MULTICORE PROGRAMMING

(Dis)ordered data structures

Lecture 3

Trevor Brown

ANNOUNCEMENTS

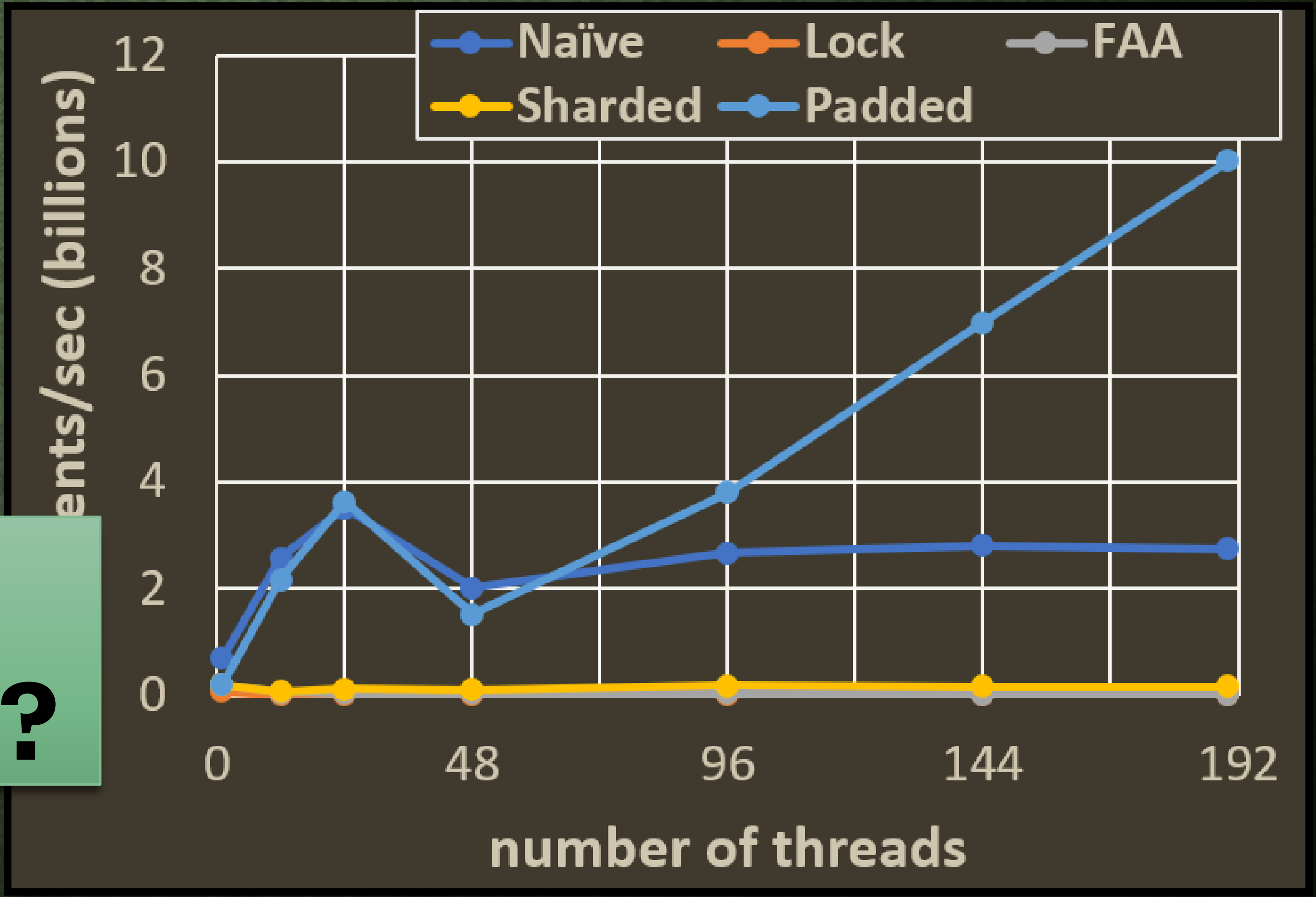
- Reminder: A1 due soon
- A2 to be released soon!

MODELING PERFORMANCE ON A REAL SYSTEM

- Processor caches largely determine performance
  - Last-level cache is most important
  - (Memory is 10-100x slower)
- Cache lines accessed by only one thread
  - Cheap to access (even exclusive mode creates **no contention**)
- Read-only cache lines accessed by many threads
  - Cheap to access (shared mode **allows concurrency**)
- Cache lines accessed by many threads **and often modified**
  - Expensive to access (exclusive mode **blocks** other threads)
  - Possibly susceptible to false sharing

RECALL: PADDED SHARDED COUNTER PERFORMANCE

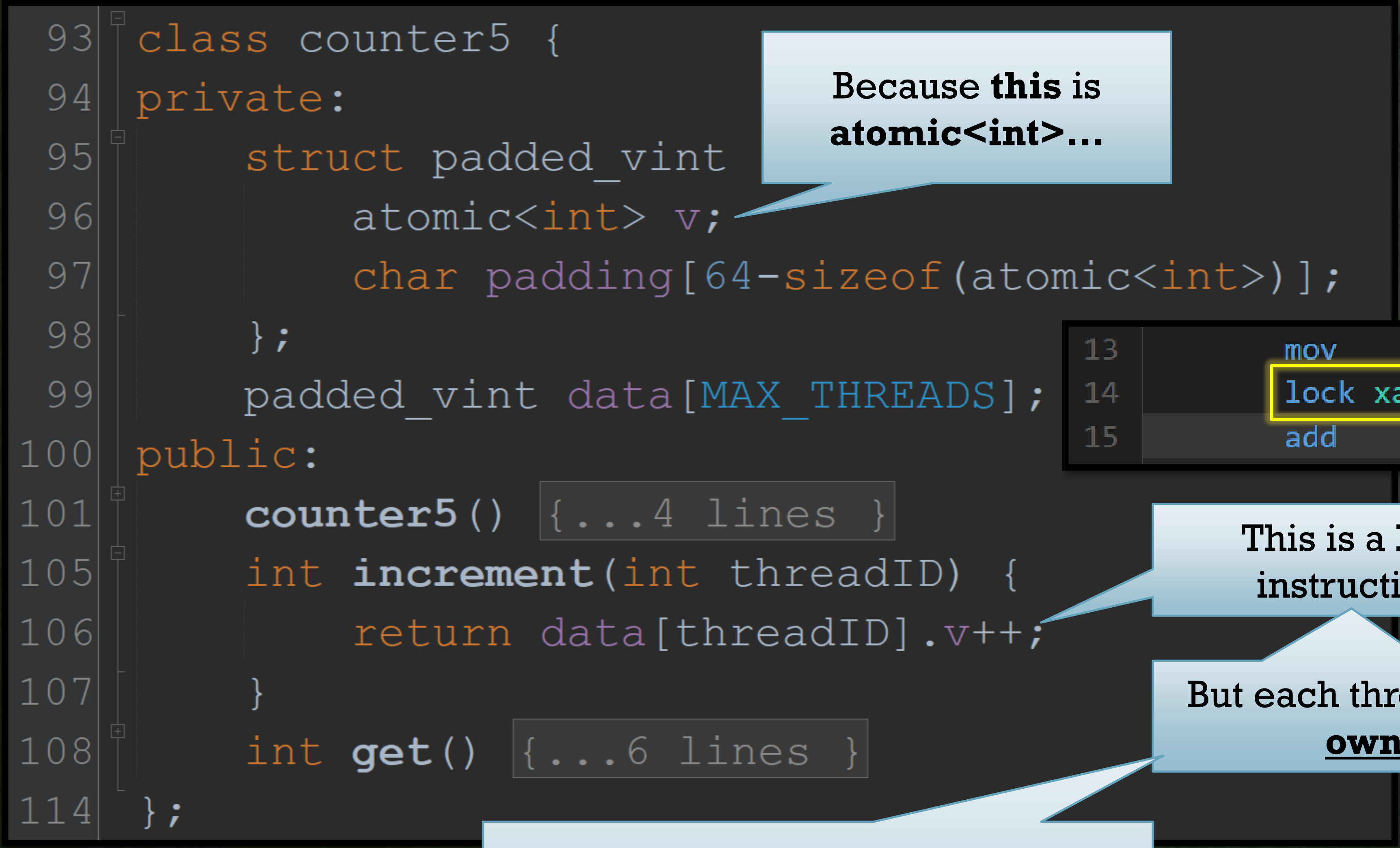
- **Timed** experiment comparing **naïve sharding** with a **padded counter**
- **Pretty good scaling**



