MULTICORE PROGRAMMING

Versioned locks, snapshots, version-lock-based KCAS

Lecture 15

Trevor Brown

LAST TIME

- When does TLE perform well? Poorly?
- More sophisticated uses of hardware transactional memory (HTM)
  - Concept: algorithms that **allow** fast-path transactions to run concurrently with fallback-path operations
  - **Accelerating** lock-free KCAS

THIS TIME

- Try-locks
  - Implementing try-locks
  - Try-lock set (a simple engineering trick for **easily** acquiring/releasing multiple locks)
  - Naïve (incorrect) KCAS using try-locks
- **Versioned** (try-)locks
  - Implementing versioned locks
  - KCAS using versioned locks

TRY-LOCKS

- Traditional lock operations
  - lock(): blocks until lock is acquired – may take a LONG time
  - unlock(): release the lock (usually a small number of instructions w/o blocking)
- Try-lock
  - tryLock(): either acquires the lock and returns true,
    or does not and returns false
  - unlock(): release the lock
- tryLock and unlock should both be wait-free
  (but algorithms that **use** them are not; those algorithms are lock-based)

WHY DO WE CARE ABOUT TRY-LOCKS?

- Two-phased locking with try-locks
  - Try to lock all data
  - If a tryLock returns false, release all locks and try again
  - Make changes
  - Release all locks
- Deadlock is **impossible** in this algorithm,
  even if you do not lock addresses in any consistent order
- Livelock is possible, but can be extremely unlikely

IMPLEMENTING A TRY-LOCK

```
class Lock {
    atomic<bool> lock;
public:
    Lock() {
        atomic_init(&lock, false);
    }
    bool isLocked() {
        return lock;
    }
    bool tryLock() {
        bool expected = lock.load(memory_order_acquire);
        if (expected) return false;
        return lock.compare_exchange_strong(expected, true);
    }
    void unlock() {
        lock.store(false, memory_order_release);
    }
};
```

Fail fast (try to avoid CAS when contended) Why?

A write is enough. No CAS needed. Why?

TRY LOCK SET: A SIMPLE ABSTRACTION FOR TAKING MULTIPLE TRY LOCKS

```
template <int MAX_K>
class TryLockSet { // contains info on locks we have locked
  Lock * locks[MAX_K];
  int k; // number of locks acquired
public:
  TryLockSet() { k = 0; }
  ~TryLockSet() {
    for (int i=0;i<k;++i) locks[i]->unlock();
  }
  bool tryLock(Lock * lock) {
    if (lock->tryLock()) {
      locks[k++] = lock;
      return true;
    }
    return false;
  }
}
```

Supply an upper bound on k via a template parameter

Just some simple software engineering

Locks we currently hold

Unlock all of our locks

Destructor: code executed when this object goes out of scope

Try to acquire lock, and add it to **locks** if successful

TRY-LOCK BASED KCAS: WHERE ARE THE LOCKS STORED?

- **Option 1: alongside program data**
- For each program **value,** place a **lock next to it** in memory

data	lock	data	lock	data	lock
data	lock	data	lock	data	lock

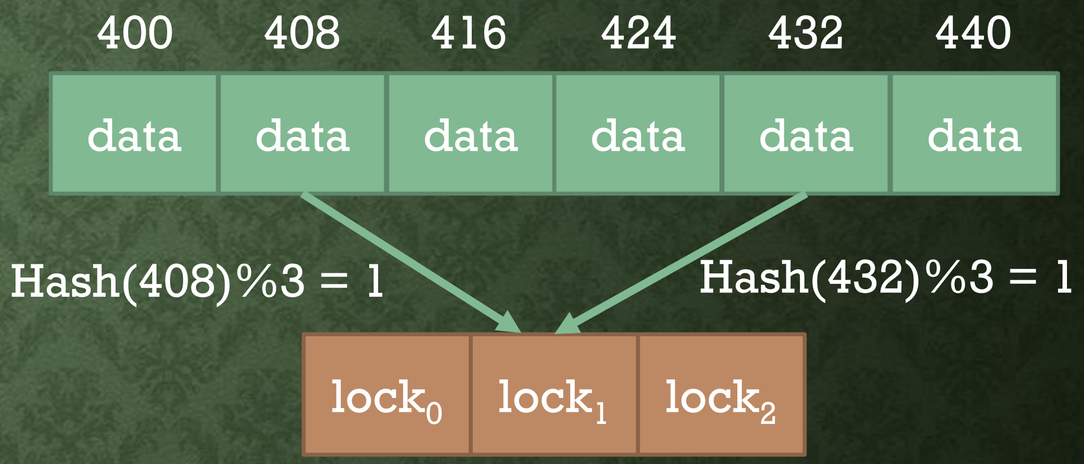
- Upsides: simple, good locality (locks usually in same cache lines as data)
- Downsides:
  - Requires program memory layout changes
  - Can **double** memory requirements

```
class addr_t {
private:
  Lock lock;
public:
  word_t value;
  Lock * addrOfLock() { return &lock; }
};
```

