Atomicity, Locks, Consistency & Project 1

August 22, CS 6530
Yuvaraj Chesetti

Multi-threaded programming

```
chesetti@sn4622111117:~$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   46 bits physical, 57 bits virtual
CPU(s):                          128
On-line CPU(s) list:             0-127
Thread(s) per core:              2
Core(s) per socket:              32
Socket(s):                       2
NUMA node(s):                    2
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           106
Model name:                      Intel(R) Xeon(R) Gold 6338 CPU @ 2.00GHz
Stepping:                        6
CPU MHz:                         800.846
CPU max MHz:                     3200.0000
CPU min MHz:                     800.0000
BogoMIPS:                        4000.00
Virtualization:                  VT-x
L1d cache:                       3 MiB
L1i cache:                       2 MiB
L2 cache:                        80 MiB
L3 cache:                        96 MiB
```

Multithreaded programming can be unintuitive!

Intuition 1 - Operations are atomic by default

```
void incX(int *x, int times) {
  for (int i = 0; i < times; i++) {
    *x = *x + 1;
  }
}
```

Single Threaded

```
x = 0;
incX(&x, 500000);
incX(&x, 500000);
std::cout<< x <<std::endl;
```

Multi Threaded

```
x = 0;
std::thread t1(incX, &x, 500000);
std::thread t2(incX, &x, 500000);
t1.join();
t2.join();
std::cout<< x <<std::endl;
```

```
void incX(int *x, int times) {
  for (int i = 0; i < times; i++) {
    *x = *x + 1;
  }
}
```

Single Threaded

```
x = 0;
incX(&x, 500000);
incX(&x, 500000);
std::cout<< x <<std::endl;
```

→ 1000000

Multi Threaded

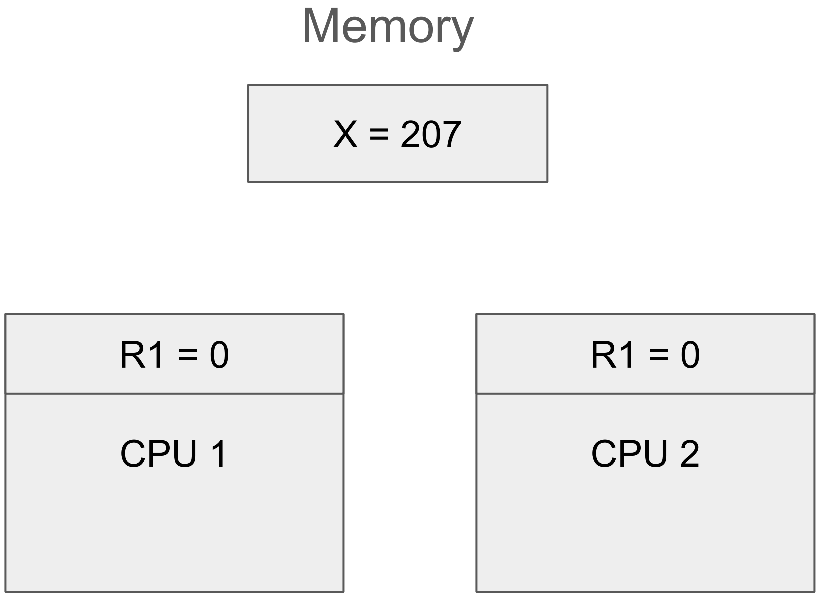
```
x = 0;
std::thread t1(incX, &x, 500000);
std::thread t2(incX, &x, 500000);
t1.join();
t2.join();
std::cout<< x <<std::endl;
```

→ Random (~ 500000)

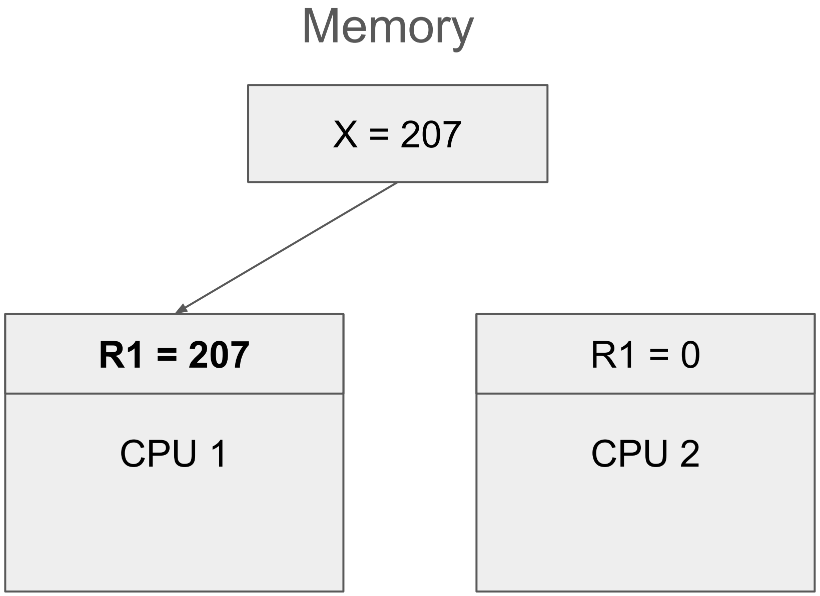
What’s happening?

x = x + 1 is not really 1 instruction

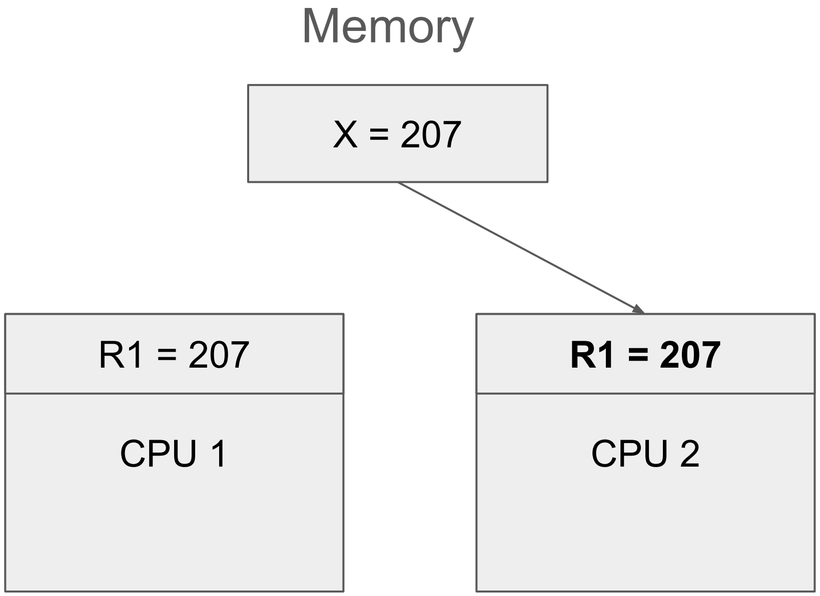
```
ld x, r1
add r1, r1, 1
str x, r1
```