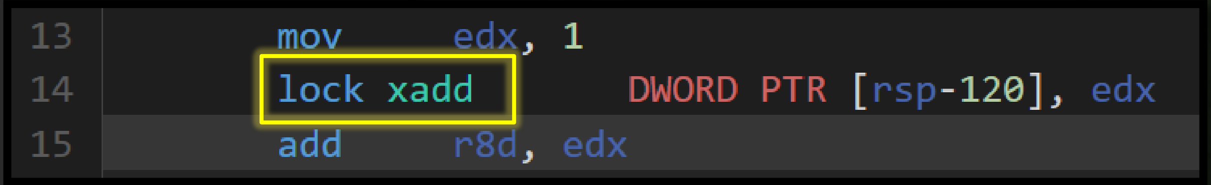
Can we make this even faster?

ENTERING THE DANGER ZONE

To learn about instruction reordering and weak memory models…



```
class counter5 {
private:
    struct padded_vint
        atomic<int> v;
        char padding[64-sizeof(atomic<int>)];
    };
    padded_vint data[MAX_THREADS];
public:
    counter5() {...4 lines }
    int increment(int threadID) {
        return data[threadID].v++;
    }
    int get() {...6 lines }
};
```



```
        mov         edx, 1
        lock xadd         DWORD PTR [rsp-120], edx
        add         r8d, edx
```

Because **this** is **atomic<int>…**

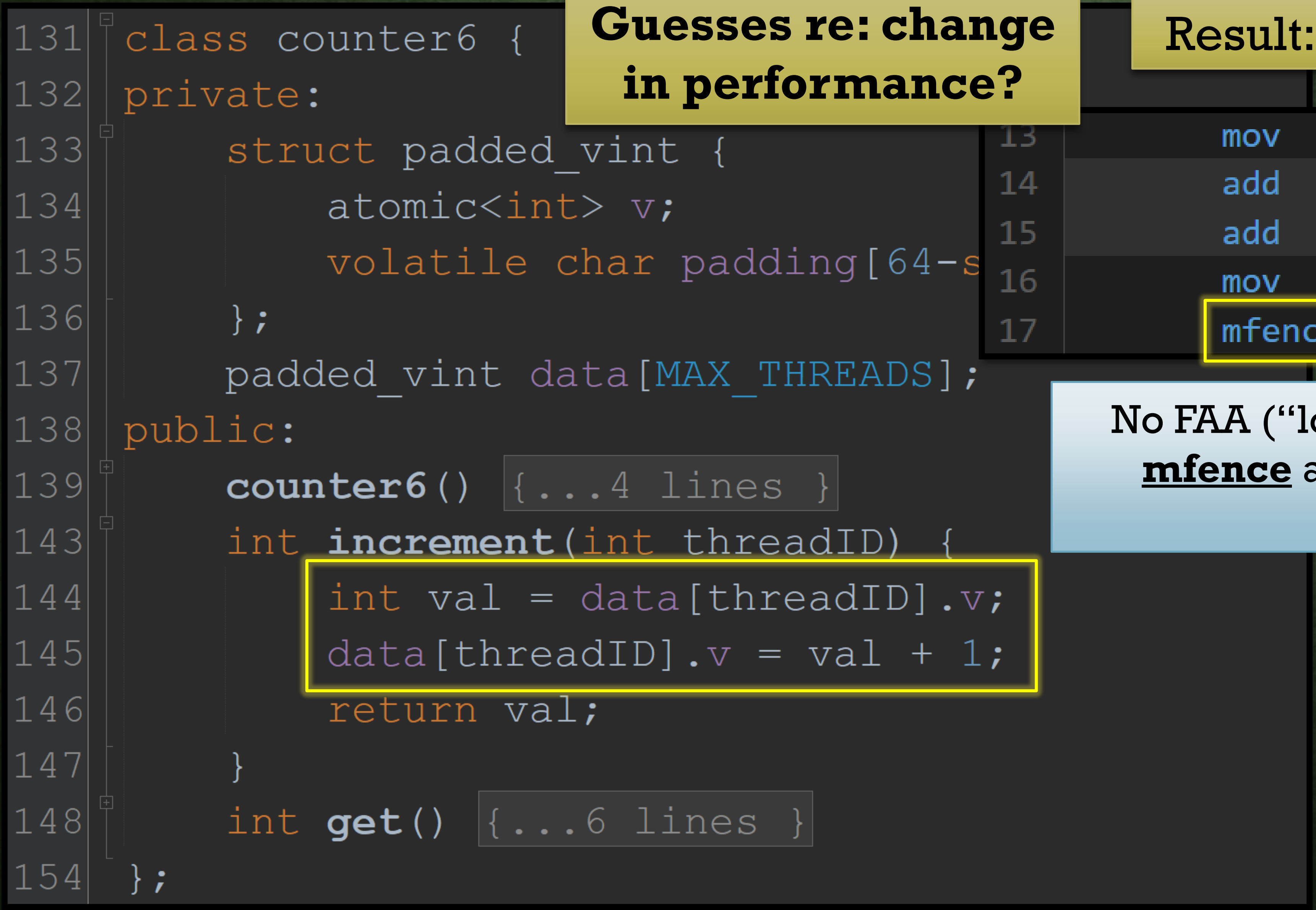
This is a **FAA** instruction!