Each variable that can be modified by KCAS becomes an instance of addr_t

TRY-LOCK BASED KCAS: WHERE ARE THE LOCKS STORED?

- **Option 2: in a dedicated <u>lock table</u>**
  - **Don't** give each address a unique lock
  - Have each lock protect a **<u>set</u>** of addresses
  - How to map an address to "its" lock?
    - **<u>Hash addresses</u>** to obtain a **lock ID**
    - lockID = hash(addr) % numLocks



What if I want to lock **both** 432 and 408?
Careful about acquiring the same lock **twice**!

Lock() should be re-entrant. (For example, the lock can store the ID of the thread currently holding it, and you can check if you've already locked it, so you don't try to lock again.)

WHY WOULD YOU WANT TO USE A LOCK TABLE?

- If you can't (or don't want to) change the program's memory layout
- If you want to save space in data structure nodes (and the cache) by using fewer locks
- Maybe you can even use the same lock table across many data structures? (if the table is large enough)
- Practical consideration: false sharing on locks in the table? (they are next to each other, after all…)
    - Padding is probably a bad idea (can try and see…)
    - If number of locks is huge, relative to # of threads, expected contention should be low…

IMPLEMENTING A LOCK TABLE

How many locks do we need to get good scalability?

e.g., 1 million

Should be **much larger** than (max # threads) * (max # addresses written by KCAS)

```
// put padding before
Lock lockTable[LOCKTAB_SZ];
// put padding after

class addr_t {
public:
    word_t value;
    Lock * addrOfLock() {
        int idx = hash(&value) % LOCKTAB_SZ; // hash the address of value
        return &lockTable[idx];
    }
};
```

Small optimization: if the lock table size is a power of 2, then **h & (LOCKTAB_SZ-1)** is exactly the same as **h % LOCKTAB_SZ,** but bitwise-& can be cheaper than modulo