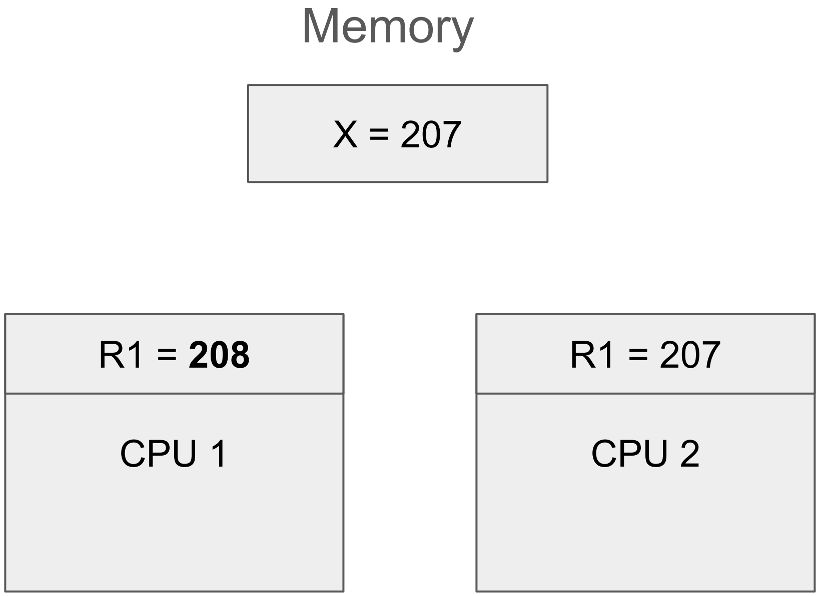
CPU 1	CPU 2



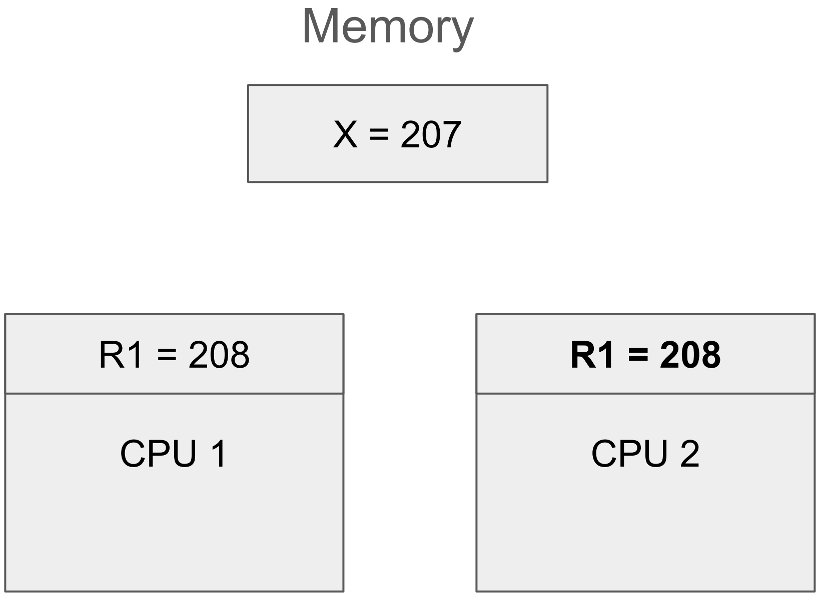
CPU 1	CPU 2
ld x, r1	



CPU 1	CPU 2
ld x, r1	ld x, r1



CPU 1	CPU 2
ld x, r1	ld x, r1
add r1, r1, 1	

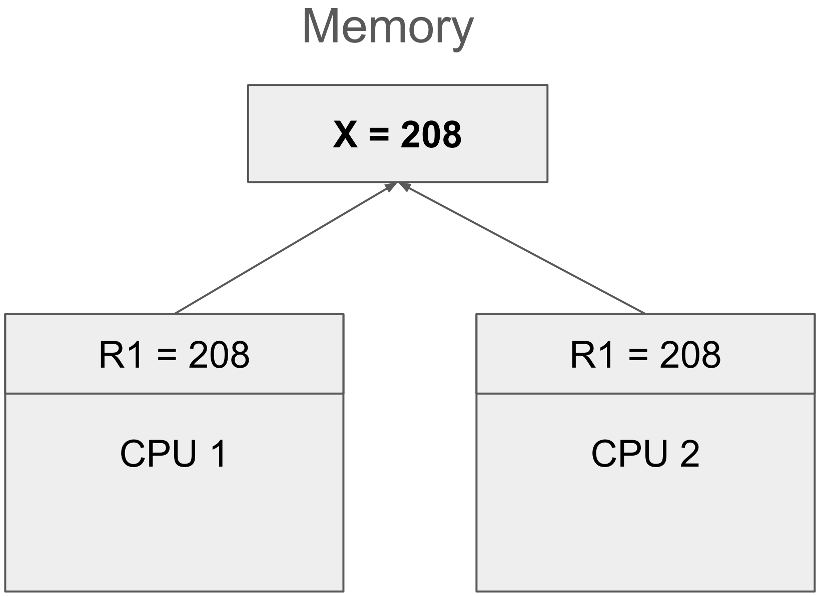


CPU 1

```
ld x, r1
add r1, r1, 1
```

CPU 2

```
ld x, r1
add r1, r1, 1
```

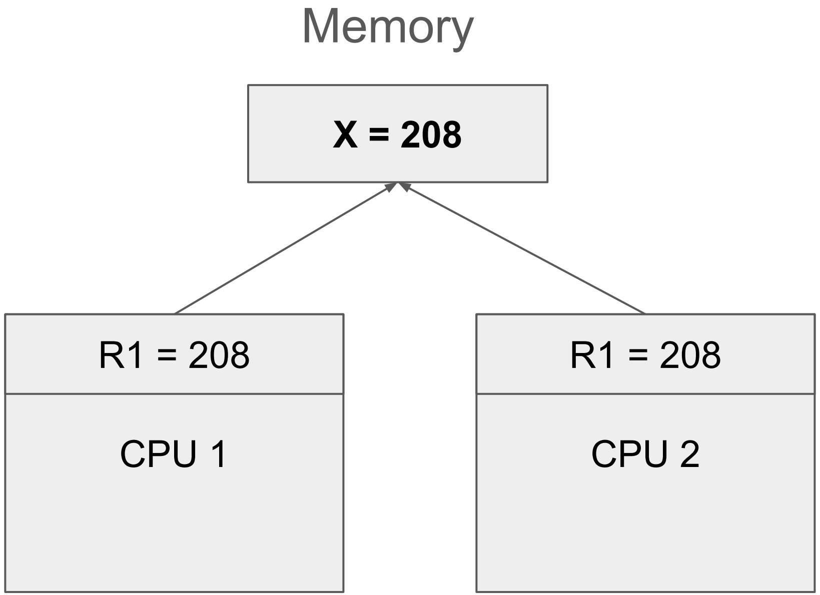


CPU 1

ld x, r1
add r1, r1, 1
st x,r1

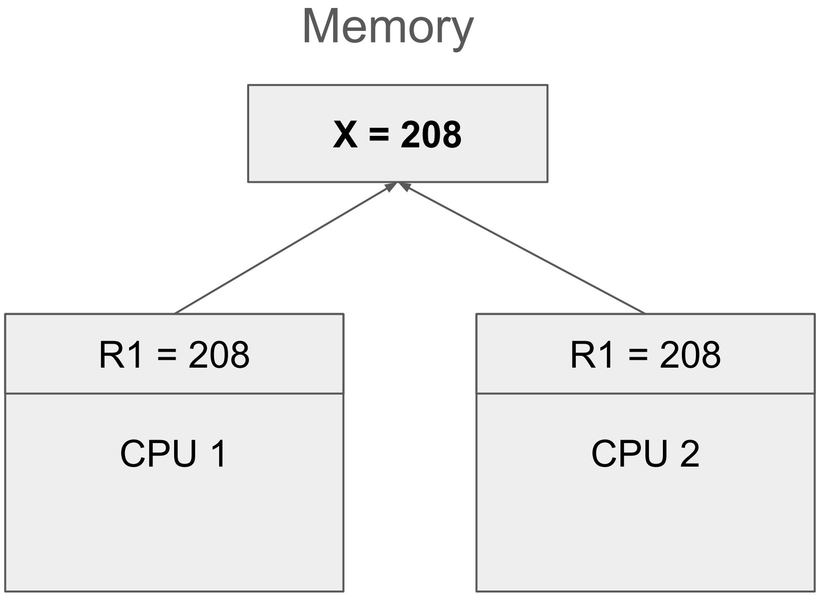
CPU 2

ld x, r1
add r1, r1, 1
st x, r1



CPU 1	CPU 2
ld x, r1	ld x, r1
add r1, r1, 1	add r1, r1, 1
st x,r1	st x, r1

What went wrong?



CPU 1

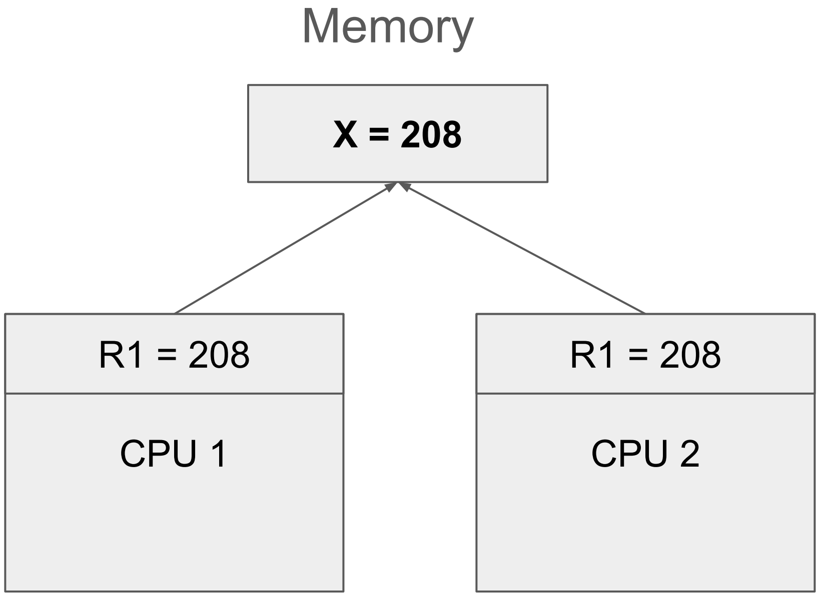
```
ld x, r1
add r1, r1, 1
st x,r1
```

CPU 2

```
ld x, r1
add r1, r1, 1
st x, r1
```

What went wrong?

We expect
x = x + 1 to be executed as
one step by one thread



CPU 1	CPU 2
ld x, r1	ld x, r1
add r1, r1, 1	add r1, r1, 1
st x,r1	st x, r1

What went wrong?

We expect
x = x + 1 to be **Atomic**!

Atomics in C

GNU builtin atomics

```
void __atomic_load (type *ptr, type *ret, int memorder)

void __atomic_store (type *ptr, type *val, int memorder)

type __atomic_add_fetch (type *ptr, type val, int memorder)


type __sync_lock_test_and_set (type *ptr, type value, ...)
     Atomically set *ptr to value, return old value

void __sync_release (type *ptr)
     Atomically set *ptr to 0
```

FULL LIST AT: https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html

```cpp
void incX(int *x, int times) {
  for (int i = 0; i < times; i++) {
    *x = *x + 1;
  }
}
```

Single Threaded

```cpp
x = 0;
incX(&x, 500000);
incX(&x, 500000);
std::cout<< x <<std::endl;
```

Multi Threaded

```cpp
x = 0;
std::thread t1(incX, &x, 500000);
std::thread t2(incX, &x, 500000);
t1.join();
t2.join();
std::cout<< x <<std::endl;
```

```
void incX(int *x, int times) {
  for (int i = 0; i < times; i++) {
    __atomic_add_fetch(x, 1, __ATOMIC_SEQ_CST);
  }
}
```

Single Threaded

```
x = 0;
incX(&x, 500000);
incX(&x, 500000);
std::cout<< x <<std::endl;
```

→ 1000000

Multi Threaded

```
x = 0;
std::thread t1(incX, &x, 500000);
std::thread t2(incX, &x, 500000);
t1.join();
t2.join();
std::cout<< x <<std::endl;
```

→ 1000000

Level of Atomicity

- Are atomics enough?
- What about objects or Read-Modify-Writes?

```
mutateObject(*obj, f1, f2) {
    atomic_store(obj->field_1, f1)
    atomic_store(obj->field_2, f2)
}
```

Level of Atomicity

- Are atomics enough?
- What about objects or Read-Modify-Writes?

```
mutateObject(*obj, f1, f2) {
   atomic_store(obj->field_1, f1)
   atomic_store(obj->field_2, f2)
}

Thread 1 -> mutateObject(obj, x, x)
Thread 2 -> mutateObject(obj, y, y)
```

Level of Atomicity

- Are atomics enough?
- What about objects or Read-Modify-Writes?

```
mutateObject(*obj, f1, f2) {
    atomic_store(obj->field_1, f1)
    atomic_store(obj->field_2, f2)
}

Thread 1 -> mutateObject(obj, x, x)
Thread 2 -> mutateObject(obj, y, y)

assert(obj->field_1 == obj->field_2)
```

Can this assertion fail?

Level of Atomicity

```
t1 sets field_1 to x
t2 sets field_1 to y
t2 sets field_2 to y
t1 sets field_2 to x
```

```
Result:
{field_1 = y, field_2 = x}
```

Individual operations are atomic, but the entire function is not!

Problem: Function is not atomic

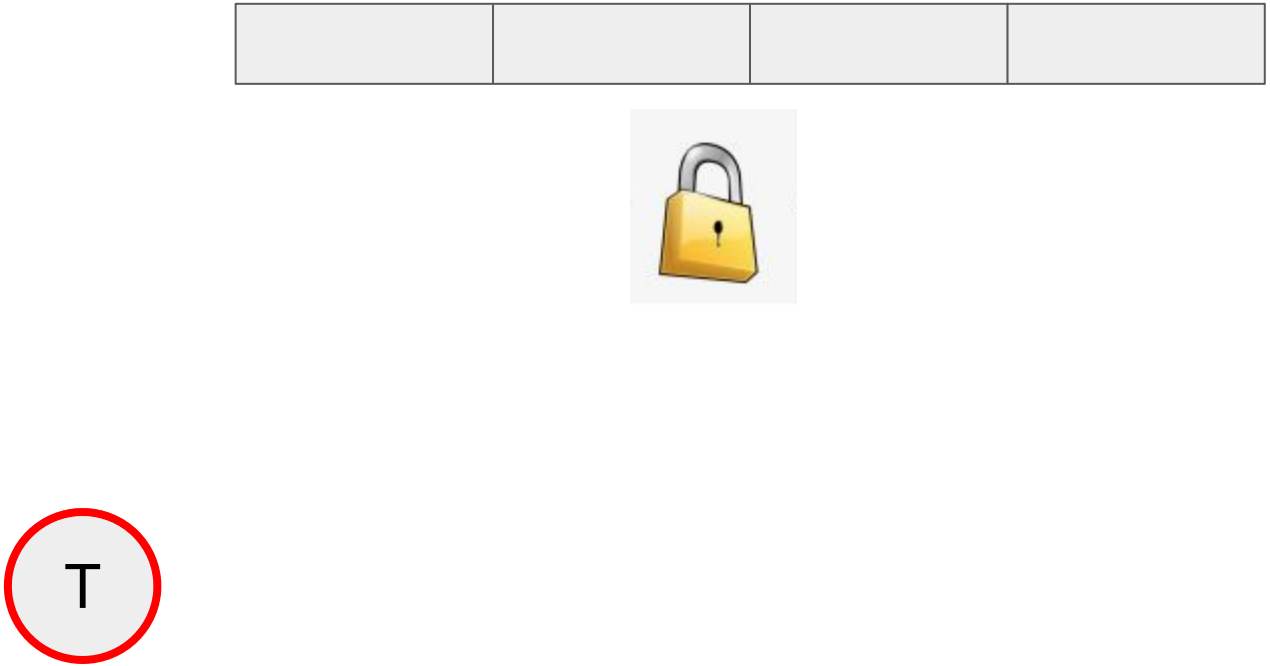
Critical Section - Locks

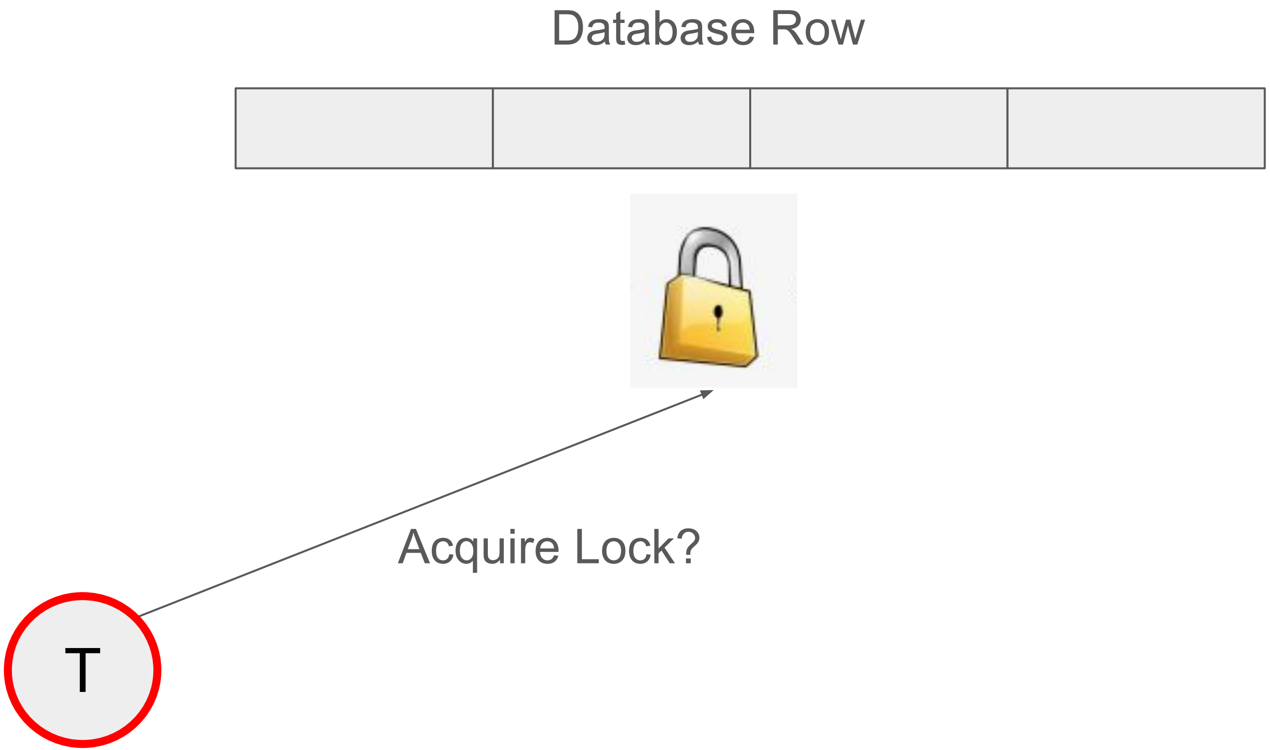
- Locks - barriers that prevent multiple threads entering critical section