But each thread only modifies **<u>its own</u>** sub counter!

So, FAA is not needed for increments to be **atomic relative to other *increments*!**

(We aren't worrying about *get* operations for now…)

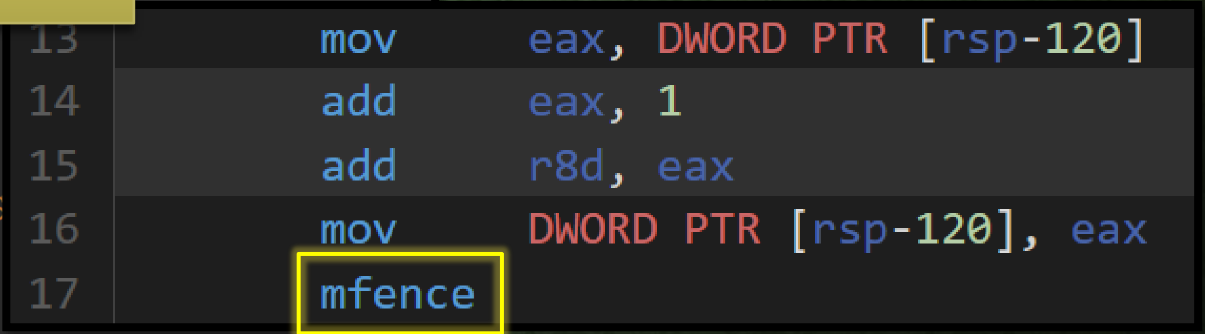
ATTEMPT 1: BREAK ++ INTO LOAD & STORE



```
class counter6 {
private:
    struct padded_vint {
        atomic<int> v;
        volatile char padding[64-s
    };
    padded_vint data[MAX_THREADS];
public:
    counter6() {...4 lines }
    int increment(int threadID) {
        int val = data[threadID].v;
        data[threadID].v = val + 1;
        return val;
    }
    int get() {...6 lines }
};
```

Guesses re: change in performance?

Result: **10x slowdown! Why?**



```
mov     eax, DWORD PTR [rsp-120]
add     eax, 1
add     r8d, eax
mov     DWORD PTR [rsp-120], eax
mfence
```

No FAA ("lock xadd"), but there is an **mfence** added by the compiler…
What is this?

INSTRUCTION REORDERING

- When you write code, are the lines of code executed in the order you write them?
- Compiler can reorder your code!
  (as long as doing so wouldn't break **sequential** code)
- Processor can **also** reorder your code
  - On modern Intel and AMD, the only permitted reordering is taking a read that is **after a write**, and moving it so it is now **before the write**
  - This is called "read before write" reordering (and it helps hide cache miss latency)
- **Both** compiler and processor reordering are **prevented** by using C++ atomics as shown

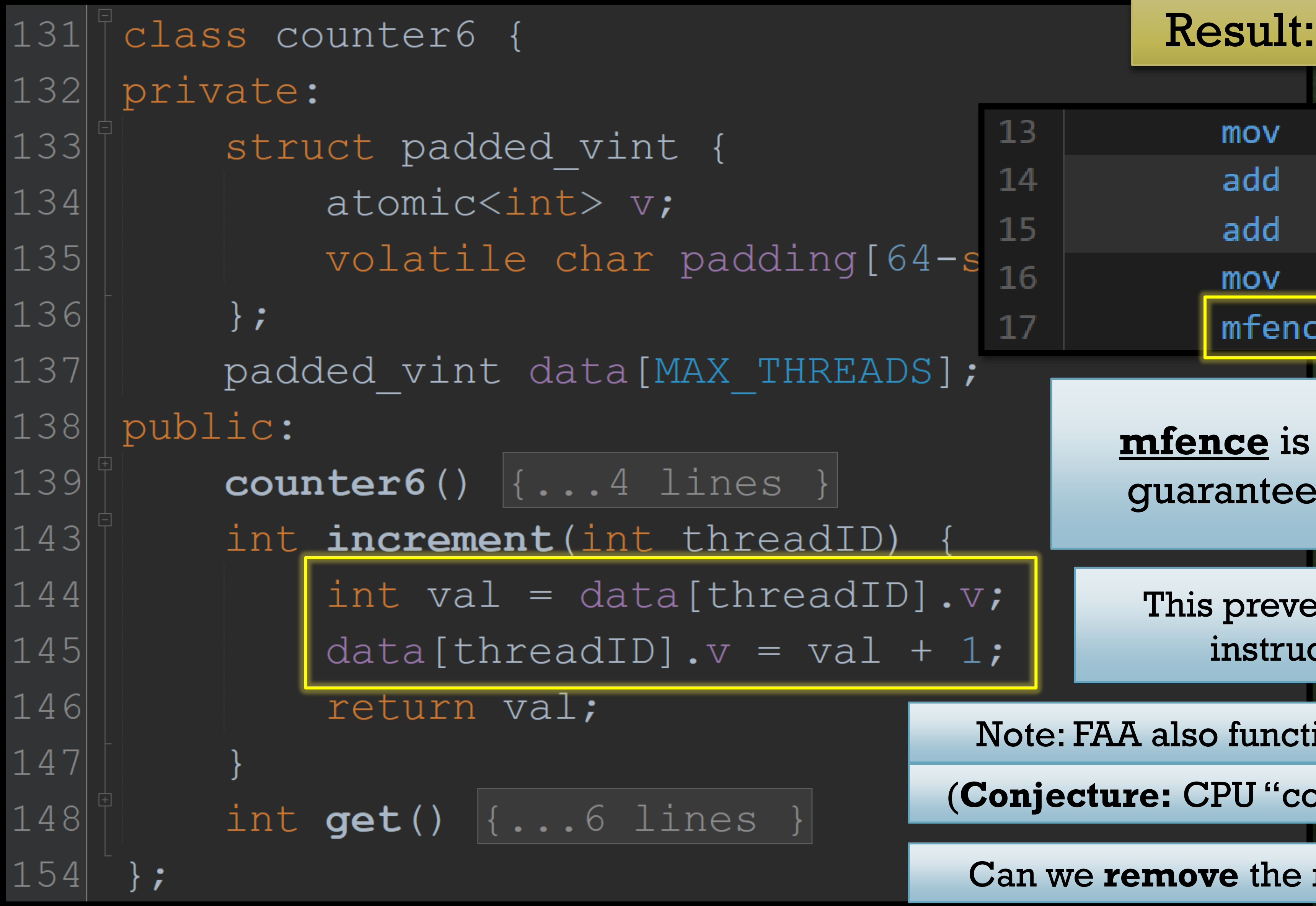
Before atomics, we used explicit **mfence** instructions to prevent this (`__sync_synchronize` in GCC)

Not a concern if you **always** lock objects before accessing them – locks are specifically designed prevent reordering!