Can use either option for our try-lock based KCAS… Option 2 is cleaner, IMO…

NAÏVE TRY-LOCK BASED KCAS

Supply K via a template parameter

Array of **addr_t ***, each having **a lock and a value**

Try to acquire all locks

Write, unlock, return

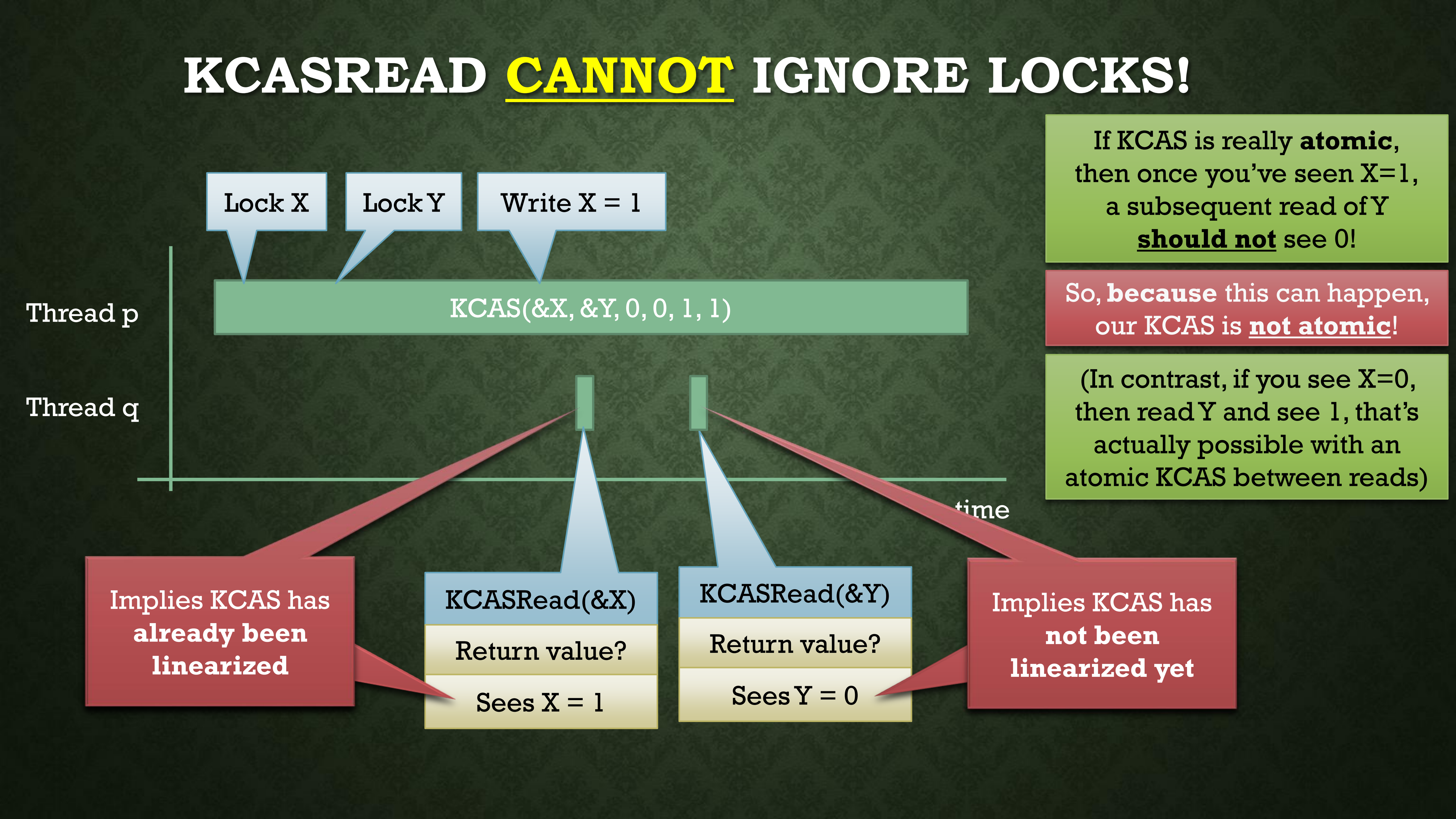
```
template <int K>
bool KCAS_locks(addr_t ** addr, val_t * expv, val_t * newv) {
  retry:
  TryLockSet<K> tls;          // auto-unlock when this goes out of scope
  for (int i=0;i<K;++i) {     // try to acquire locks
    if (!tls.tryLock(addr[i]->addrOfLock()))
      goto retry;             // (release locks and) retry
    if (addr[i]->value != expv[i])
      return false;           // (release locks and) fail
  }
  for (int i=0;i<K;++i) *addr[i] = newv[i];
  return true;                // (release locks and) succeed
}
```

Is it OK if KCASRead **ignores locks**?

```
val_t KCASRead(addr_t * addr) {
  return addr->value;
}
```