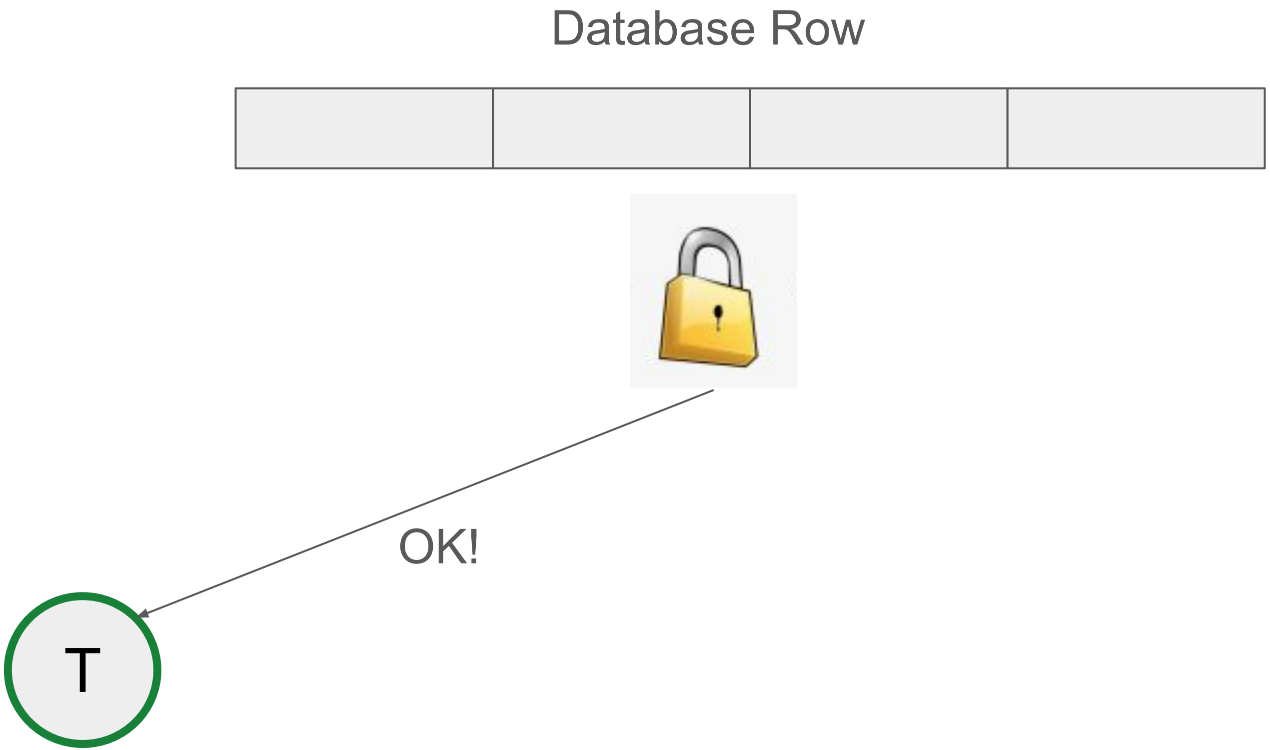
```
mutateObject(*obj, f1, f2) {
    acquire(obj->lock)
// Critical Section Start

    atomic_store(obj->field_1, f1)
    atomic_store(obj->field_2, f2)

// Critical Section End
    release(obj->lock)
}
```