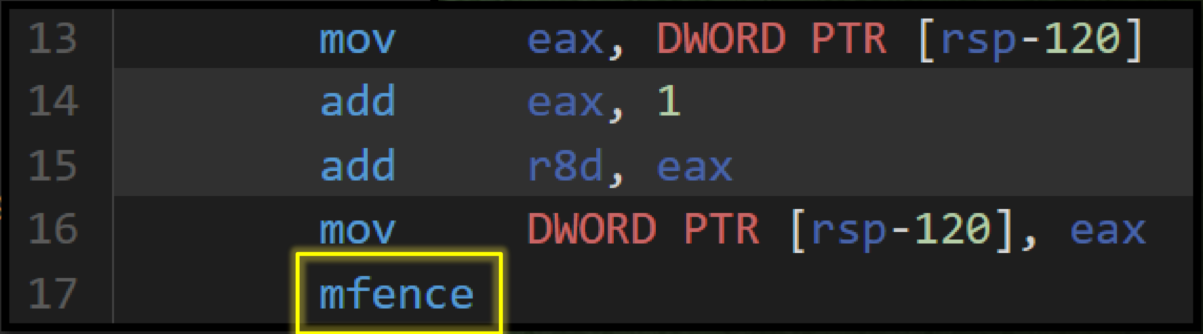
(In this case, **no need** to use atomic types for fields protected by locks!)

ATTEMPT 1: BREAK ++ INTO LOAD & STORE

Result: **10x slowdown! Why?**



```
class counter6 {
private:
    struct padded_vint {
        atomic<int> v;
        volatile char padding[64-s
    };
    padded_vint data[MAX_THREADS];
public:
    counter6() {...4 lines }
    int increment(int threadID) {
        int val = data[threadID].v;
        data[threadID].v = val + 1;
        return val;
    }
    int get() {...6 lines }
};
```



```
        mov     eax, DWORD PTR [rsp-120]
        add     eax, 1
        add     r8d, eax
        mov     DWORD PTR [rsp-120], eax
        mfence
```

mfence is added by the compiler to guarantee **sequential consistency**

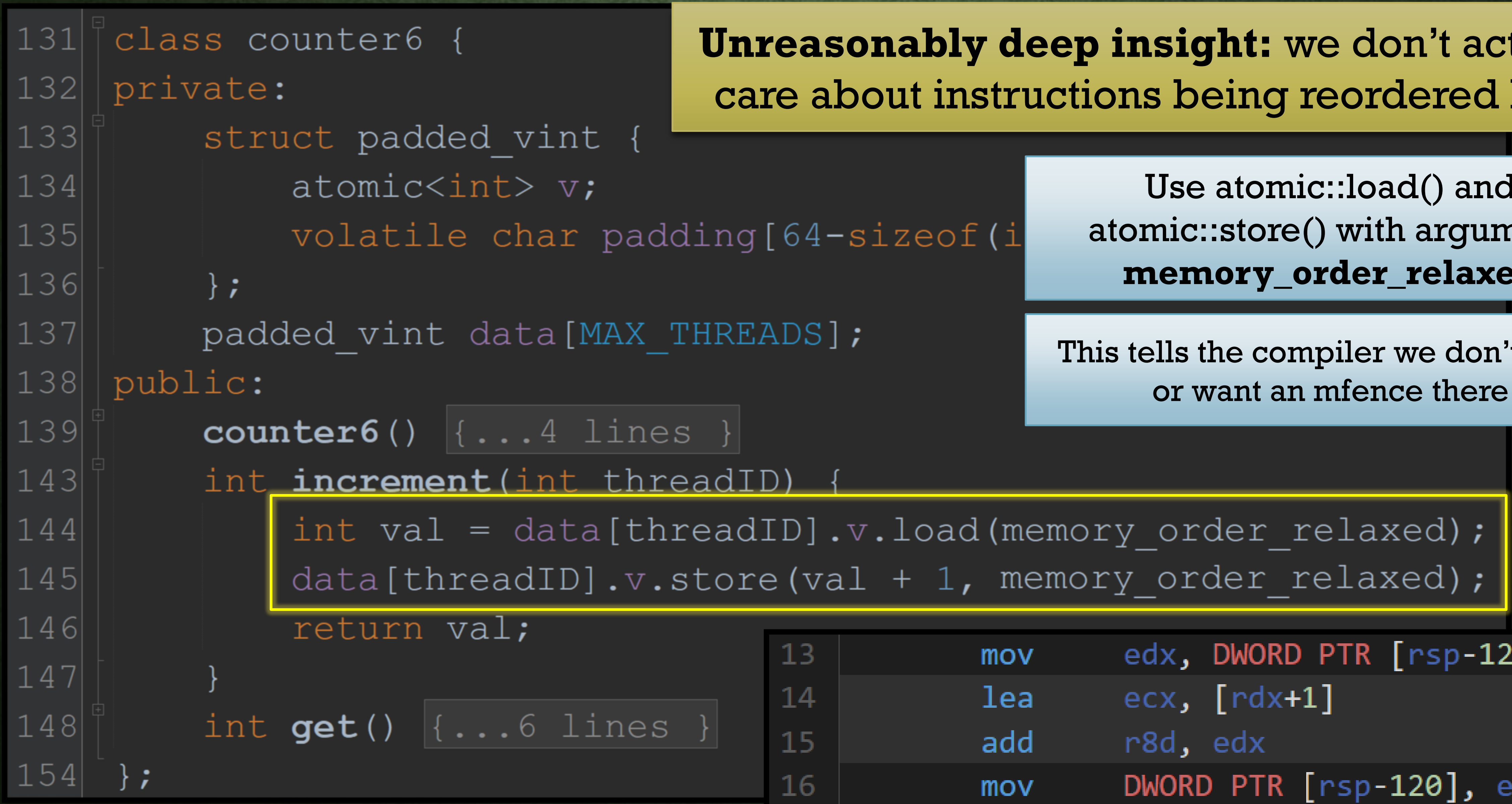
This prevents the CPU from reordering instructions, but is **very costly**

Note: FAA also functions like an mfence on x86/64

(**Conjecture:** CPU "combines" FAAs, unlike mfences)

Can we **remove** the mfence? Is it **correct** to do so?