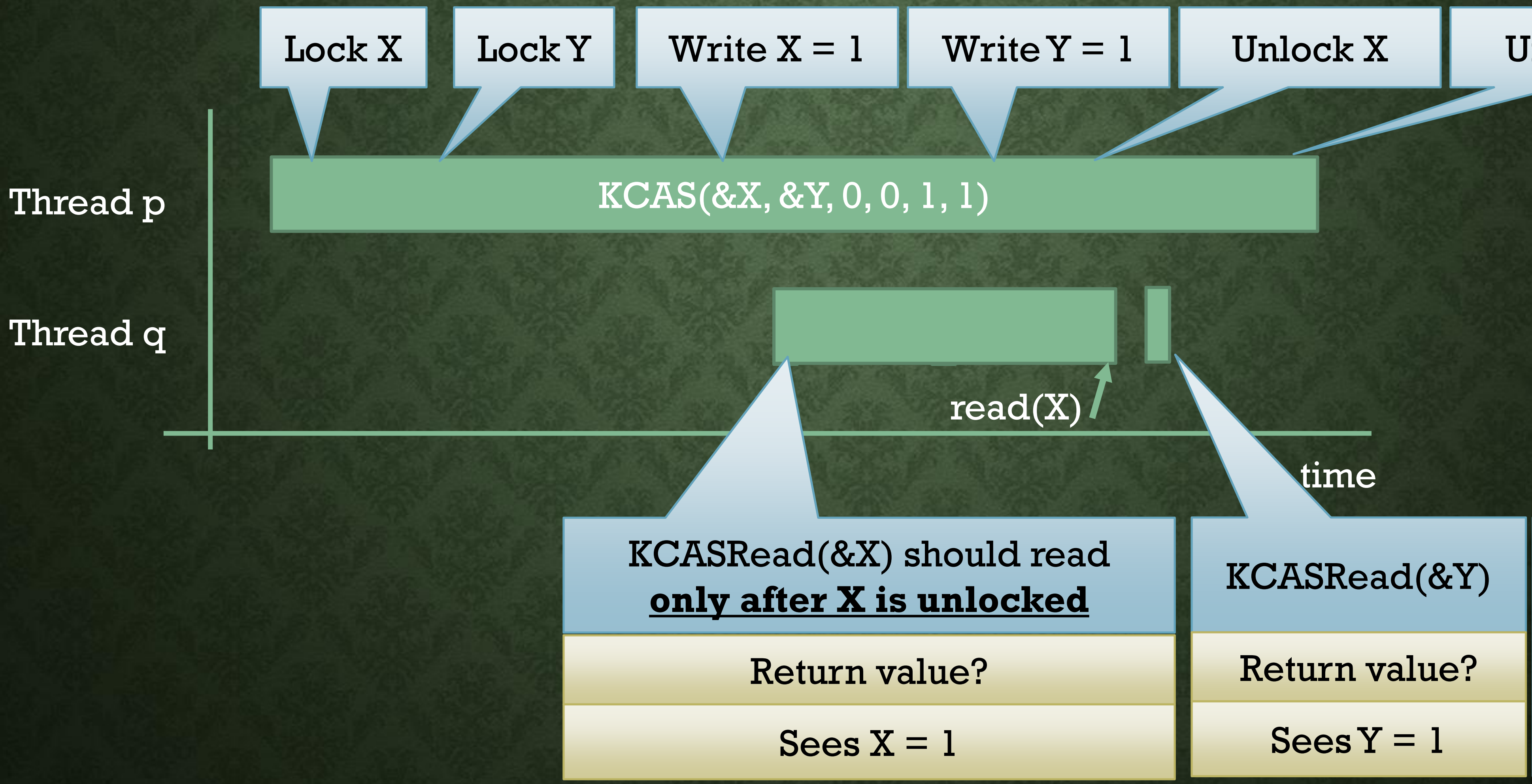
Where is KCASRead linearized?

Only **one step** to linearize at!



KCASREAD CANNOT IGNORE LOCKS!
Lock X
Lock Y
Write X = 1
Thread p
KCAS(&X, &Y, 0, 0, 1, 1)
Thread q
time
Implies KCAS has already been linearized
KCASRead(&X)
Return value?
Sees X = 1
KCASRead(&Y)
Return value?
Sees Y = 0
Implies KCAS has not been linearized yet
If KCAS is really atomic, then once you've seen X=1, a subsequent read of Y should not see 0!
So, because this can happen, our KCAS is not atomic!
(In contrast, if you see X=0, then read Y and see 1, that's actually possible with an atomic KCAS between reads)

HOW CAN WE FIX THIS PROBLEM? FIRST ATTEMPT:



This KCAS can be linearized at any time when X and Y are both locked!