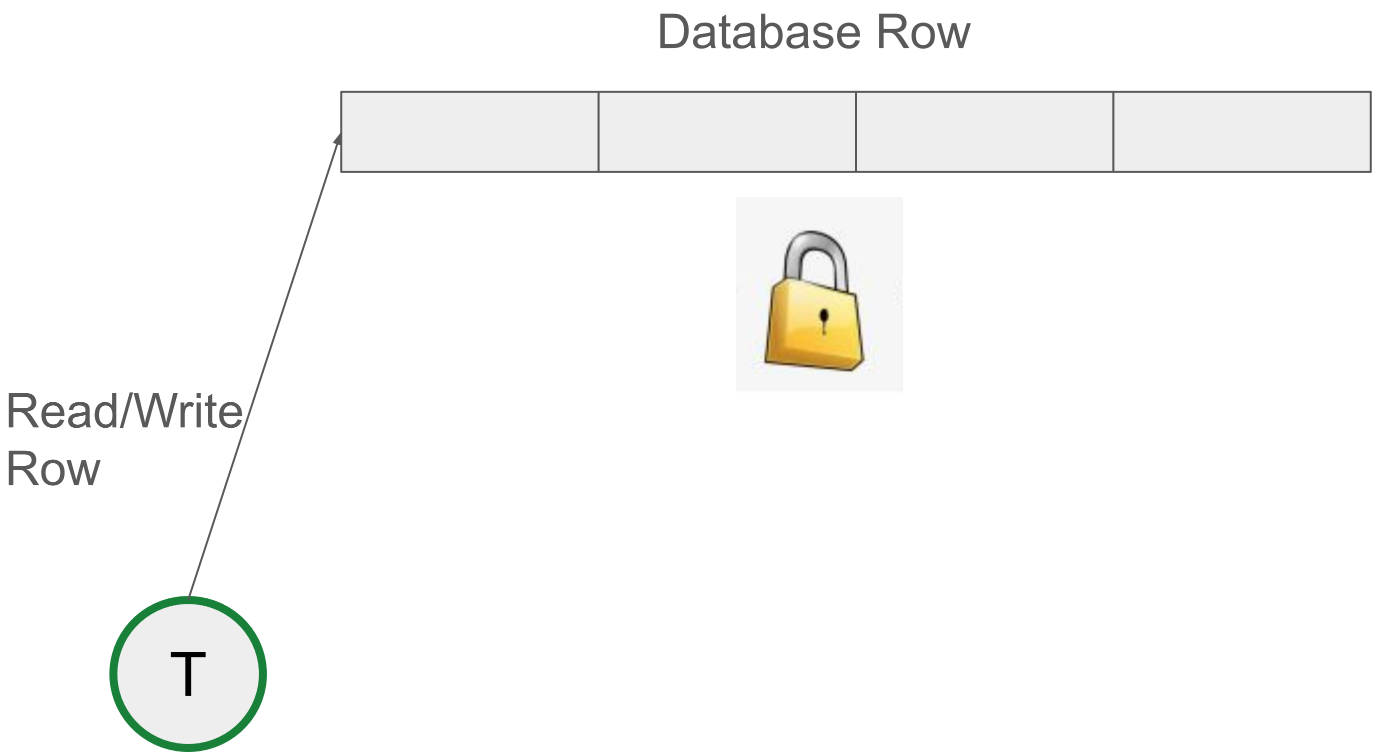
Only 1 thread should be in this section

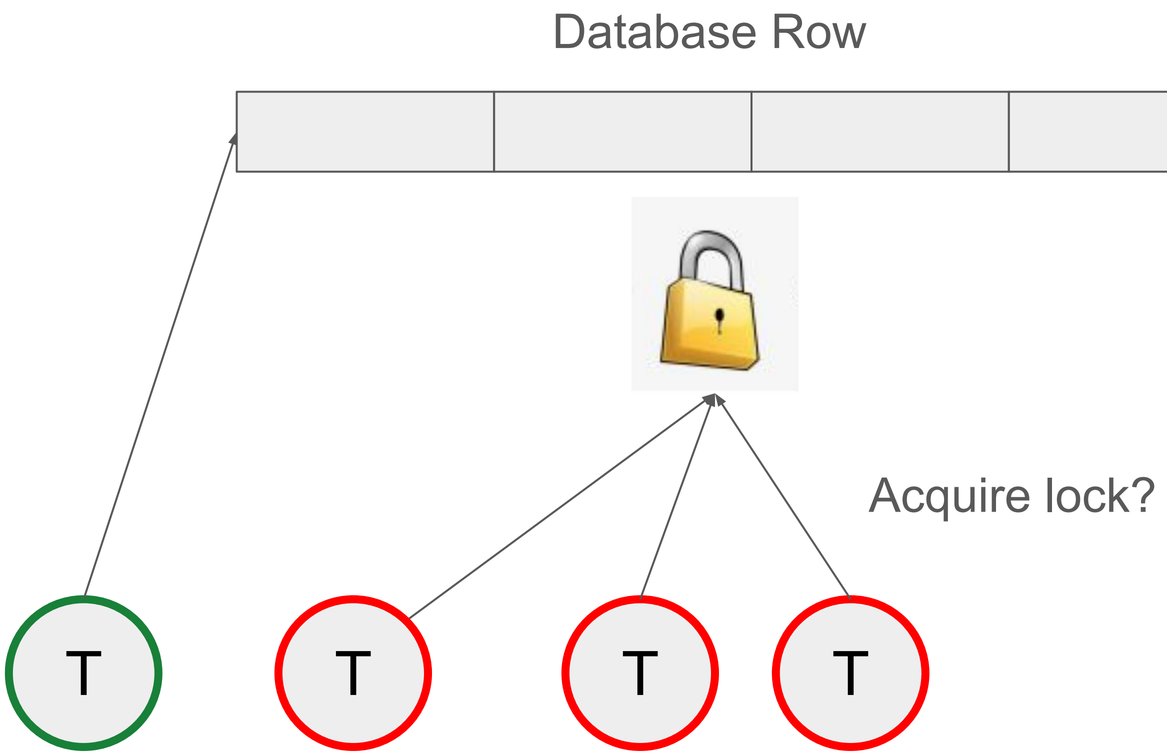
Database Row

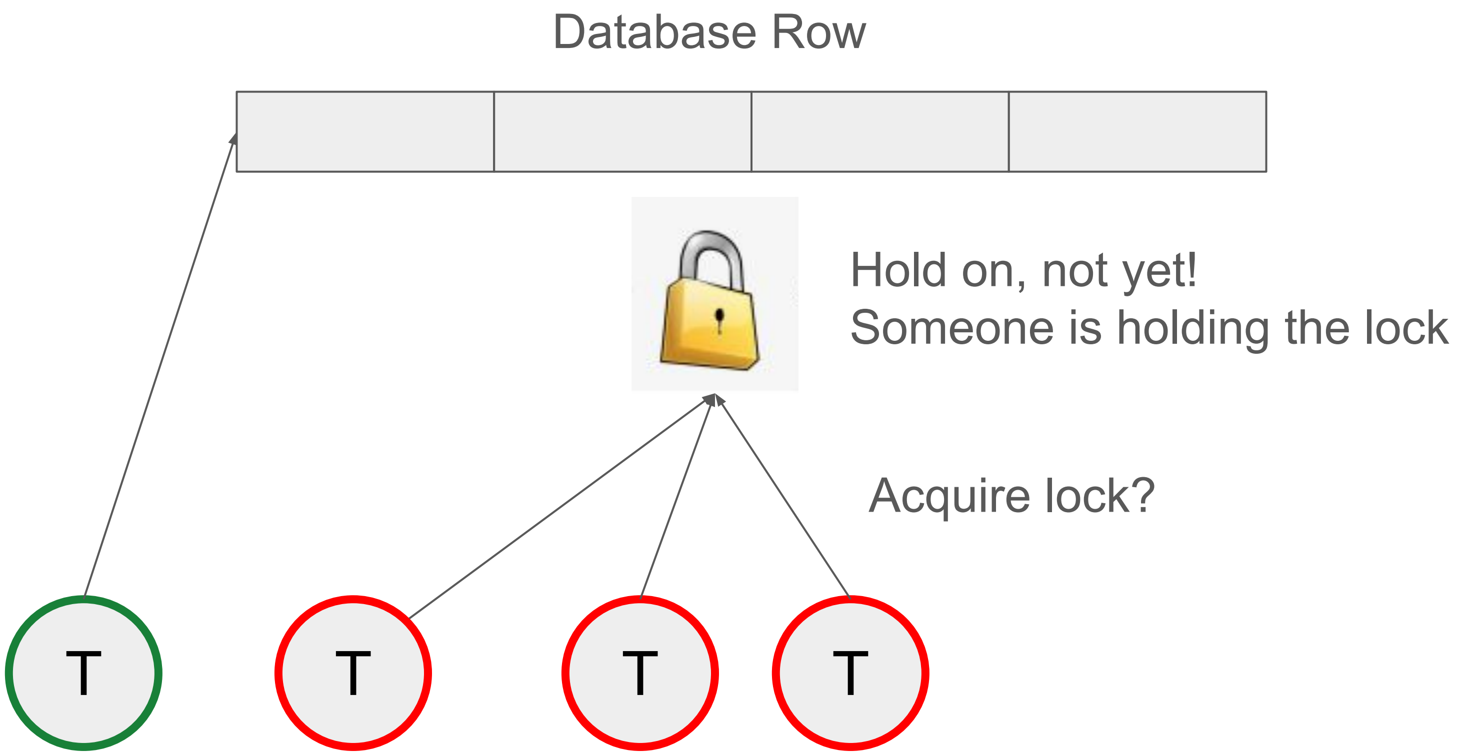



Database Row
Acquire Lock?
T

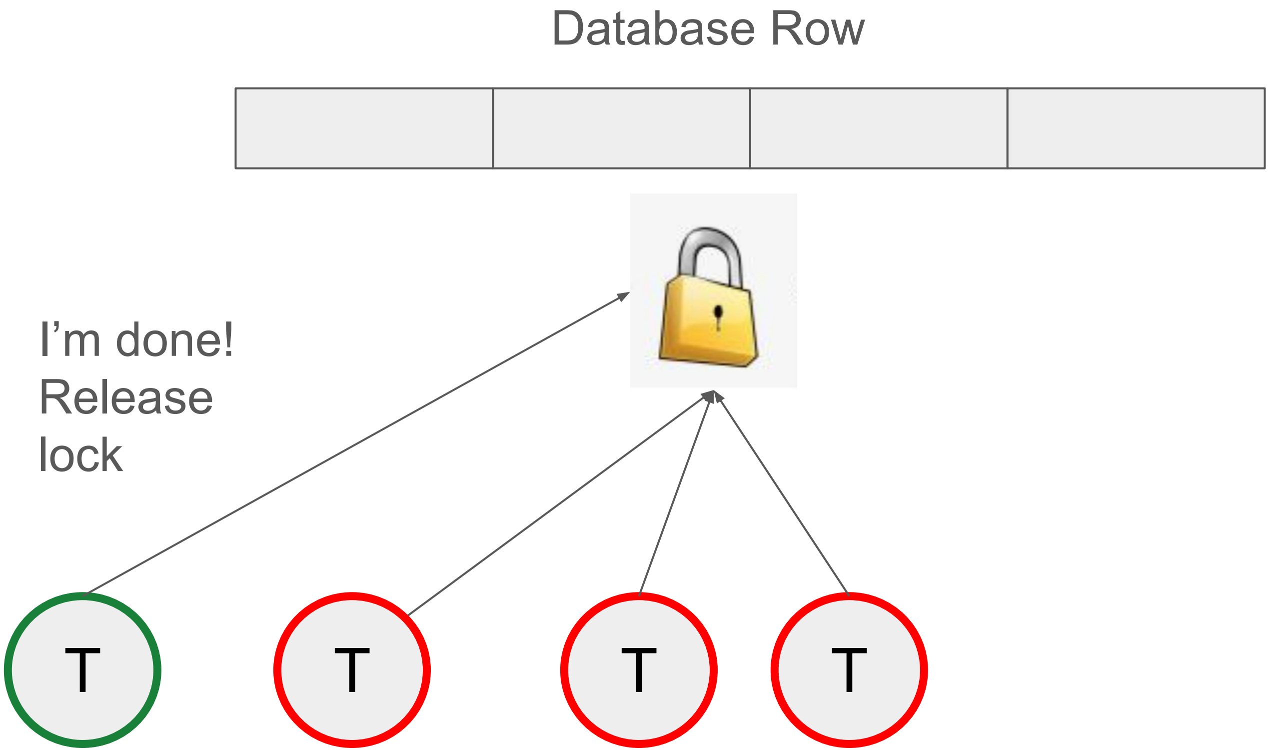

Database Row
OK!
T


Database Row
Read/Write
Row
T


Database Row
Acquire lock?
T
T
T
T

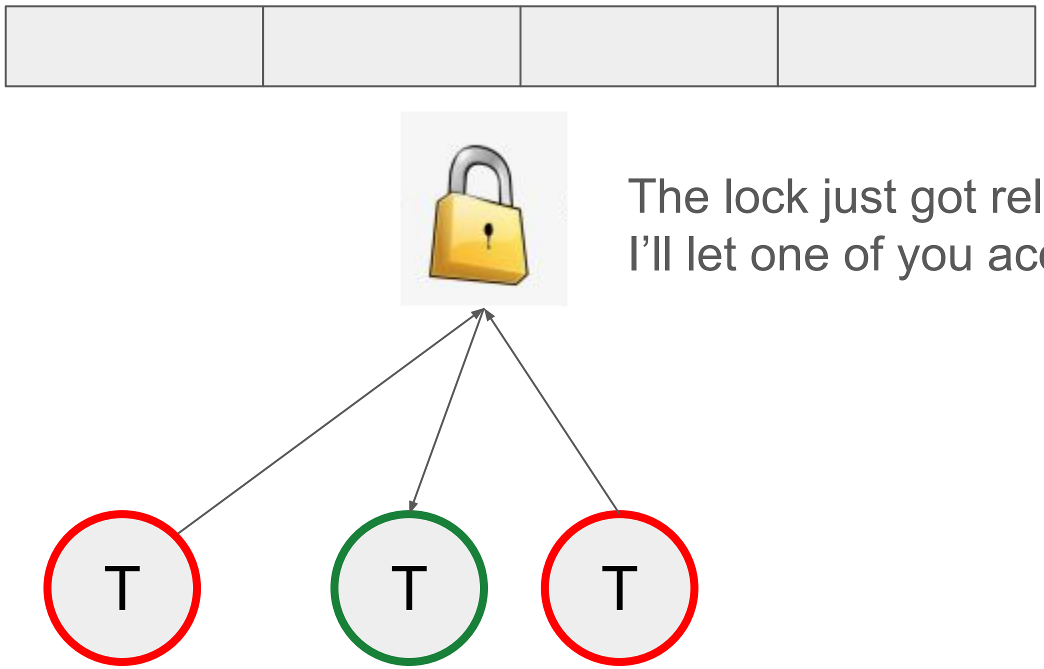


Database Row
Hold on, not yet!
Someone is holding the lock
Acquire lock?
T
T
T
T

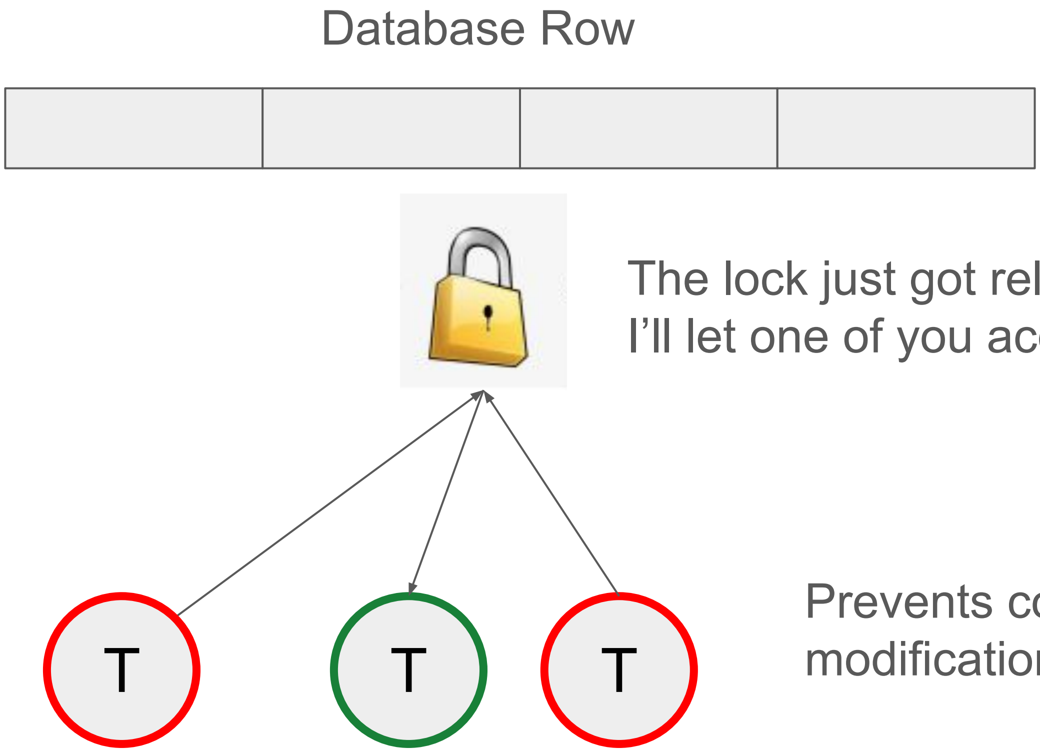


Database Row
I’m done!
Release
lock
T
T
T
T

Database Row



The lock just got released!
I’ll let one of you acquire the lock



The lock just got released!
I’ll let one of you acquire the lock

Prevents concurrent modifications

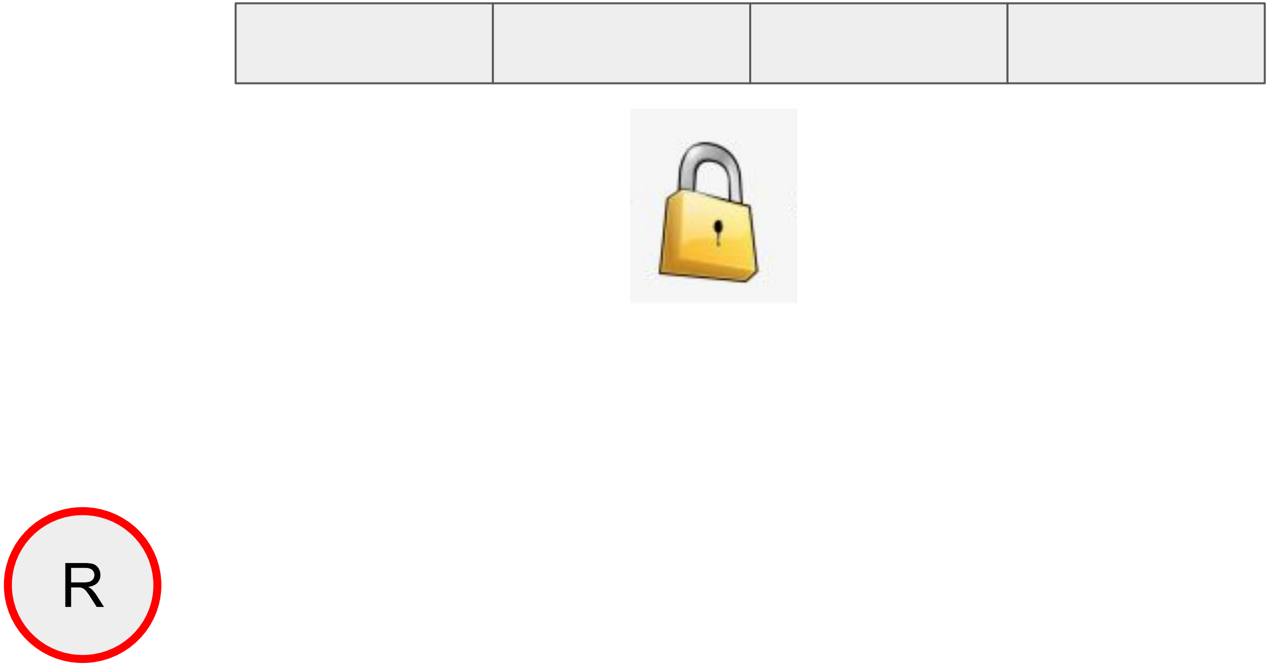
ReaderWriter Lock

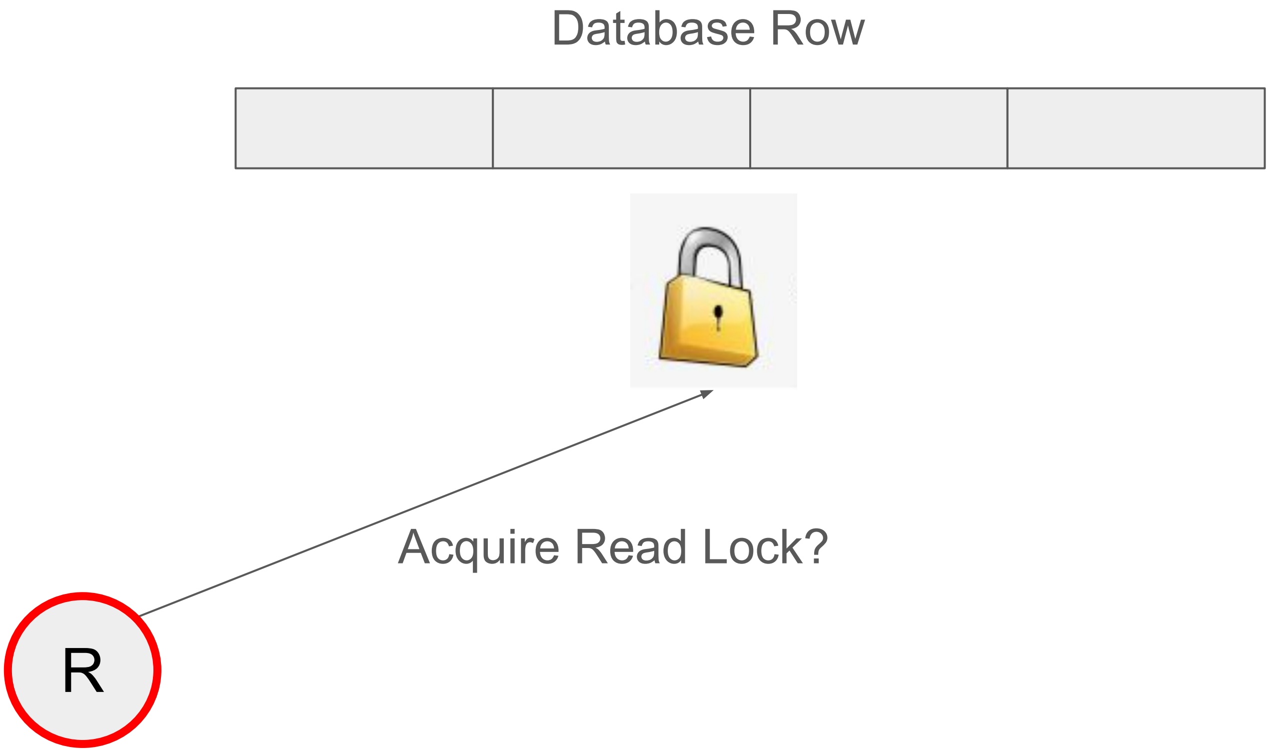
Q: If all the threads are only reading, is it ok to let them run concurrently?