ATTEMPT 2: EXPLICITLY ALLOW REORDERING

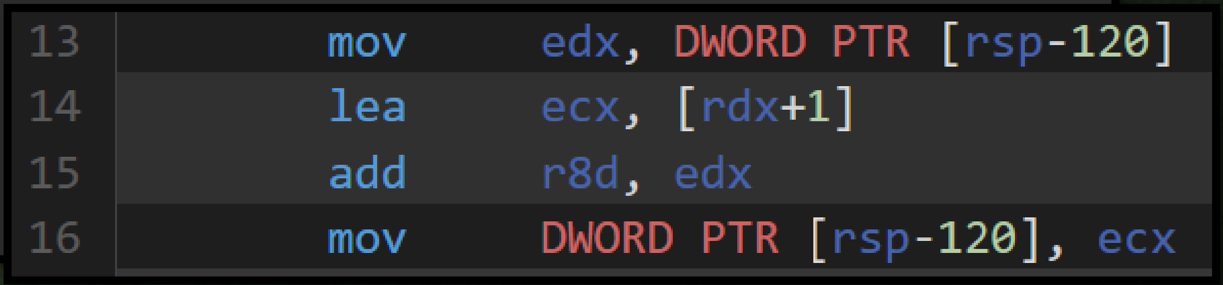


```
class counter6 {
private:
    struct padded_vint {
        atomic<int> v;
        volatile char padding[64-sizeof(i
    };
    padded_vint data[MAX_THREADS];
public:
    counter6() {...4 lines }
    int increment(int threadID) {
        int val = data[threadID].v.load(memory_order_relaxed);
        data[threadID].v.store(val + 1, memory_order_relaxed);
        return val;
    }
    int get() {...6 lines }
};
```

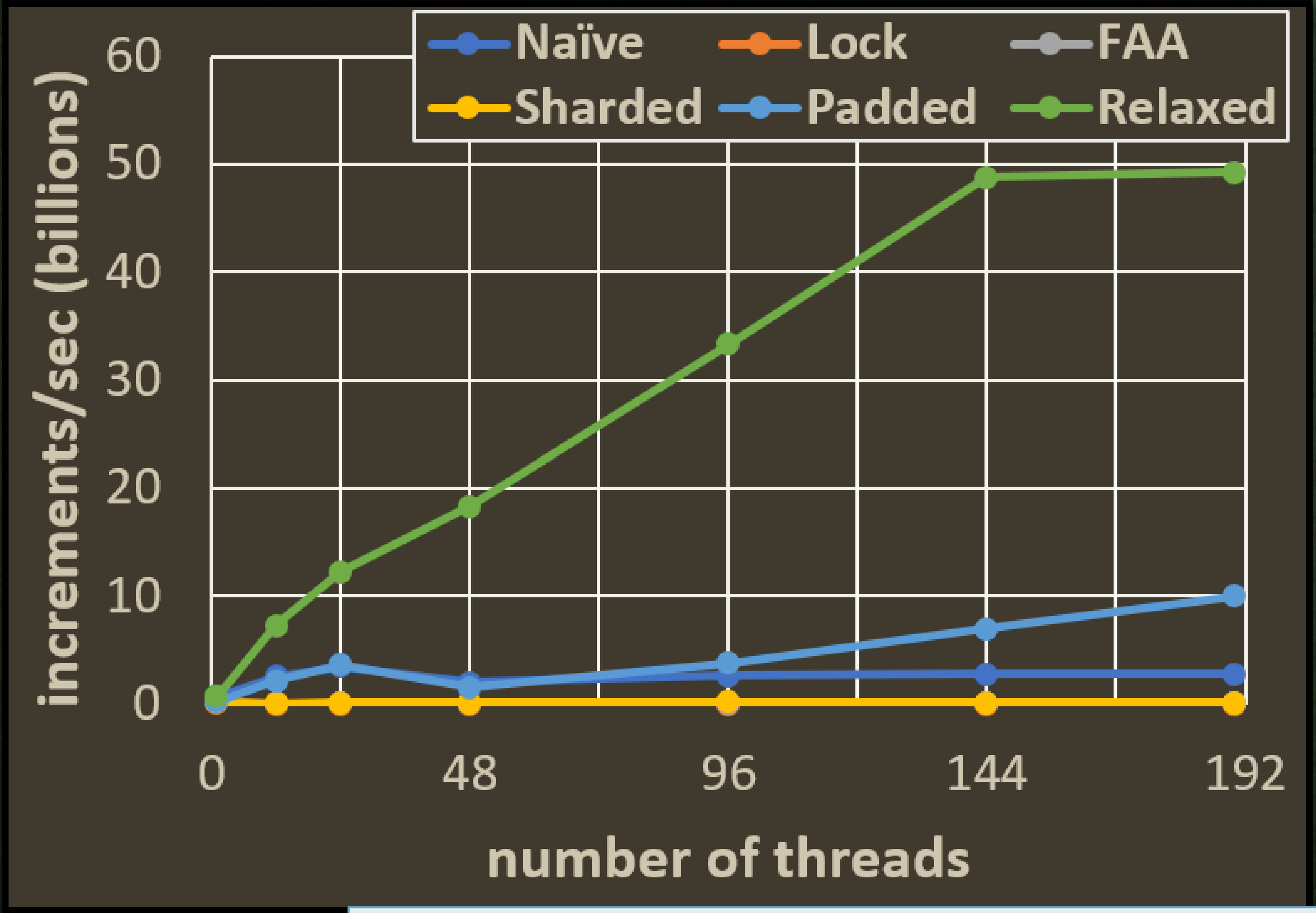
Unreasonably deep insight: we don't actually care about instructions being reordered here

Use atomic::load() and atomic::store() with argument **memory_order_relaxed**

This tells the compiler we don't need or want an mfence there



```
        mov     edx, DWORD PTR [rsp-120]
        lea     ecx, [rdx+1]
        add     r8d, edx
        mov     DWORD PTR [rsp-120], ecx
```



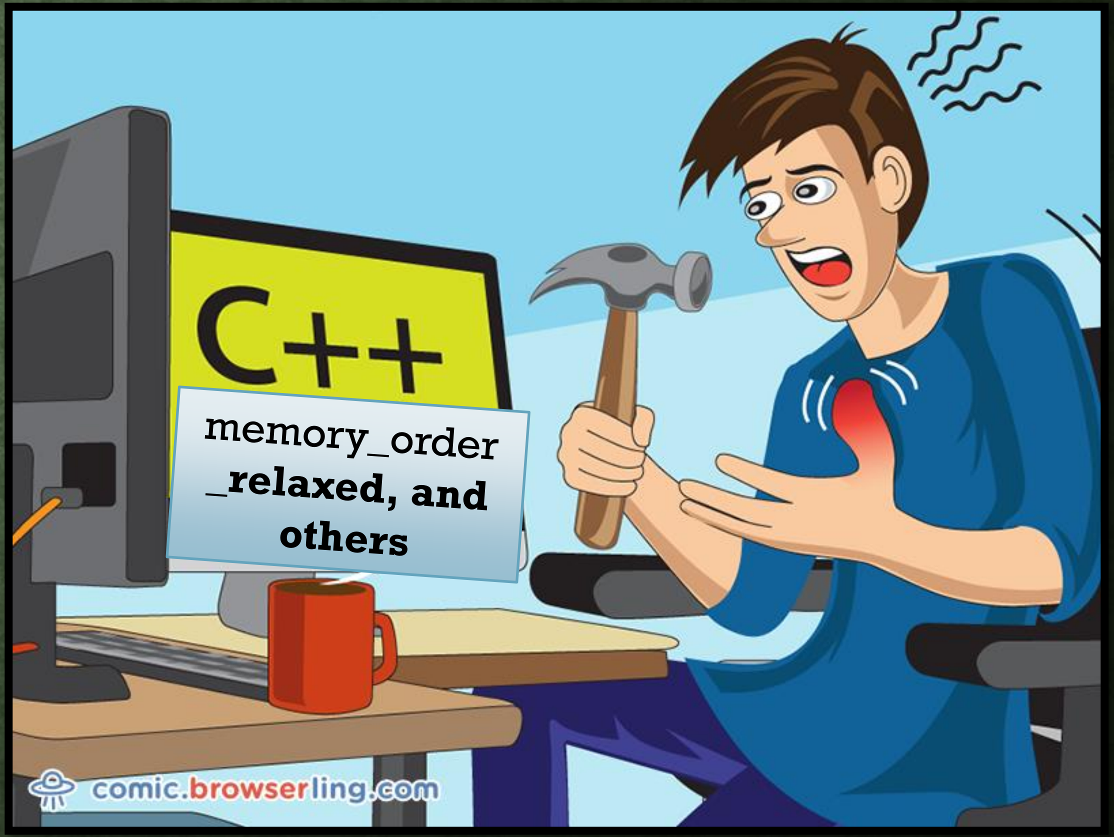
Takeaways: (1) fetch&add is much faster than a **write** followed by an **mfence**. (2) mfence is ***so slow*** that it's almost certainly faster to use an atomic exchange instruction (also called XCHG or SWAP) on x86/64 instead of a write+mfence.

