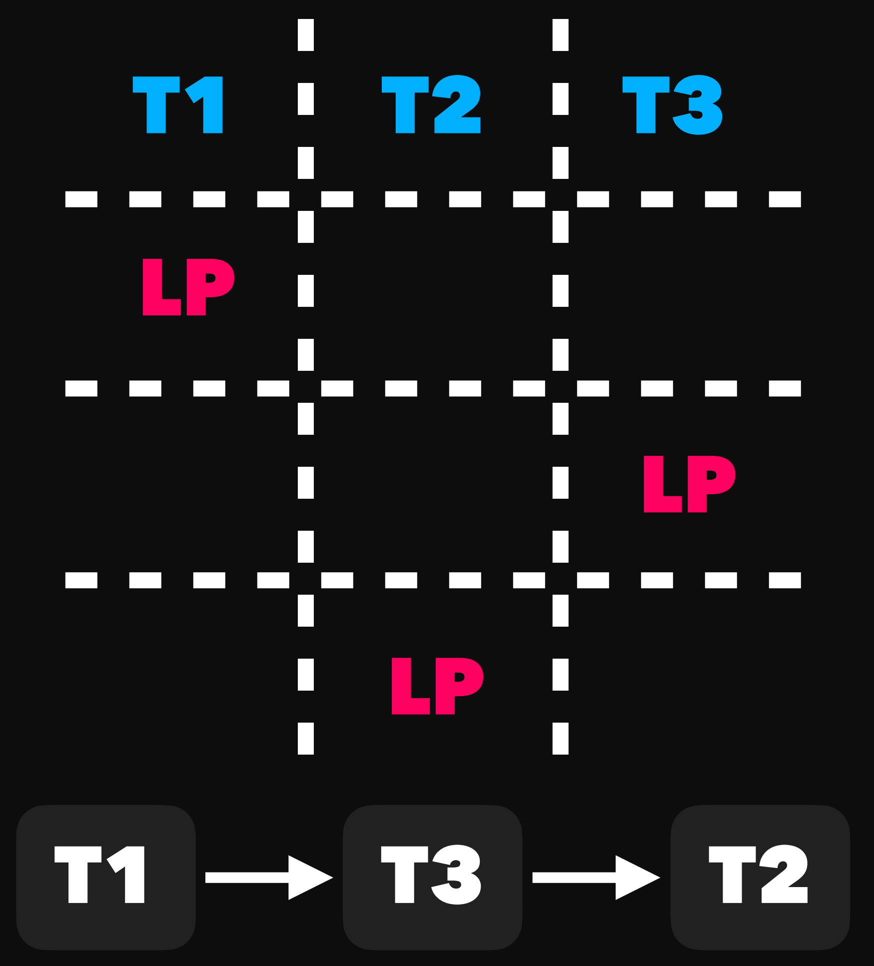


START TRANSACTION

*Locks are **acquired** and no locks are released*

END TRANSACTION

*Locks are **released** and no locks are acquired*



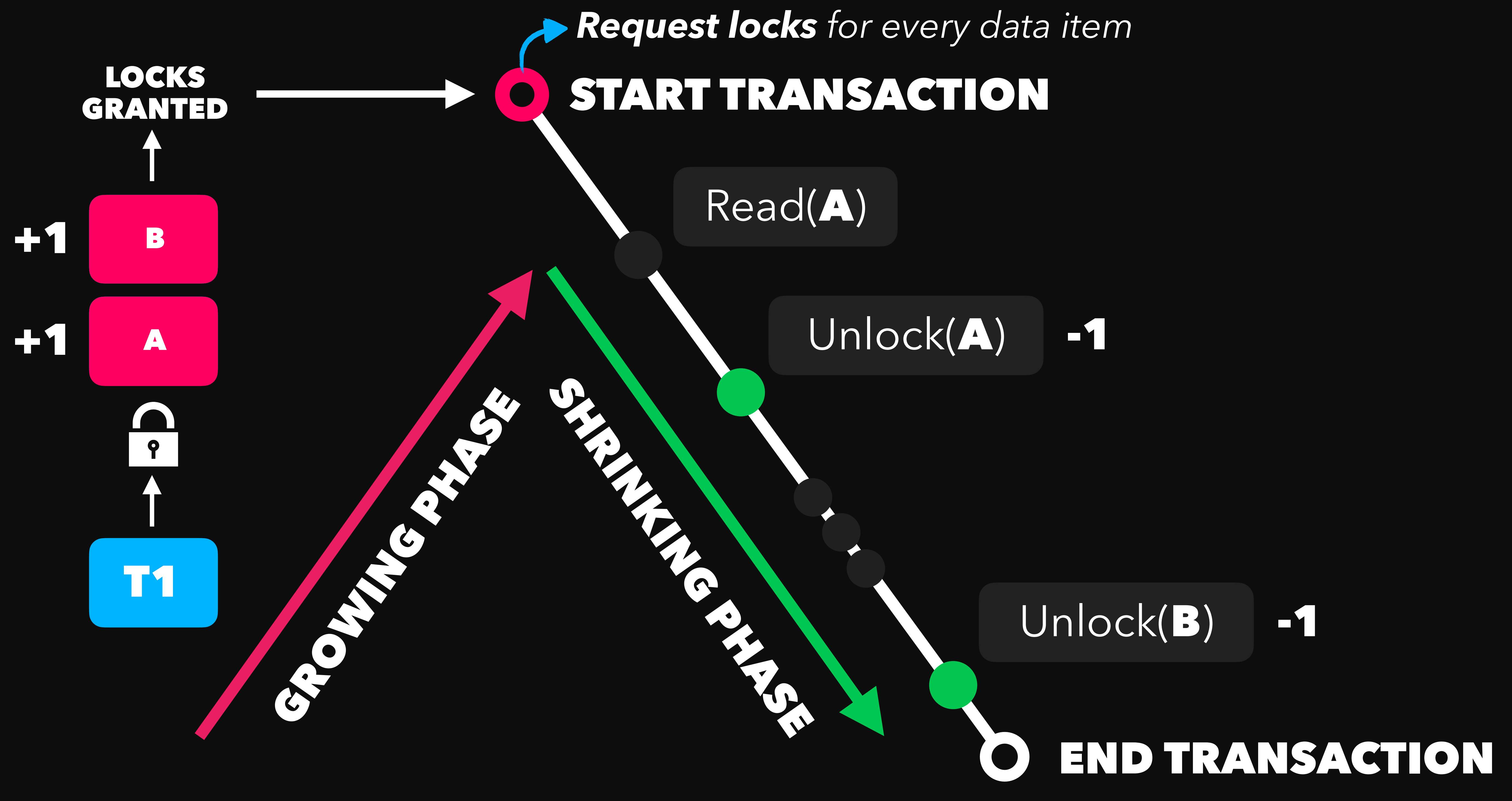
SERIALIZABLE

UNNECESSARY WAIT

DEADLOCKS

CASCADING ROLLBACKS