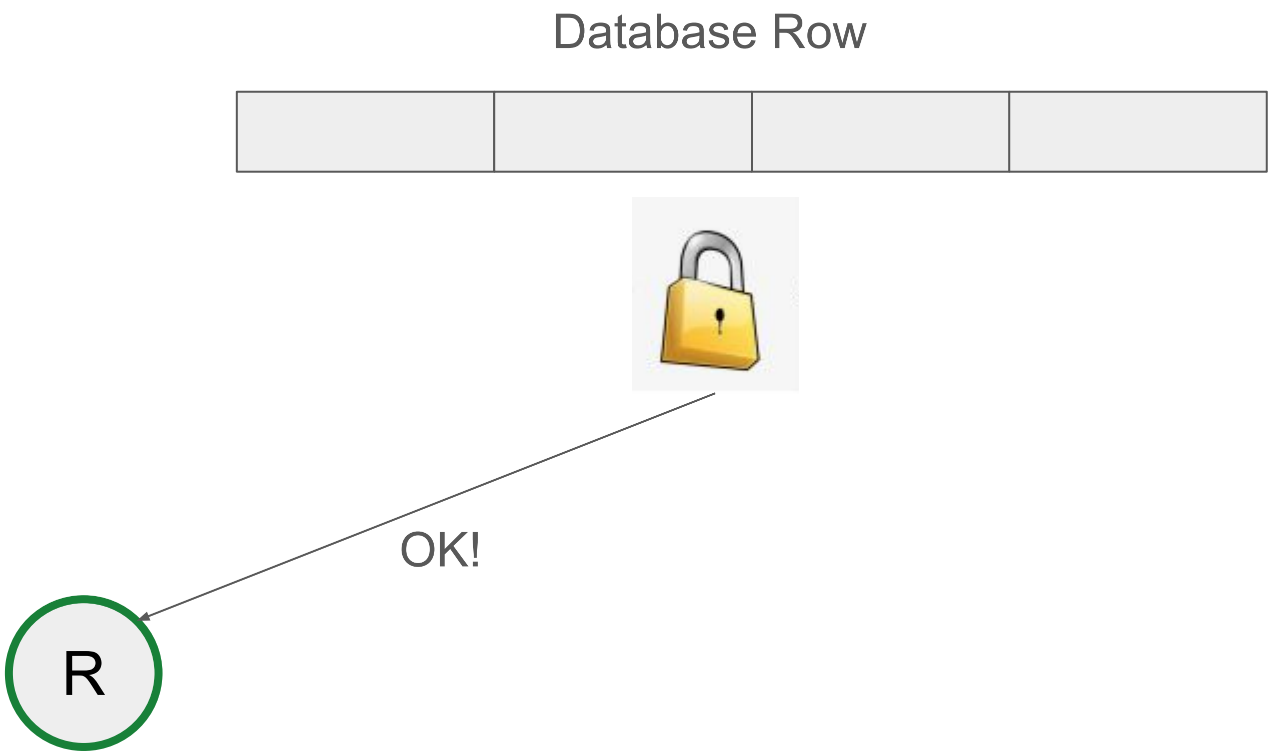
YES!

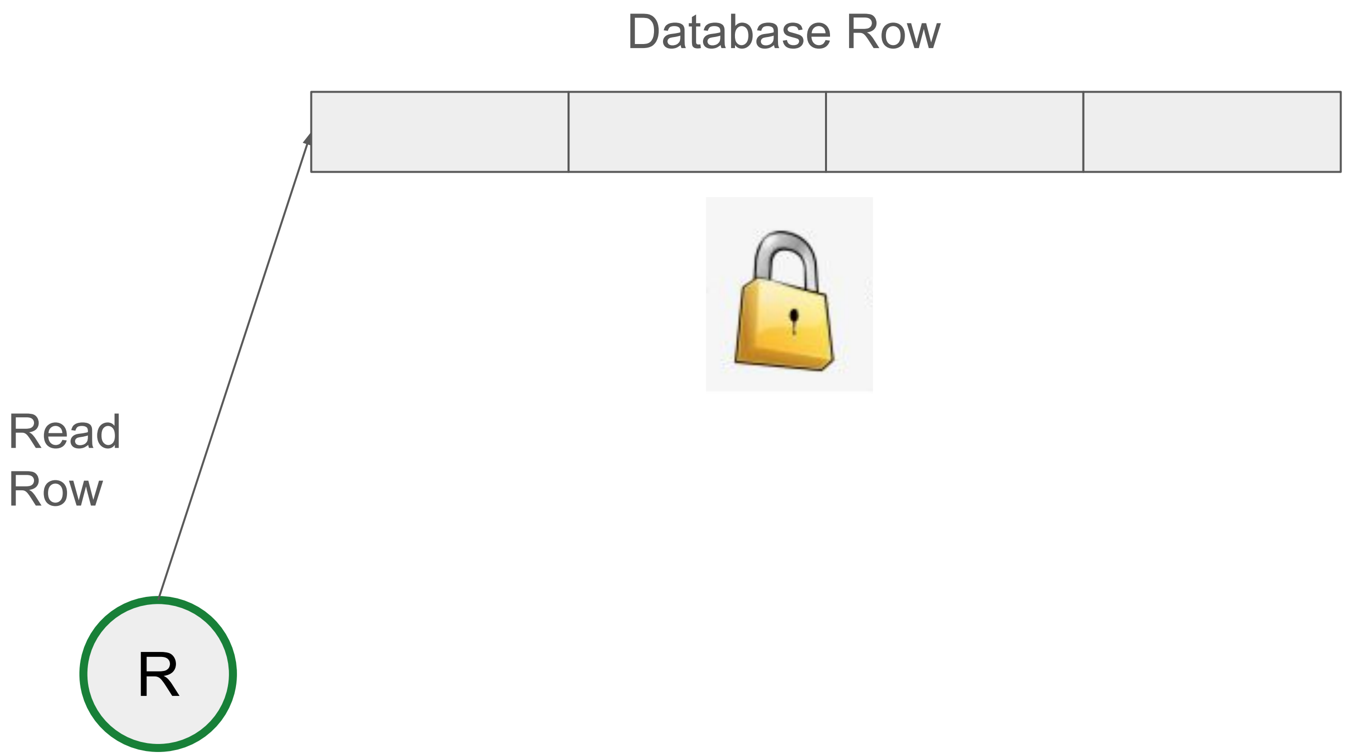
- The ReaderWriter lock is an extension to a simple lock which
    - Allows concurrent access to readers
    - Exclusive access to writers