HOW *COULD* YOU USE C++ ATOMICS

- atomic<int> x offers functions:
  - int r = x.**load**(memory_order)
  - x.**store**(new_value, memory_order)
- memory_order has **default value** of memory_order_seq_cst
  - If all accesses to shared variables use this, you get **sequential consistency** (as if there is **no** reordering)
- Other possible arguments:
  - memory_order_acquire and memory_order_release (used together)
    - If a thread p "acquires" data that another thread q "released", then p also sees the effects of anything else q did before releasing the data (difficult…)
  - memory_order_consume (poorly defined, probably deprecated, **ignore!**)
  - memory_order_relaxed (**all reordering allowed!**)

HOW ***SHOULD*** YOU USE C++ ATOMICS?

- **Prevent all reordering** by using the default memory_order_**seq_cst**
- C++14 specification 29.3, note 8:
  - "memory_order_seq_cst ensures sequential consistency only for a program that is free of data races and uses **exclusively** memory_order_seq_cst operations. **Any use of weaker ordering will invalidate this guarantee unless extreme care is used.**"
- When the spec. says "extreme care," you should be terrified


C++
memory_order
_relaxed, and
others
comic.browserling.com

ANOTHER APPROACH: **APPROXIMATE** OBJECTS

- Sometimes **approximate** results are enough for some applications!
  - Example: if a counter is used to decide when to **expand** a hash table (just to improve *performance --- no impact on hash table's correctness*)
- Could imagine an **approximate counter ADT**
  - Parameterized by an **error constant c**
- Abstract state is an integer, initially zero
- Operations
  - Increment: increases the abstract state by 1
  - ExactGet: returns the abstract state
  - Get: returns a value that is **within ±cn** of the abstract state, where n is the number of threads that access the counter

Just in case you want an exact answer

For typical use

AN **APPROXIMATE** COUNTER IMPLEMENTATION

- **Global** data: atomic<int64_t> globalCount
- **Per-thread** data: int64_t localCount
- Increment:
  - increments the thread's own **private** localCount
  - after the thread has done $c \cdot n$ increments,
    it does Fetch&Add(&globalCount, localCount)
    then sets its private localCount to 0

Reduces contention on size vs F&A every time

- How far off can globalCount be from the true value?
  - $Error = c \cdot n^2$
  - Insignificant once counter value is large
  - (For c=10, 100 threads, error is 100k.
    After just 10M increments, this is 1% error)

How fast is this approach?
You'll see in A2!

NEXT TOPIC

(Dis)ordered data structures

STACK OBJECT

- Operations
  - Push(key)
    - Pushes a key onto the stack
  - Pop()
    - Returns the last key pushed onto the stack, if the stack is not empty
    - Otherwise, returns **null**

NAÏVE STACK IN C++

Data types

```
struct node {
    const int key;
    atomic<node *> next;
    node(int _key, node * _next)
        : key(_key), next(_next) {}
};

struct stack {
    atomic<node *> top;
    stack() : top(NULL) {}
    void push(int key);
    int pop();
};
```

Operations

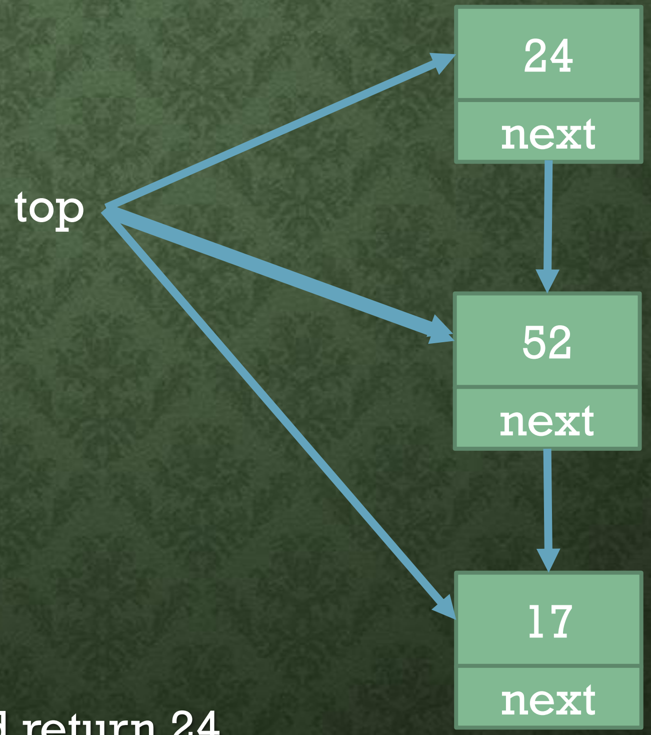
```
void stack::push(int key) {
    node * n = new node(key, top);
    top = n;
}

int stack::pop() {
    node * n = top;
    if (n == NULL) return EMPTY;
    top = n->next;
    return n->key;
}
```