C2PL PROTOCOL



NO DEADLOCKS

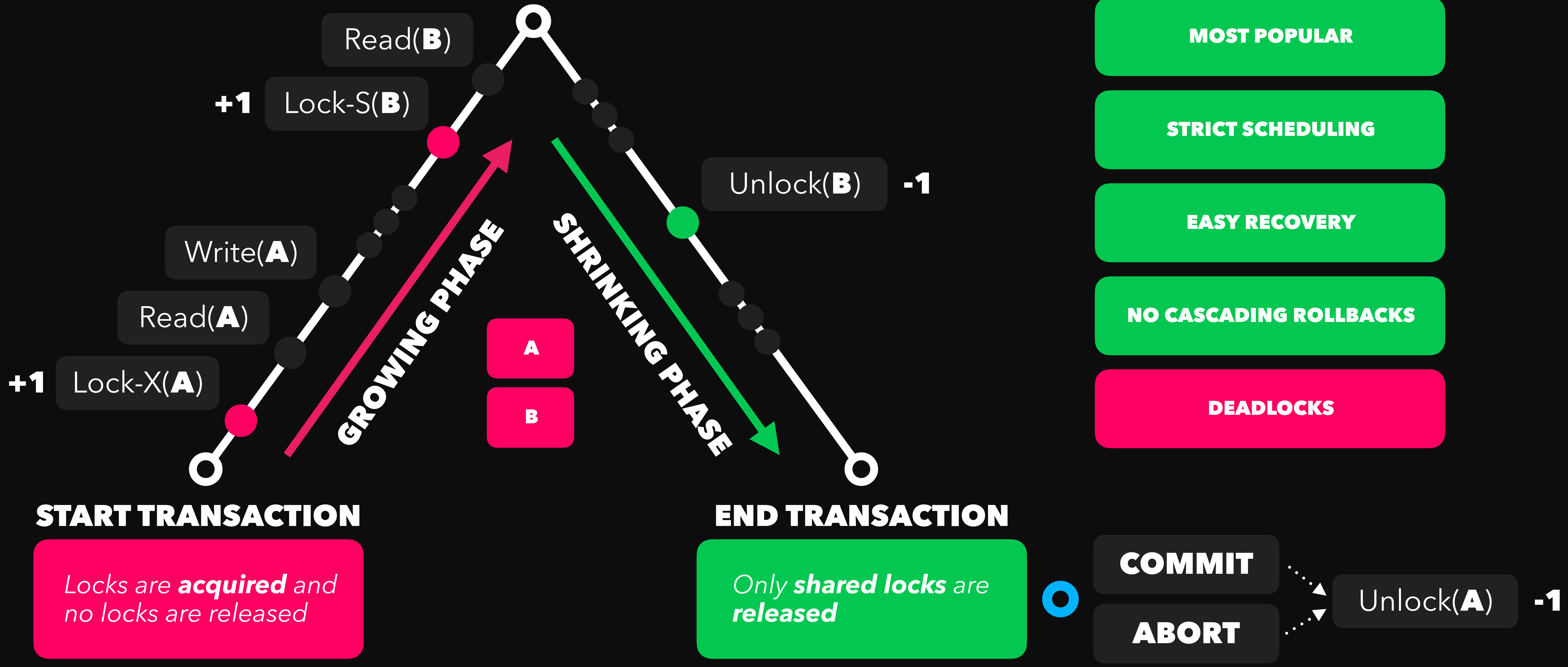
DIFICULT IMPLEMENTATION

CASCADING ROLLBACKS

All locks *are* ***acquired before*** *transaction* ***start***

Locks are ***released*** *and no locks are acquired*

S2PL PROTOCOL



MOST POPULAR

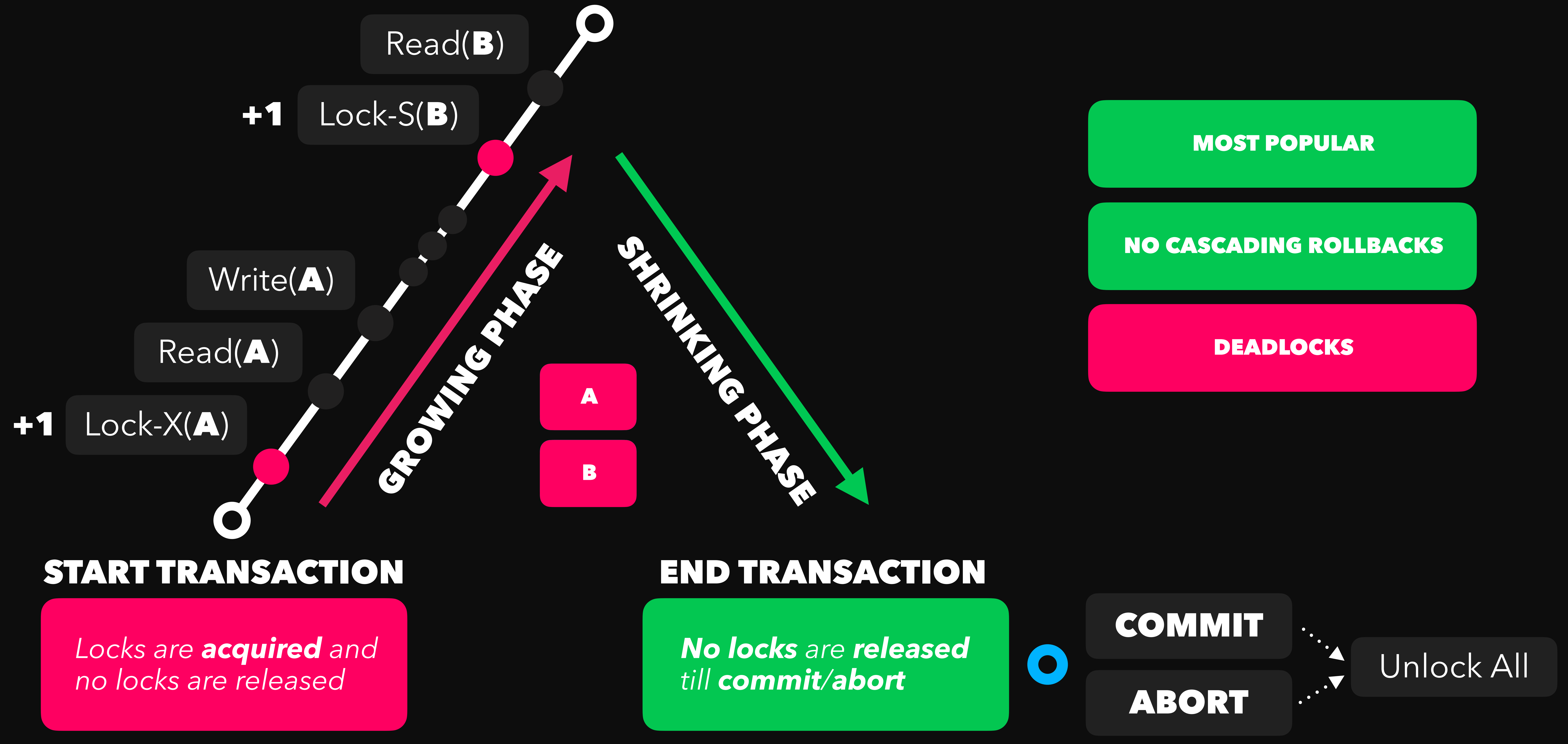