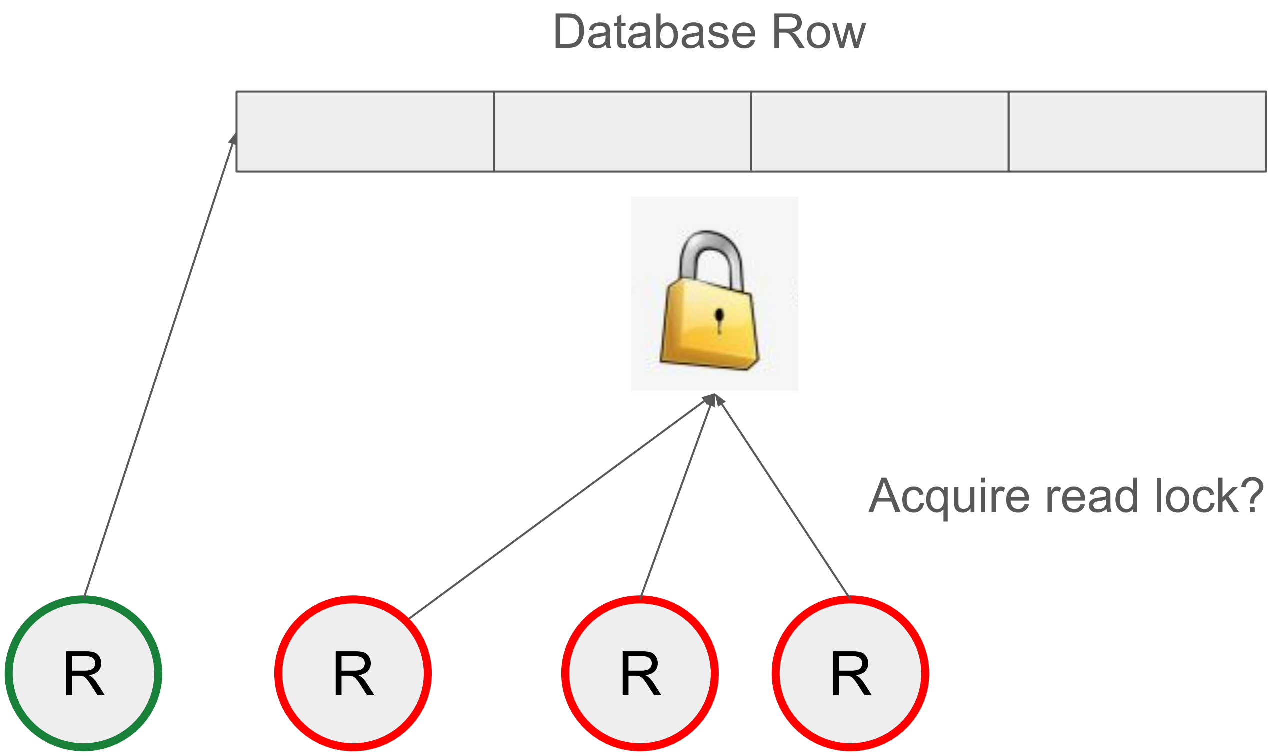
Database Row



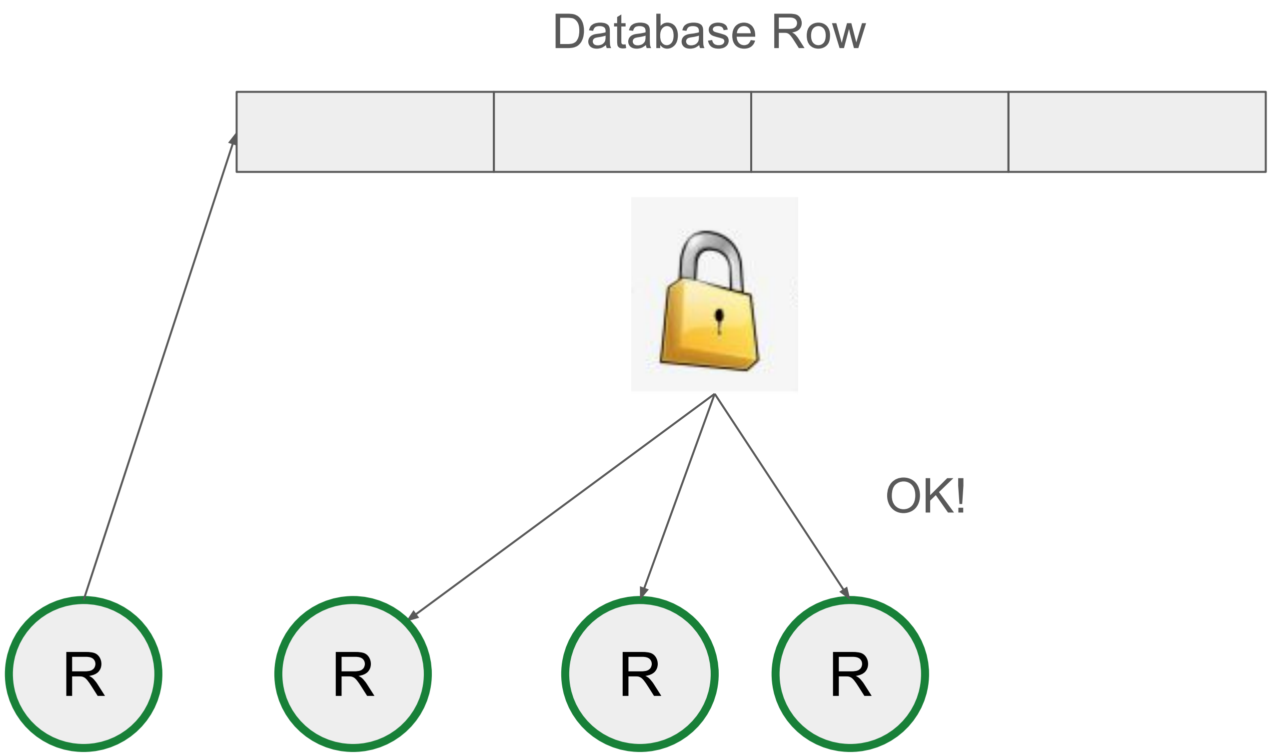

Database Row
Acquire Read Lock?
R

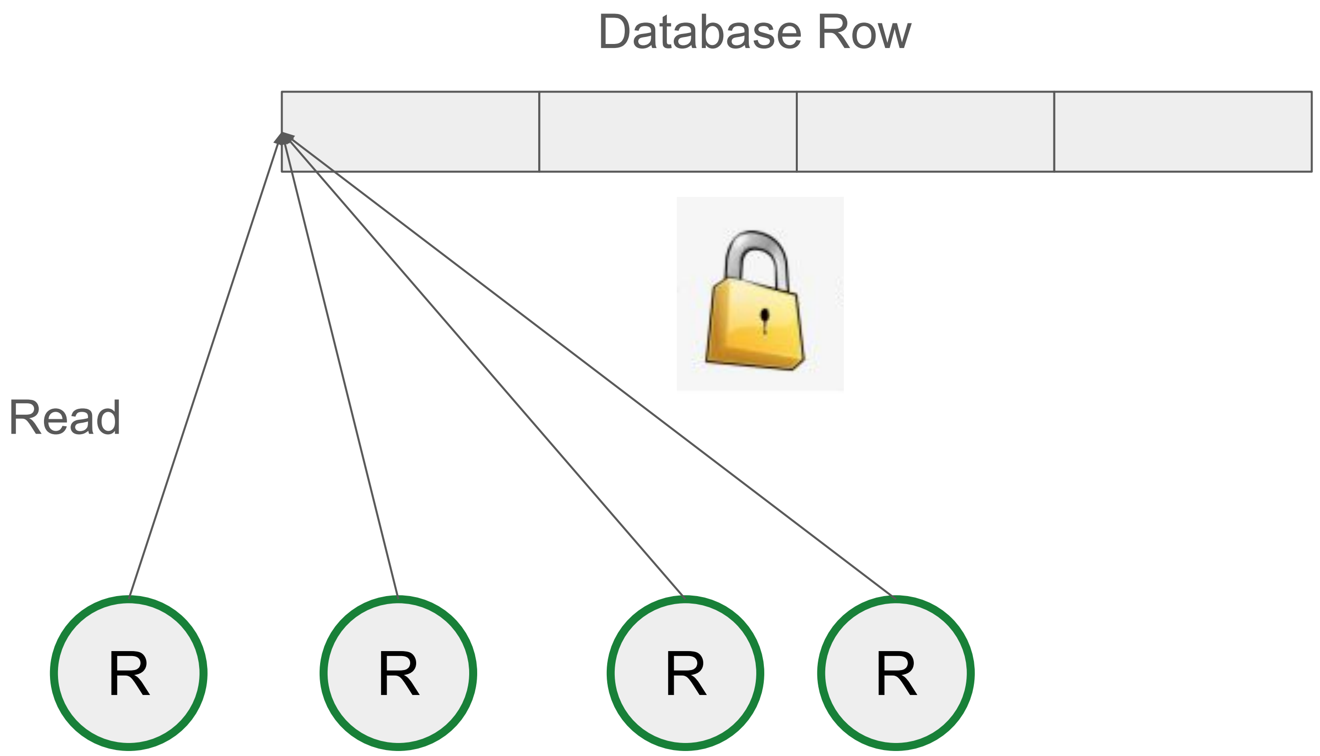

Database Row
OK!
R

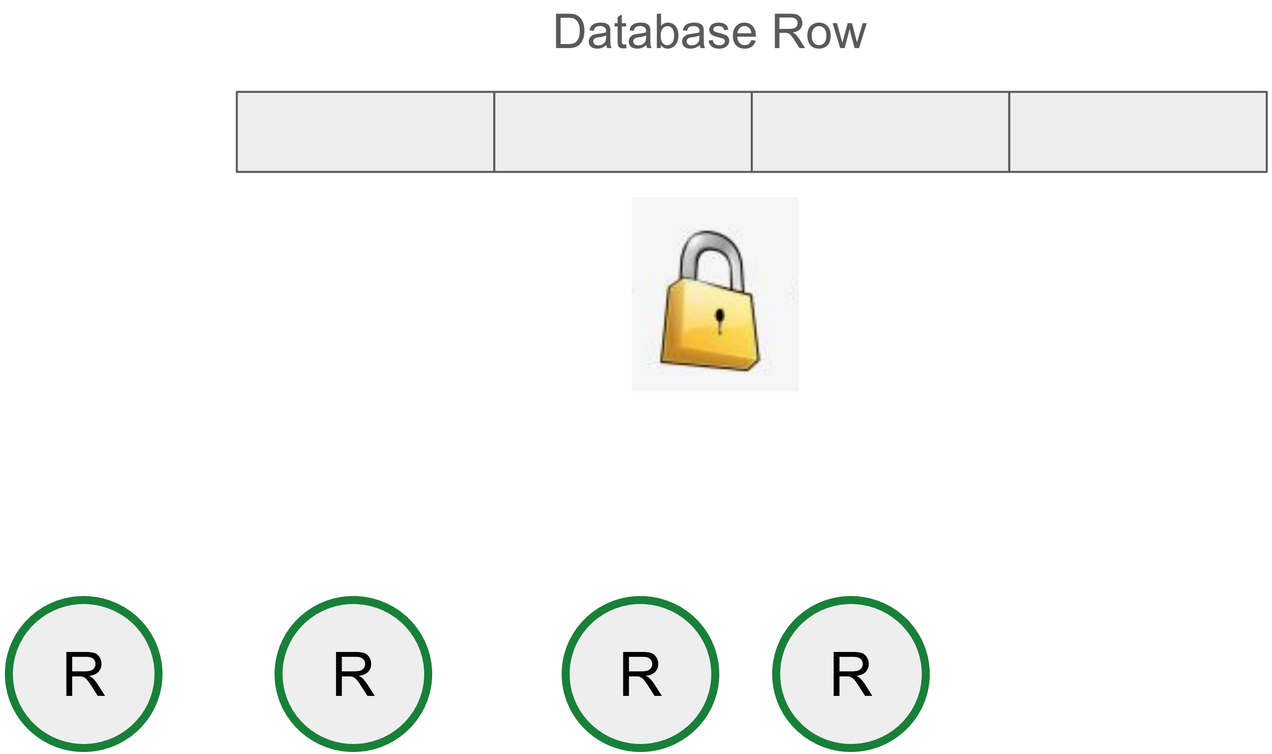

Database Row
Read
Row
R

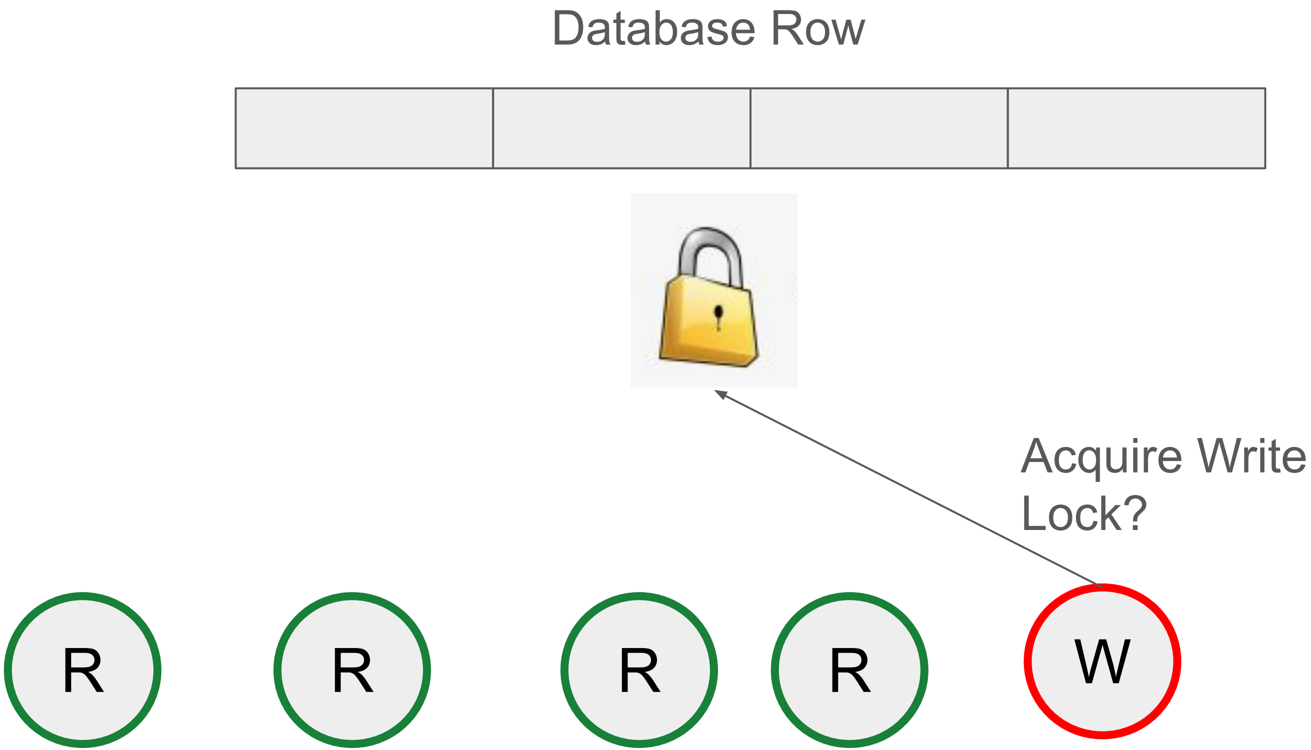


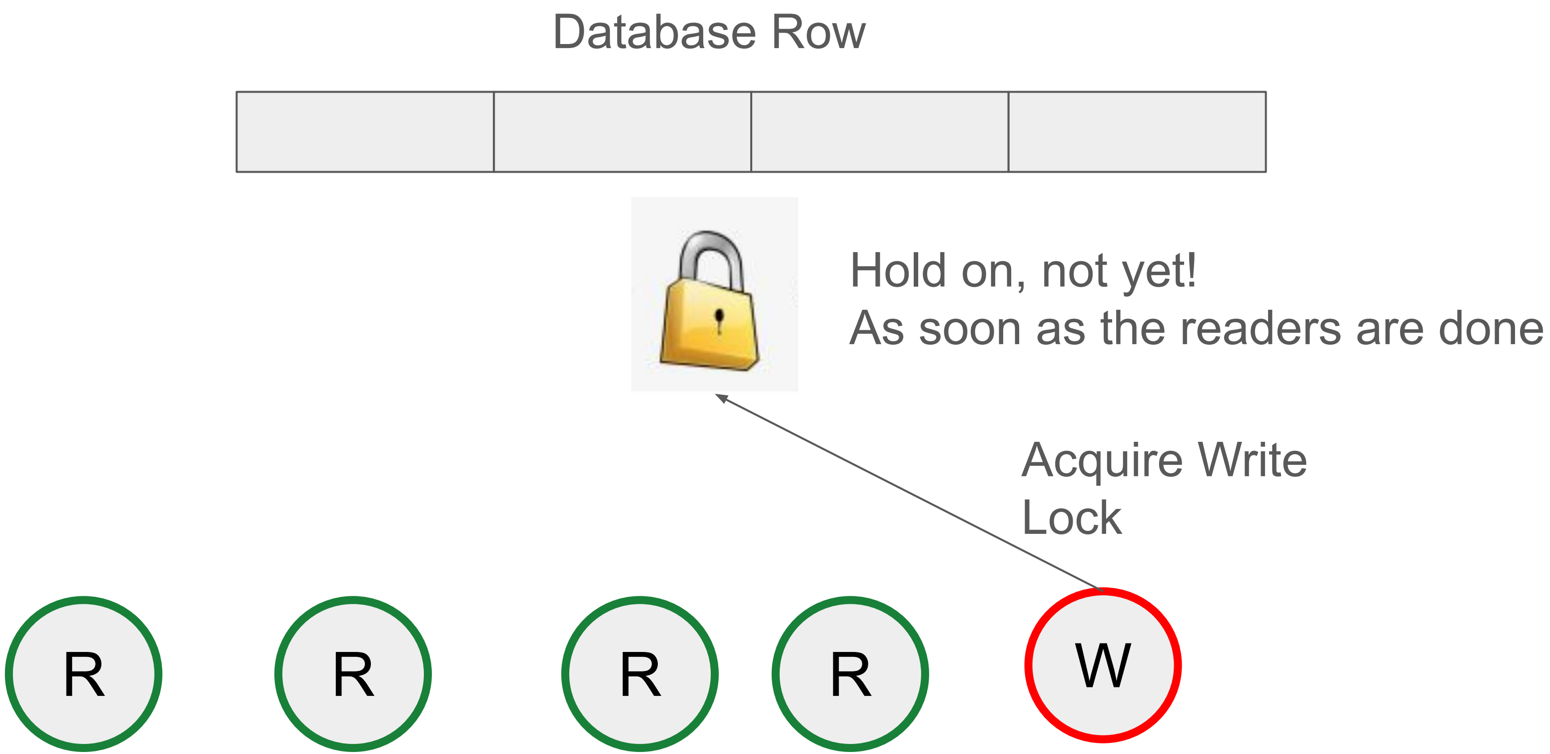
Database Row
Acquire read lock?
R
R
R