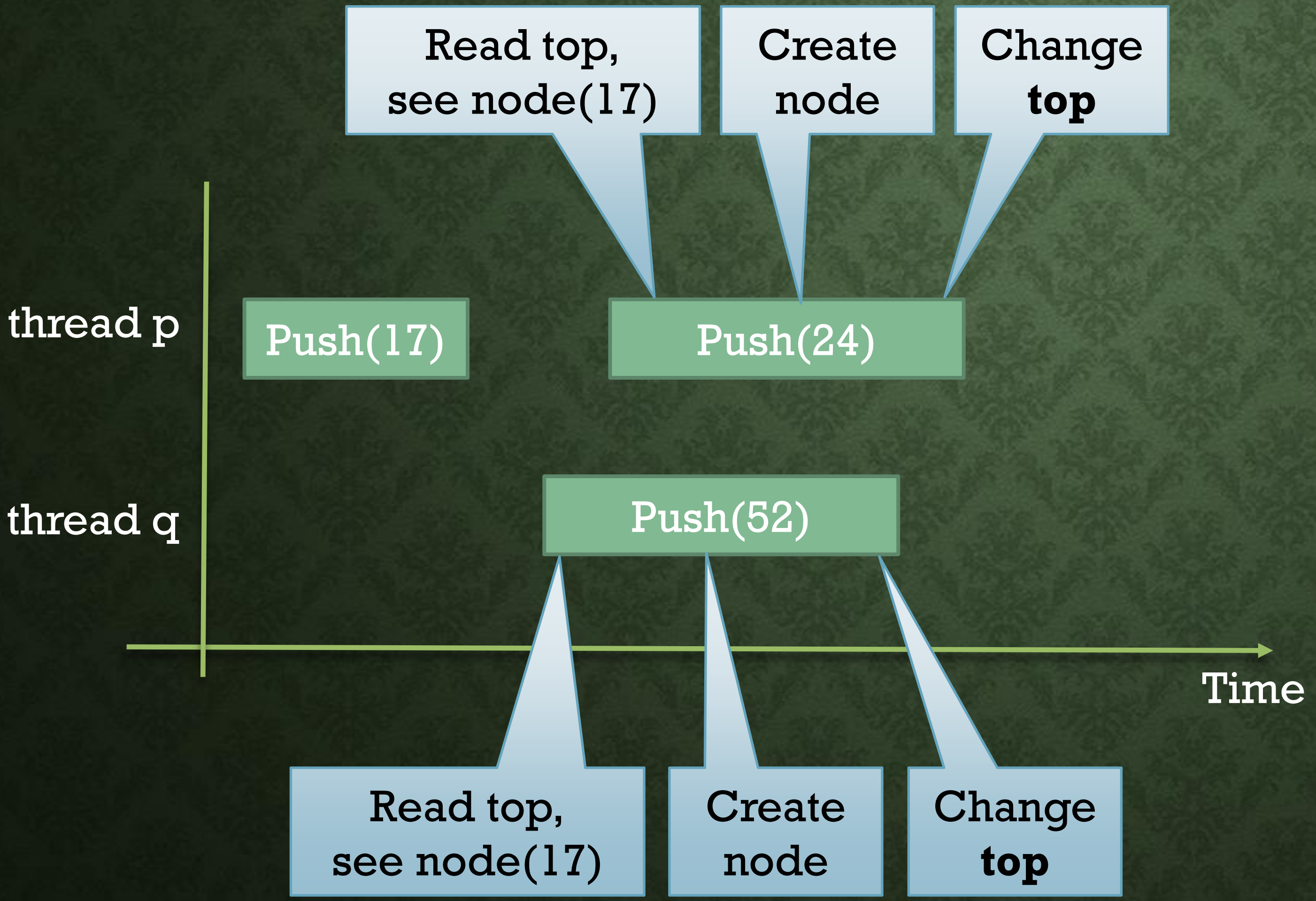
Pop() leaks memory…
more on this later…

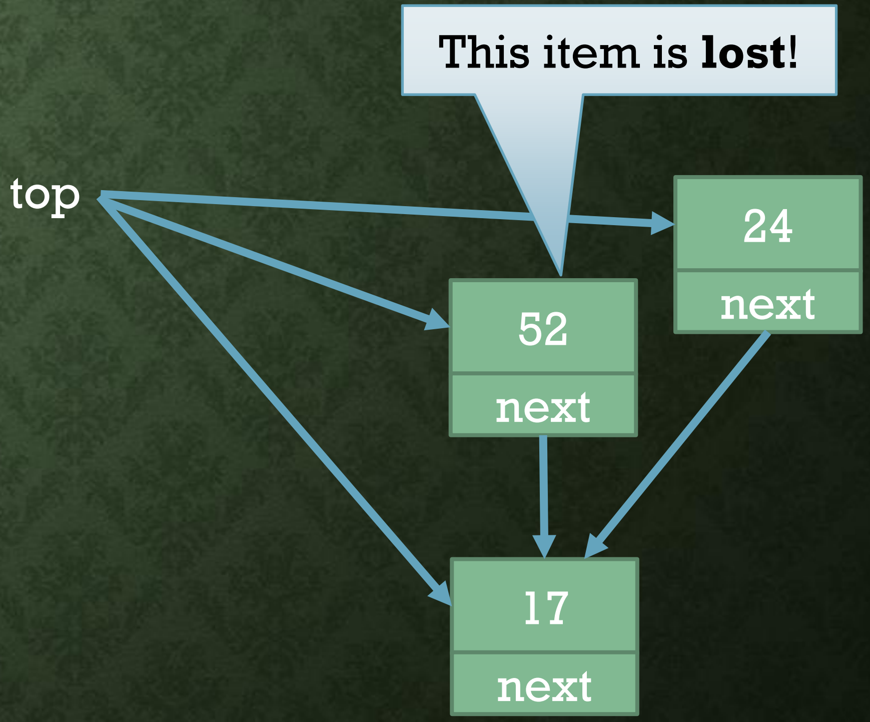
EXAMPLE EXECUTION

- Push(17)
  - Read top
  - Create node
  - Change top
- Push(52)
- Push(24)
- Pop()
  - Read top and see node(24)
  - Change top to node(52) and return 24