STRICT SCHEDULING

EASY RECOVERY

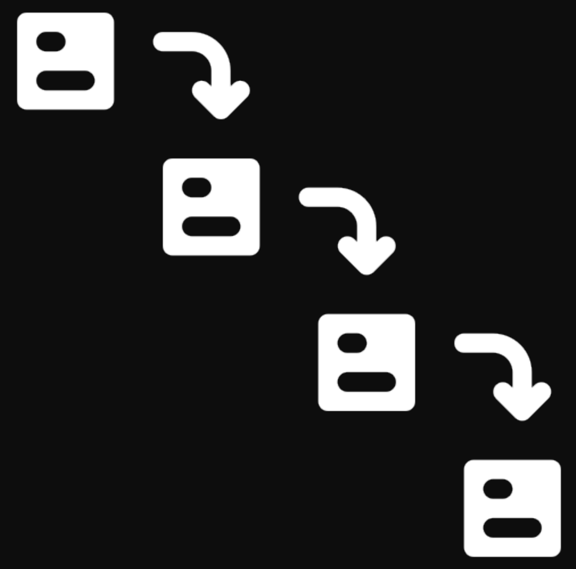
NO CASCADING ROLLBACKS

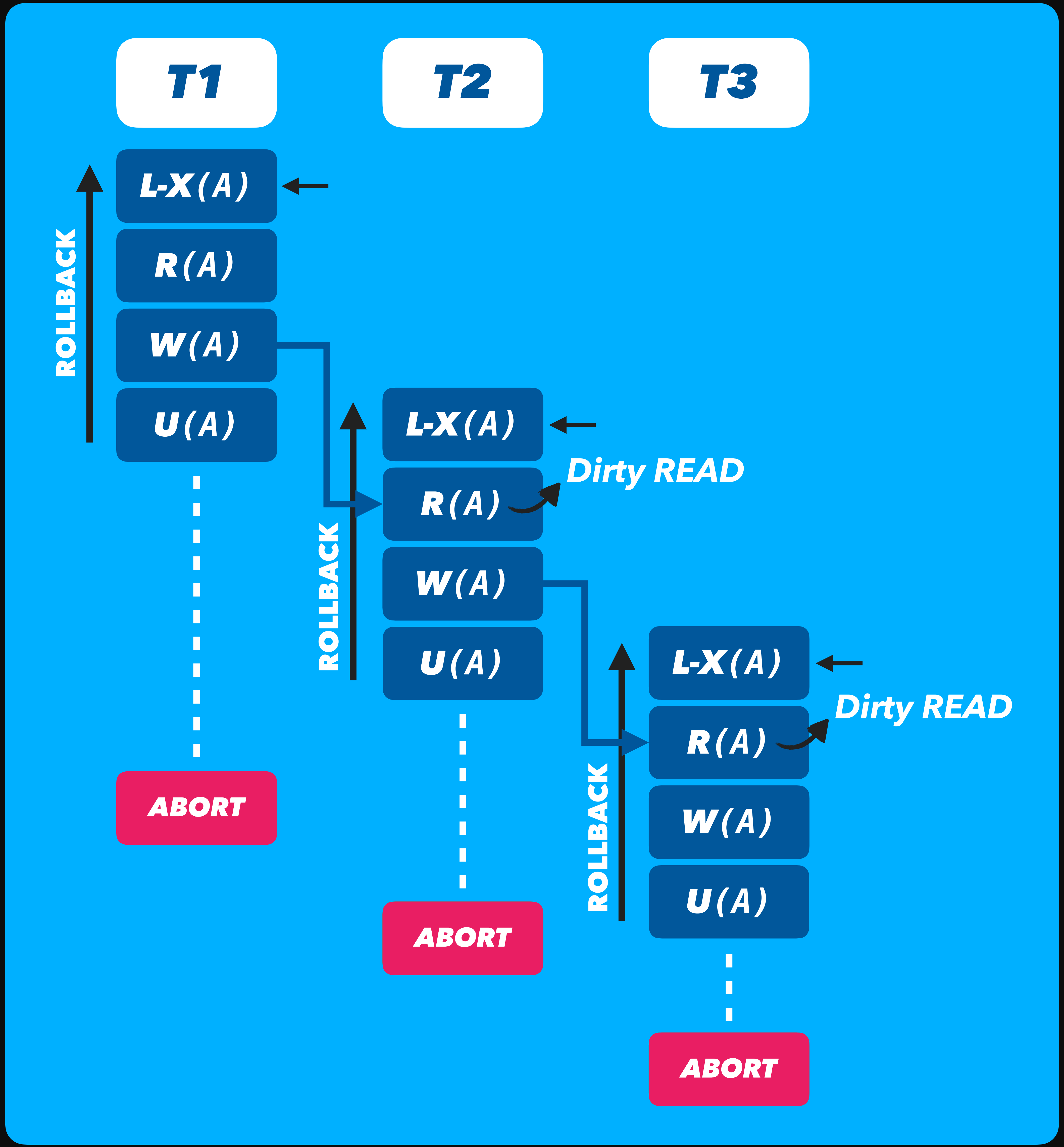
DEADLOCKS

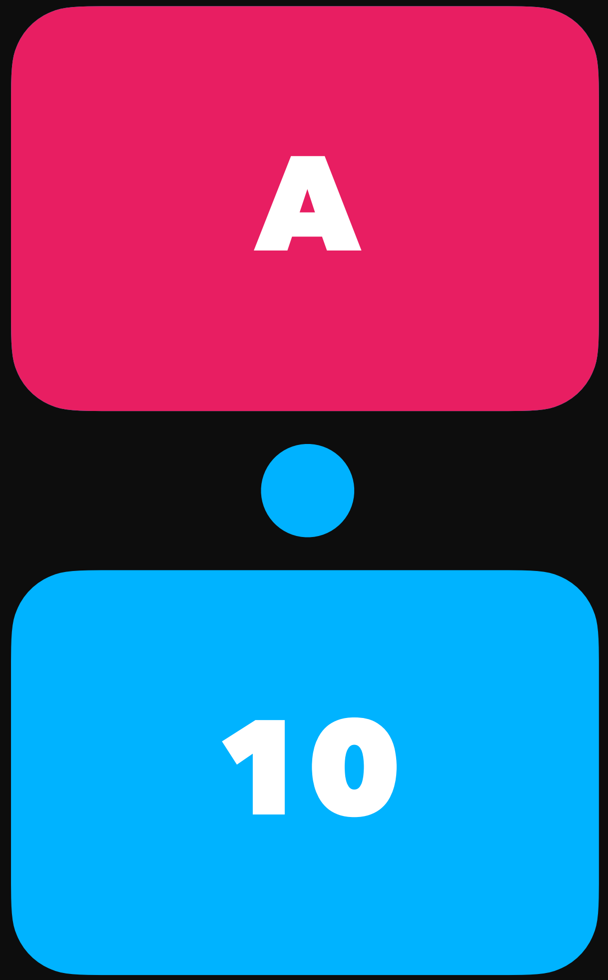
SS2PL/R2PL PROTOCOL



RECOVERABILITY







TRANSACTION ISSUES

LOST UPDATE

DIRTY READ