ANOTHER ATTEMPT AT KCASREAD

```
val_t KCASRead(addr_t * addr) {
  while (true) {
    if (addr->addrOfLock()->isLocked()) continue;
    return addr->value;
  }
}
```

Wait until addr is unlocked

Then read

Correct but slow! (Why slow?)

How about **locking** addr, reading, and unlocking?

Need a **tool** to help us find a single time when **addr contained V** **<u>and</u> was unlocked**

No, addr might be locked again by the time we read addr->value!

- Consider an invocation of KCASRead that returns **V**
- Want to linearize at a time **t** during the KCASRead when:
  - addr contained V, **<u>and</u>**
  - addr is not locked
- Can we find such a time to linearize KCASRead?

VERSIONED TRY-LOCKS

- Like a try-lock, but also has a **version number** associated with it
- Version number = # of times lock was acquired
- Represented as an integer
  - Least significant bit = lock-bit (is it currently locked)
  - Other bits = version number
  - "Unlocked, after being acquired 4819 times" `<4819, 0> = (4819 << 1) | 0`
  - "Locked, after being acquired 17 times" `<17, 1> = (17 << 1) | 1`
  - Checking if a lock is held: `if (lock & 1) { … }`
  - Getting the version number from a lock: `ver = (lock >> 1);`
- Offers operations: unlock(), tryLock() and read()

IMPLEMENTING VERSIONED TRY-LOCKS

```
class VLock {
private:
    atomic<uint64_t> lock; // <version number, lock_bit>
public:
    uint64_t read() {
        return lock;
    }
    bool tryLock() {
        uint64_t exp = lock.load(memory_order_acquire);
        if (exp & 1) return false;
        return lock.compare_exchange_strong(exp, exp|1);
    }
    void unlock() {
        uint64_t old = lock.load(memory_order_relaxed);
        lock.store(old+1, memory_order_release);
    }
};
```

If lock is held

Try to acquire

Since the lock bit (LSB) is 1, incrementing **lock** changes the LSB to 0 (unlocked), **and** increments the **version #**

Note: "relaxed" is strong enough here because of the CAS before and data dependency after…

CORRECT KCASREAD

```
val_t KCASRead(addr_t * addr) {
  Lock * l = addr->addrOfLock();
  while (true) {
    uint64_t s1 = l->read();
    if ((s1 & 1) == 0) {
      casword_t v = addr->value;
      uint64_t s2 = l->read();
      if (s2 == s1) return v;
    }
  }
}
```

Read lock state (i.e., <version_number, lock_bit>

If it is unlocked

Read value

Read lock state **again**

If **still** unlocked, **and same** version number as before

Then addr was not locked **at any time** between the two read() operations.

So when we read addr->value, addr is **not locked!**

The rest of the code is the same as the naïve try-lock based KCAS, but with **versioned try-locks** instead of try-locks.

WHAT ELSE CAN WE DO WITH THESE VERSION NUMBERS?

- E.g., can implement a **snapshot** operation, which **atomically** performs **many** reads (all at

```
vector<word_t> snapshot(addr_t ** addr, int size) {
retry:
    vector<uint64_t> lockStates;
    vector<word_t> values;
    for (int i=0;i<size;++i) {
        uint64_t state = addr[i]->addrOfLock()->read();
        if (state & 1) goto retry;
        lockStates.push_back(state);
        values.push_back(addr[i]->value);
    }
    for (int i=0;i<size;++i) {
        uint64_t state = addr[i]->addrOfLock()->read();
        if (state != lockStates[i]) goto retry;
    }
    return values;
```

For each addr, read the lock state (incl. version number)

Retry if we see a lock is held

Locally save the lock state we saw

Then read the **value** guarded by the lock (crucially, **after** the lock state)

Reread all lock states and check that all locks are still released, and version numbers are the same as we saw above (else retry)

If we get past this loop, then no **value** has changed since we read it!

Example of the famous "**double collect**" paradigm

This is called **version/sequence-based validation**

Can linearize the entire snapshot between loops