R

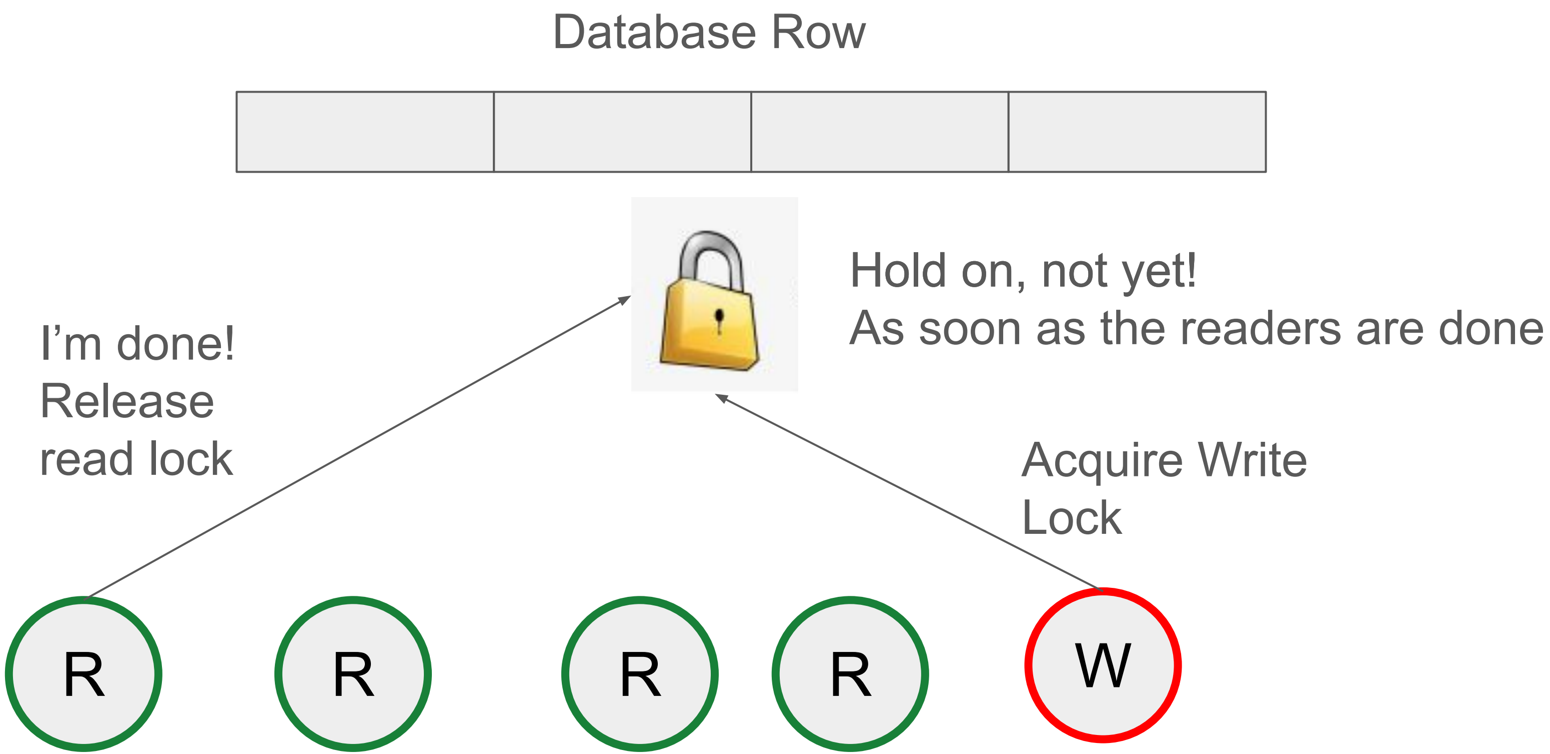

Database Row
OK!
R
R
R
R


Database Row
Read
R
R
R
R

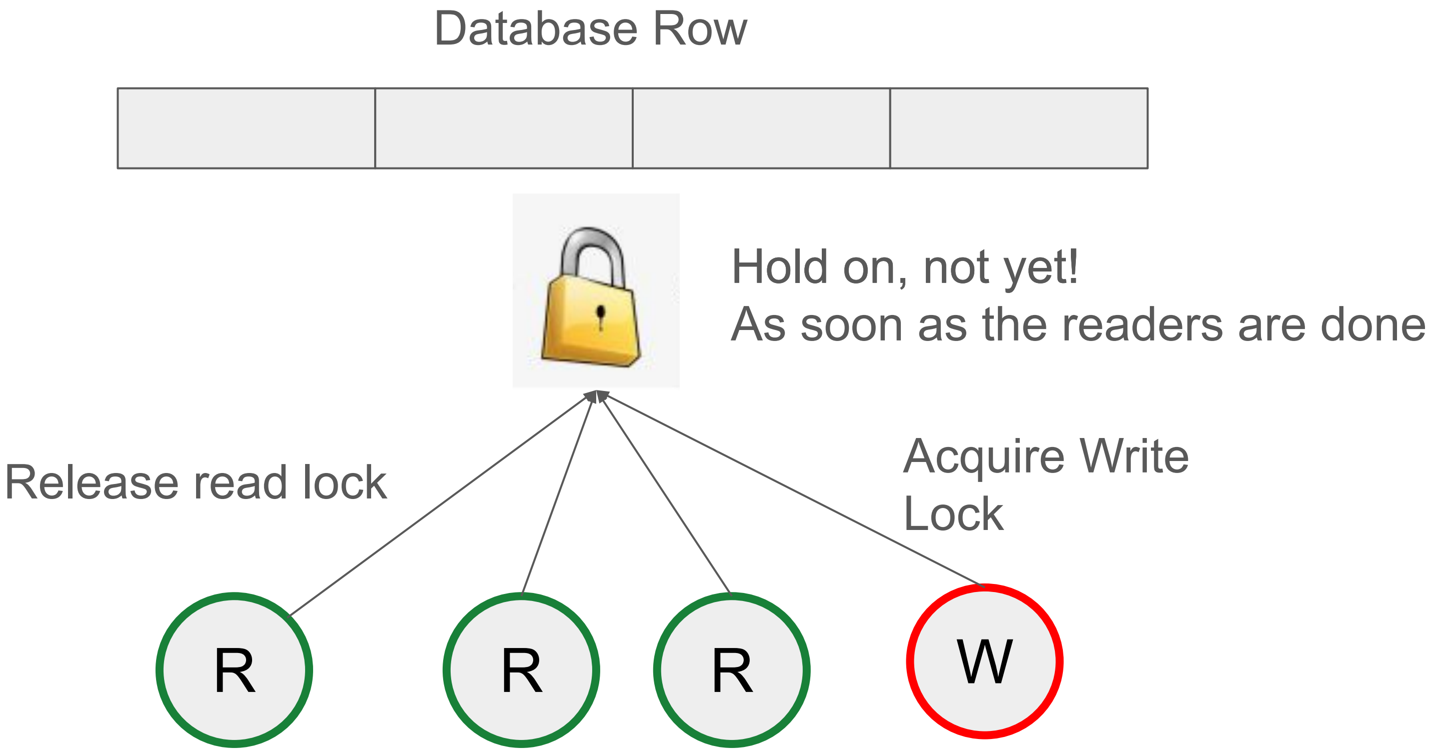

Database Row
R
R
R
R

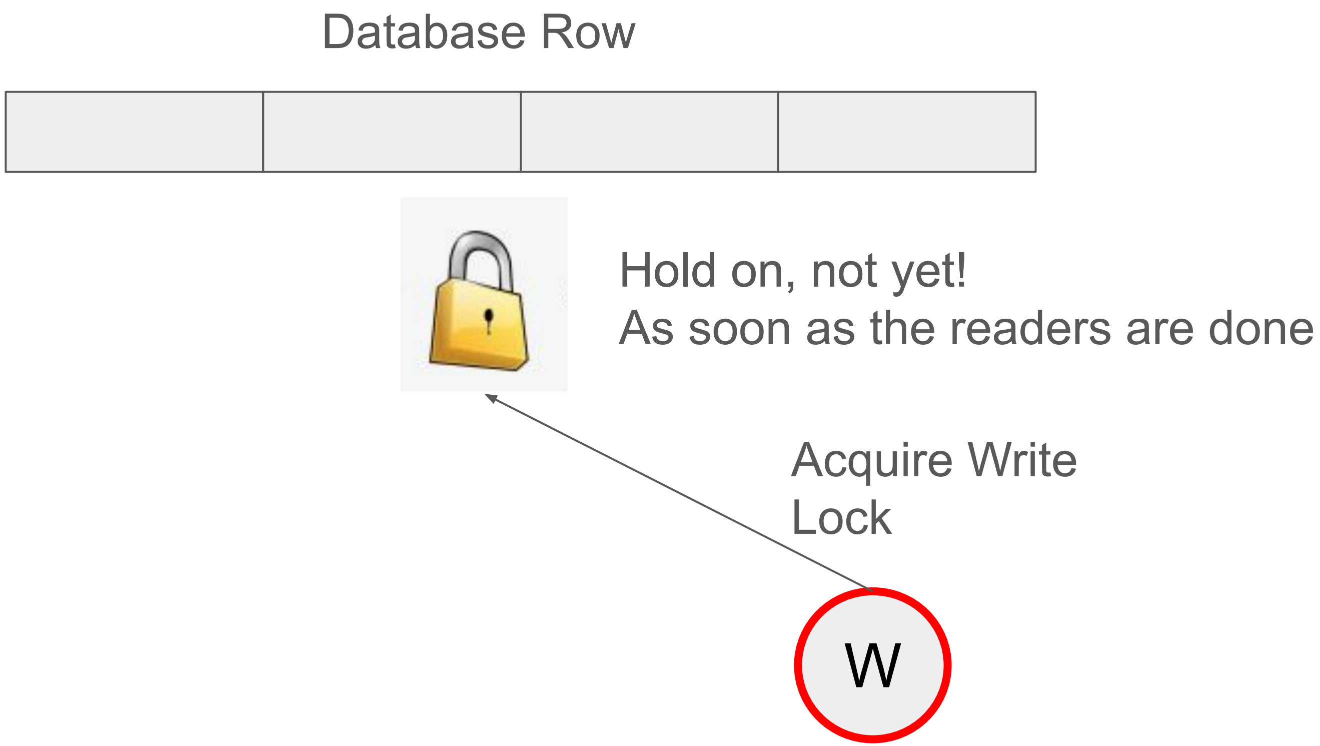

Database Row
Acquire Write
Lock?
R
R
R
R
W

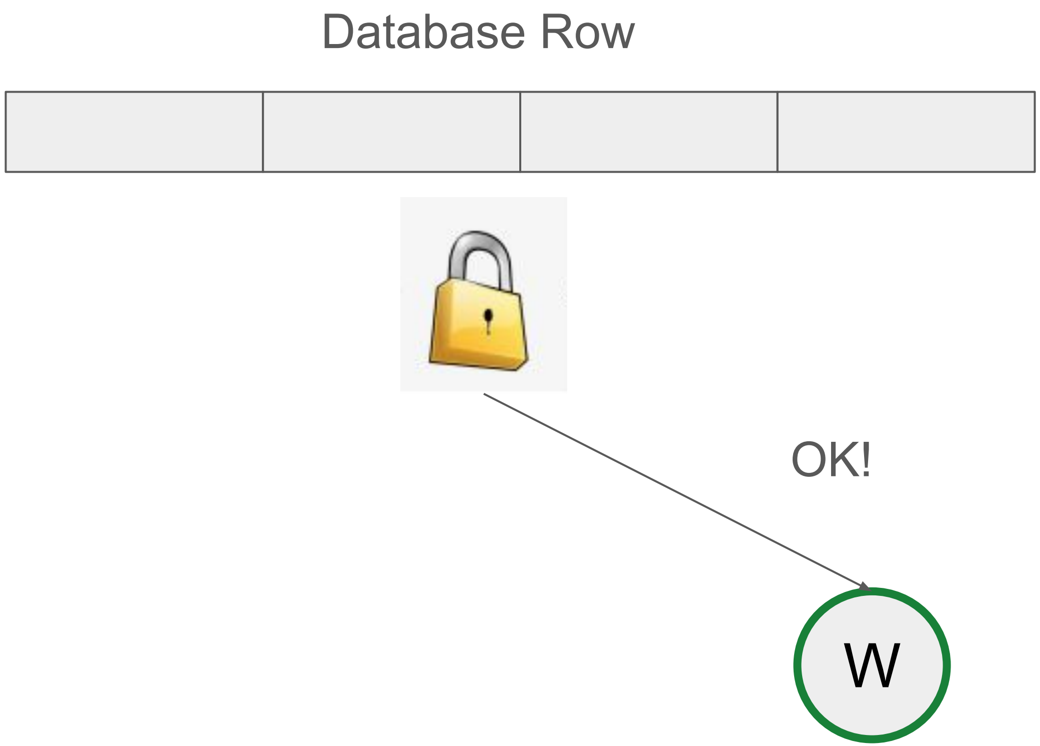

Database Row
Hold on, not yet!
As soon as the readers are done
Acquire Write
Lock
R
R
R
R
W