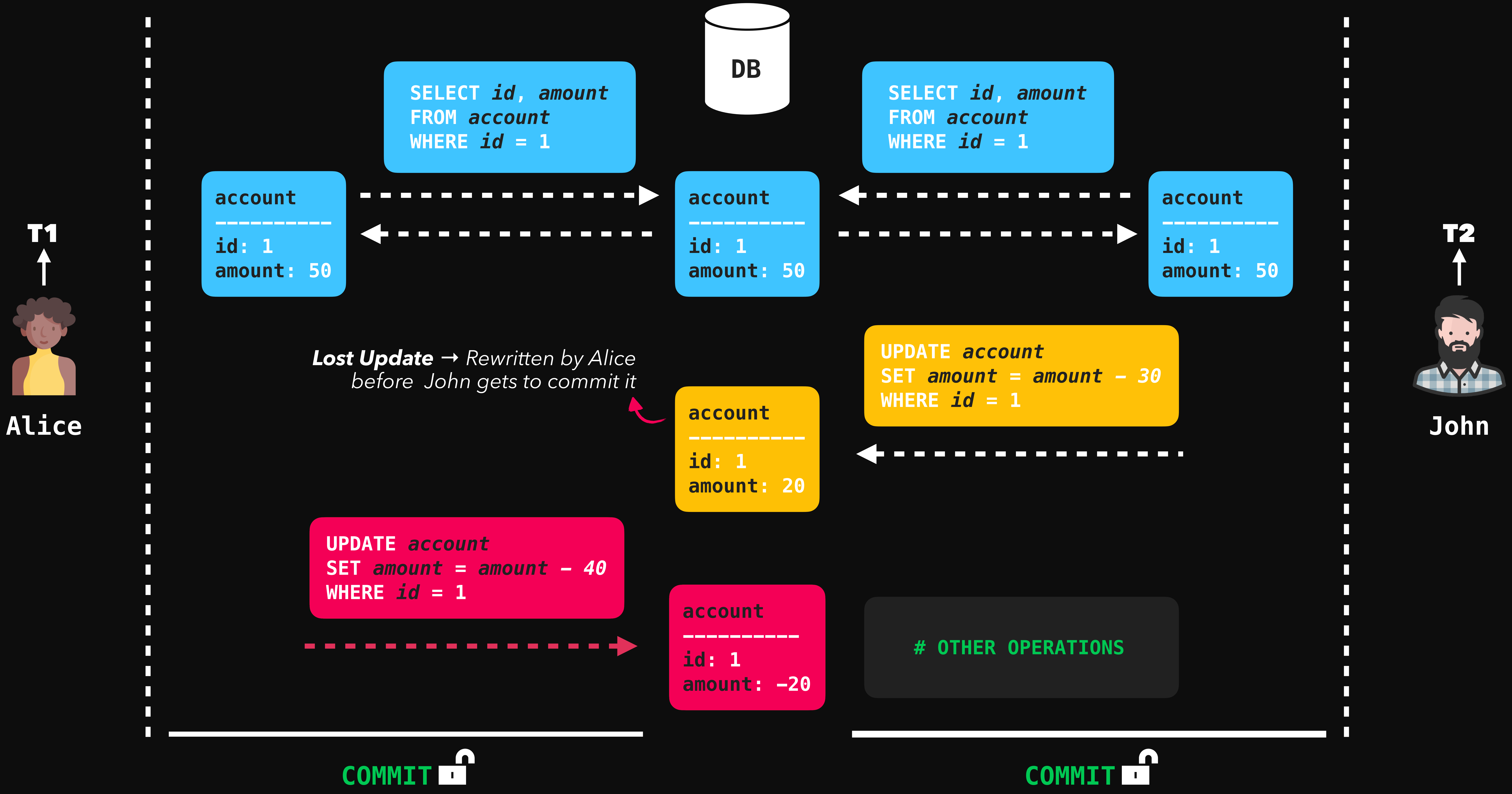
INCORRECT SUMMARY

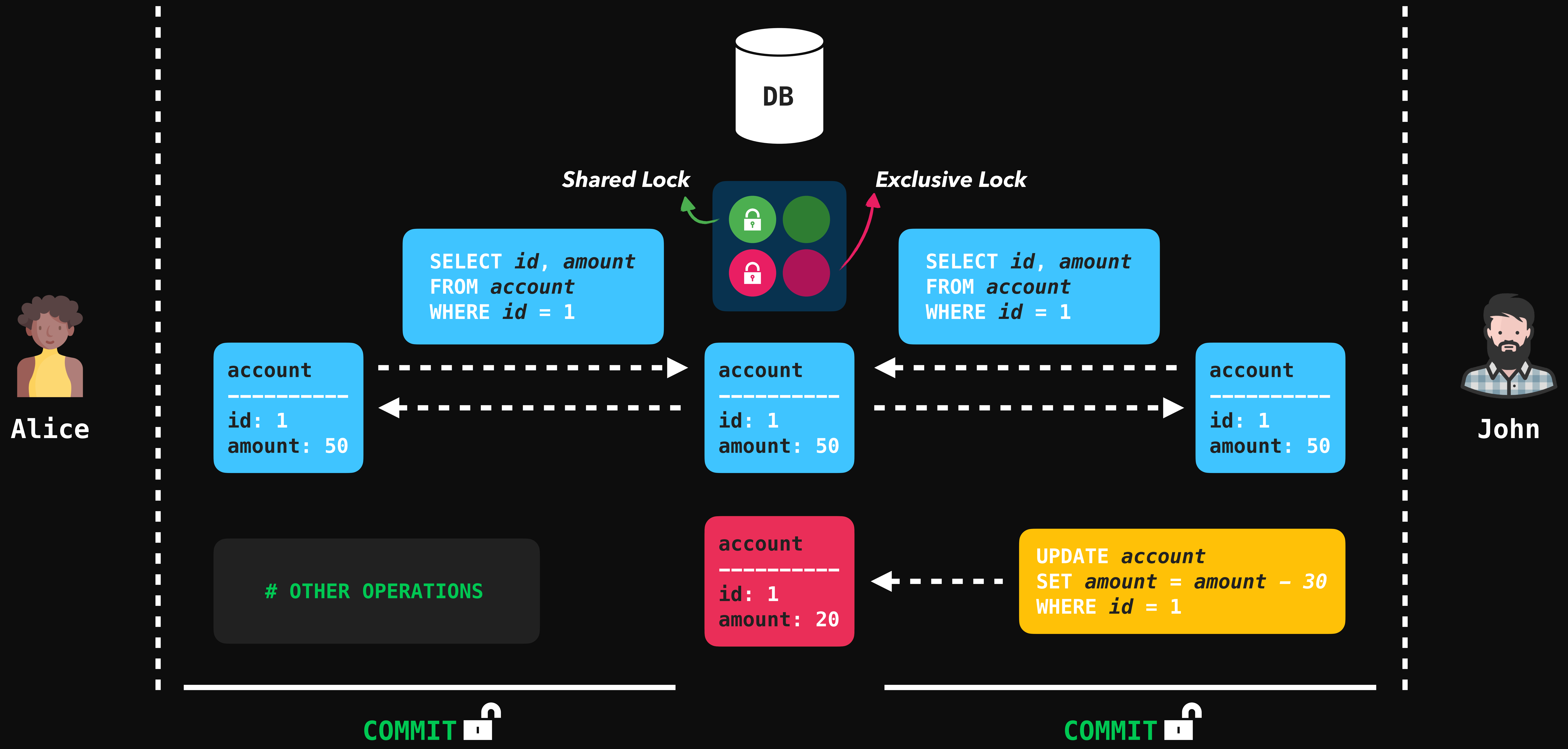
LOCKING TYPES

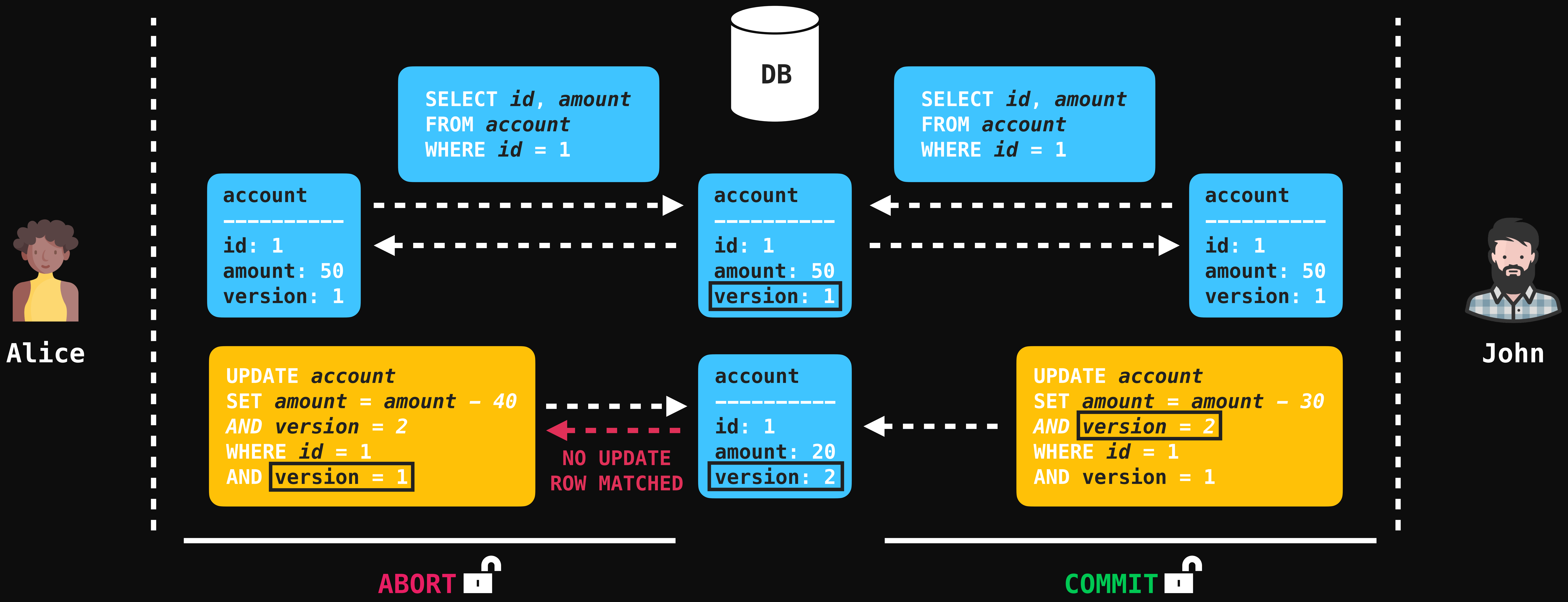
OPTIMISTIC

PESSIMISTIC

SEMI-OPTIMISTIC


DB
T1
Alice
T2
John
SELECT id, amount
FROM account
WHERE id = 1
account
----
id: 1
amount: 50
SELECT id, amount
FROM account
WHERE id = 1
account
----
id: 1
amount: 50
account
----
id: 1
amount: 50
Lost Update → Rewritten by Alice before John gets to commit it
UPDATE account
SET amount = amount - 30
WHERE id = 1
account
----
id: 1
amount: 20
UPDATE account
SET amount = amount - 40
WHERE id = 1
account
----
id: 1
amount: -20
OTHER OPERATIONS
COMMIT
COMMIT


DB
Shared Lock
Exclusive Lock
SELECT id, amount
FROM account
WHERE id = 1
SELECT id, amount
FROM account
WHERE id = 1
Alice
account
----
id: 1
amount: 50
account
----
id: 1
amount: 50
account
----
id: 1
amount: 50
John
OTHER OPERATIONS
account
----
id: 1
amount: 20
UPDATE account
SET amount = amount - 30
WHERE id = 1
COMMIT
COMMIT



Alice
John
DB
SELECT id, amount
FROM account
WHERE id = 1
SELECT id, amount
FROM account
WHERE id = 1
account
----
id: 1
amount: 50
version: 1
account
----
id: 1
amount: 50
version: 1
account
----
id: 1
amount: 50
version: 1
UPDATE account
SET amount = amount - 40
AND version = 2
WHERE id = 1
AND version = 1
NO UPDATE
ROW MATCHED
account
----
id: 1
amount: 20
version: 2
UPDATE account
SET amount = amount - 30
AND version = 2
WHERE id = 1
AND version = 1
ABORT
COMMIT