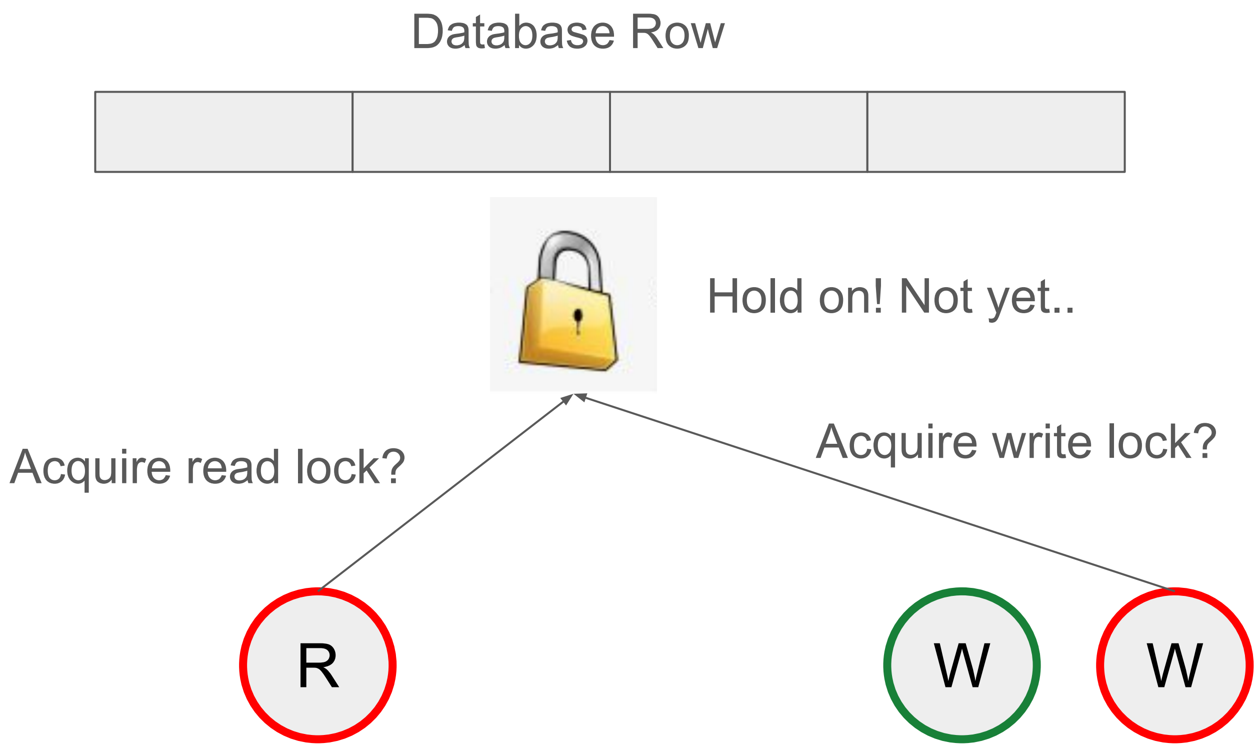


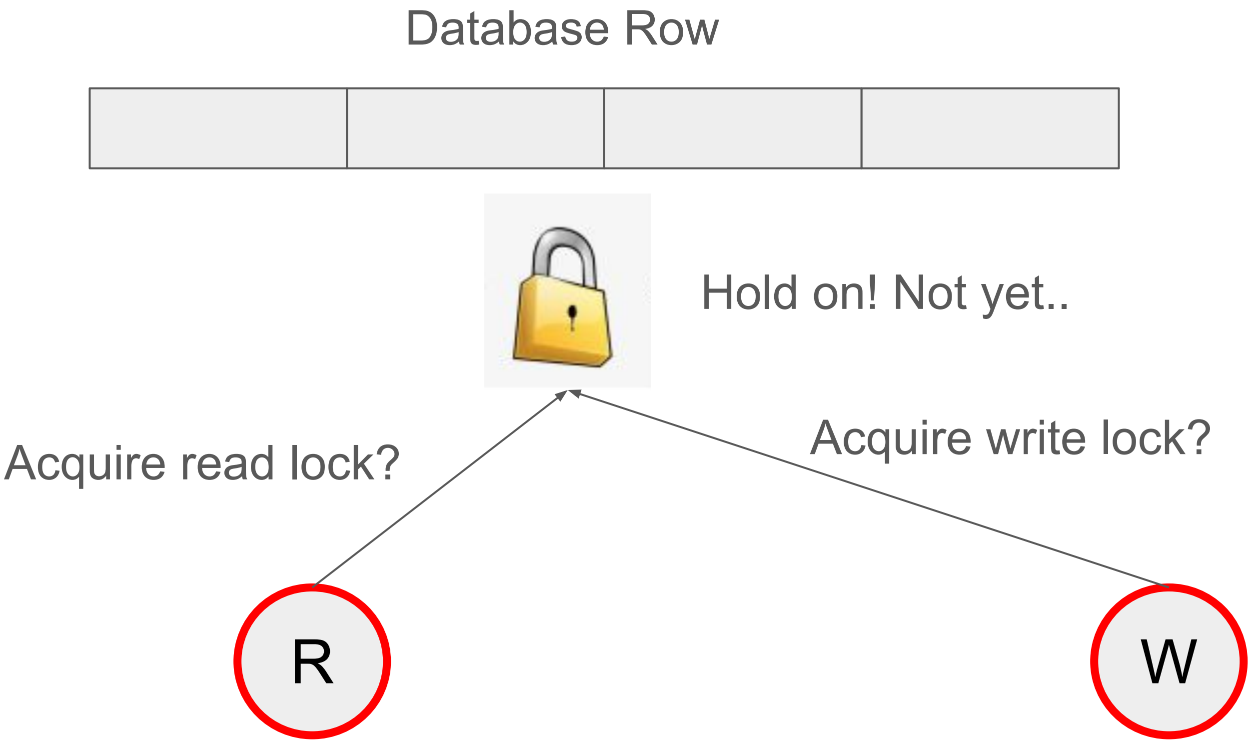
Database Row
Hold on, not yet!
As soon as the readers are done
I'm done!
Release
read lock
Acquire Write
Lock
R
R
R
R
W


Database Row
Hold on, not yet!
As soon as the readers are done
Release read lock
Acquire Write Lock
R
R
R
W

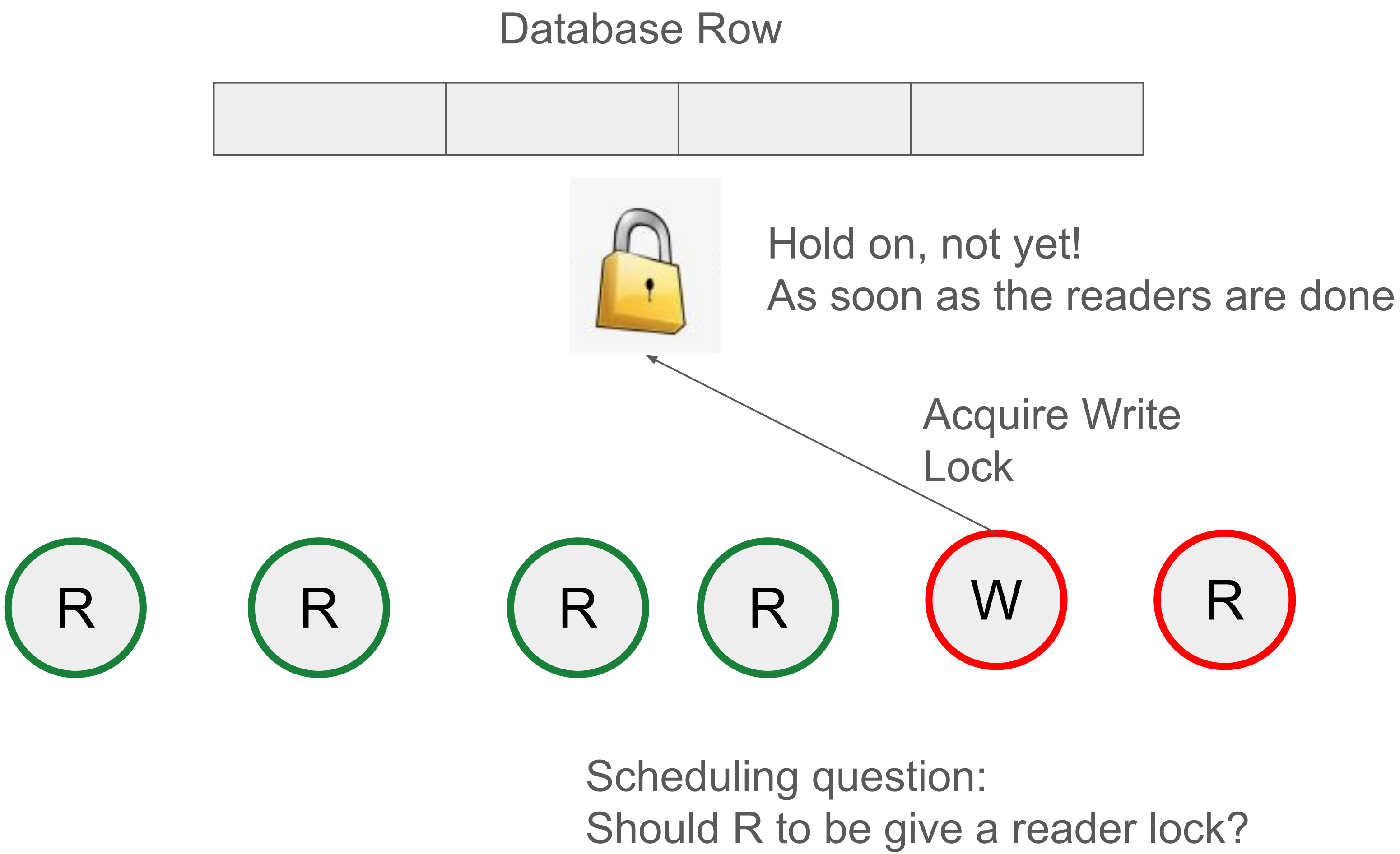

Database Row
Hold on, not yet!
As soon as the readers are done
Acquire Write
Lock
W


Database Row
OK!
W


Database Row
Hold on! Not yet..
Acquire read lock?
Acquire write lock?
R
W
W



Scheduling question: Who gets the lock now?


Database Row
Hold on, not yet!
As soon as the readers are done
Acquire Write
Lock
R
R
R
R
W
R
Scheduling question:
Should R to be give a reader lock?

Implementing Locks

Lock API

- Simple Lock
  - Acquire
  - Release
- ReadWrite Lock
  - AcquireReadLock
  - ReleaseReadLock
  - AcquireWriteLock
  - ReleaseWriteLock

Implementing Locks

```
type __sync_lock_test_and_set (type *ptr, type value, ...)

     Atomically set *ptr to value, return old value

void __sync_release (type *ptr)

     Atomically set *ptr to 0
```

```
void acquire_lock(int *lock) {
 while(__sync_test_and_set(&lock, 1));
}

void release_lock(int *lock) {
  __sync_release(&lock);
}
```

Project 1
Implement Reader/Writer Locks!

Project 1
Demo

Readers vs Writers

Atomics and synchronization primitives are not cheap!

- For readers, synchronization is an overhead
    - If there were only readers, you would not need synchronization
- For writers, synchronization is unavoidable

Lock implementation should aim to

- add minimal overhead to readers
- without giving up on correctness

Memory Consistency Model

```c
void incX(int *x, int times) {
  for (int i = 0; i < times; i++) {
    __atomic_add_fetch(x, 1, __ATOMIC_SEQ_CST);
  }
}
```

Intuition 2 - Operations are always performed in order

More unintuitive behaviour

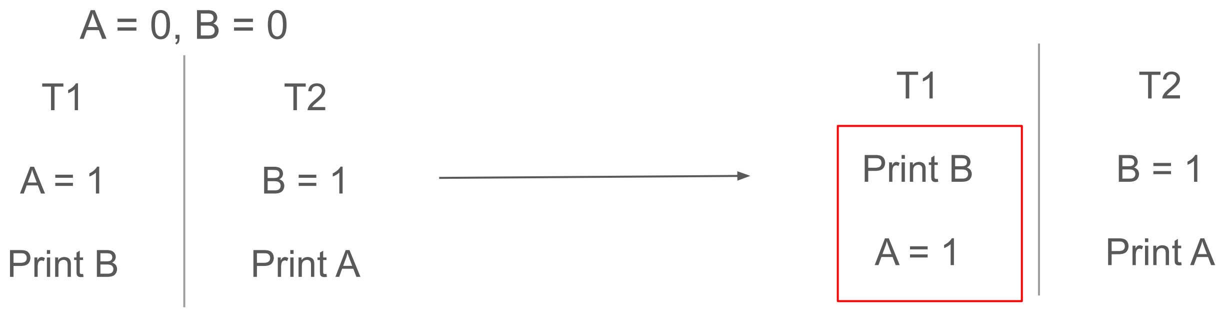
- Can the below code print (A=0,B=0) ?

A = 0, B = 0

T1	T2
A = 1	B = 1
Print B	Print A

More unintuitive behaviour

- Can the below code print (A=0,B=0) ?



CPU/Compiler thinks its ok to reorder independent statements!

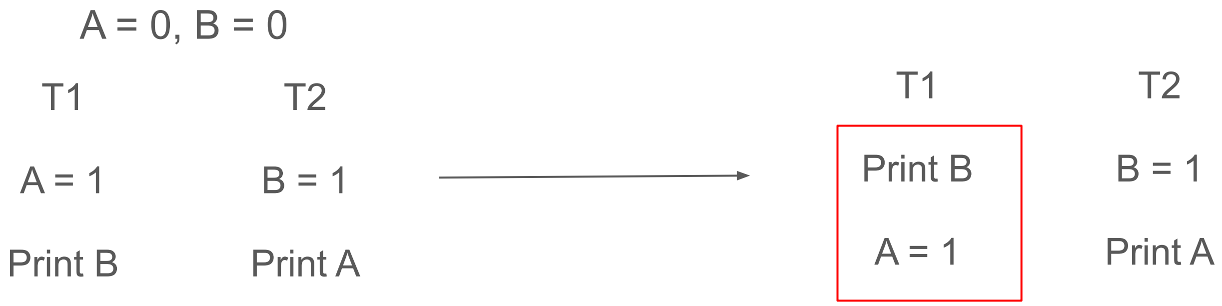
Memory Consistency Models

- Memory Consistency Models - expectations on memory behaviour
- Determines what reorderings are allowed
- Stricter consistency models at cost of performance
- Sequential Consistency
  - Interleavings must follow a order that could have been done on a single thread without breaking program order

```
void incX(int *x, int times) {
  for (int i = 0; i < times; i++) {
    __atomic_add_fetch(x, 1, __ATOMIC_SEQ_CST);
  }
}
```

Sequential Consistency

- 0, 0 not allowed in SC
- If 0,0 occurs -> one thread broke program order
- Acquire, Release, and Relaxed Semantics - allow more reorderings



Not allowed in SC