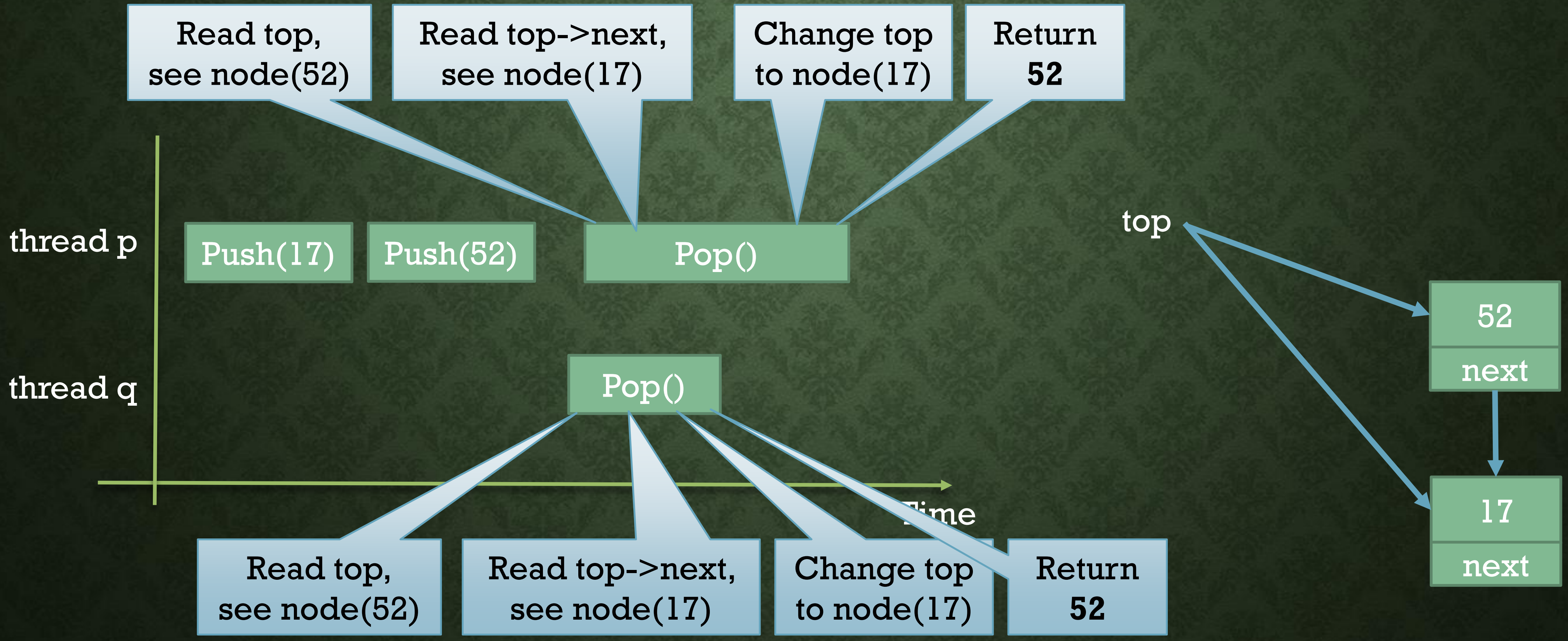


WHAT CAN GO WRONG?





ANOTHER EXAMPLE



WHAT’S THE PROBLEM HERE?

- Algorithmic step 1: read the value(s) that determine **what** we will write
- Algorithmic step 2: perform the Write
- Anything that happens in between is **ignored / overwritten**!
- Reads and writes are not enough

A MORE POWERFUL PRIMITIVE

- Compare-and-swap (CAS)
- Atomic instruction implemented in most modern architectures (even mobile/embedded)
- Idea: a write that succeeds only if a location contains an "expected" value **exp**
- Semantics

```
CAS(addr, exp, new)
    if (*addr == exp) {
        *addr = new;
        return true;
    }
    return false;
```

Implemented **atomically** in hardware

CAS-BASED STACK [TREIBER86]

```
void stack::push(int key)
  node * n = new node(key);
  while (true) {
    node * curr = top;
    n->next = curr;
    if (CAS(&top, curr, n)) return;
  }
```

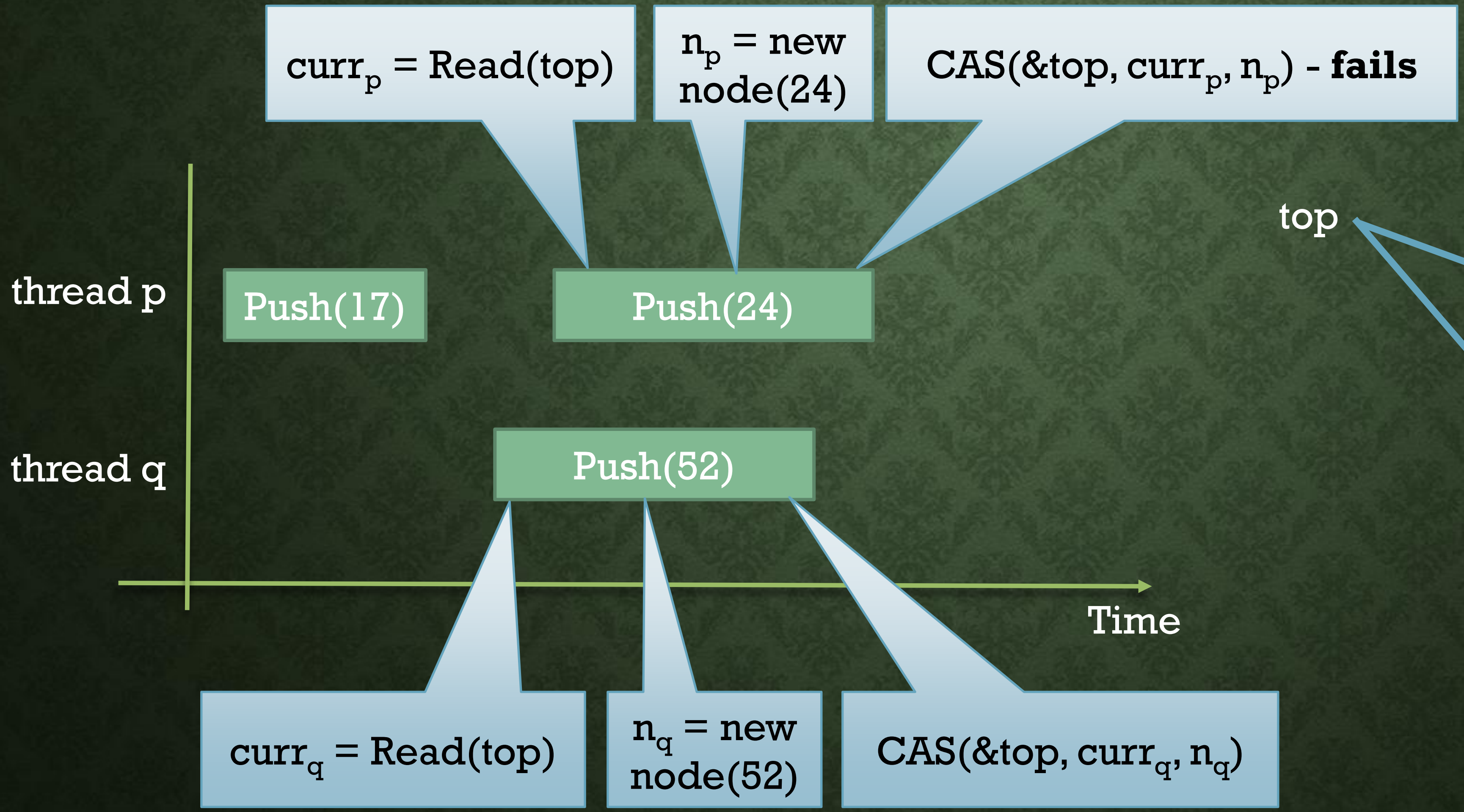
Change top **<u>from curr</u>** to n

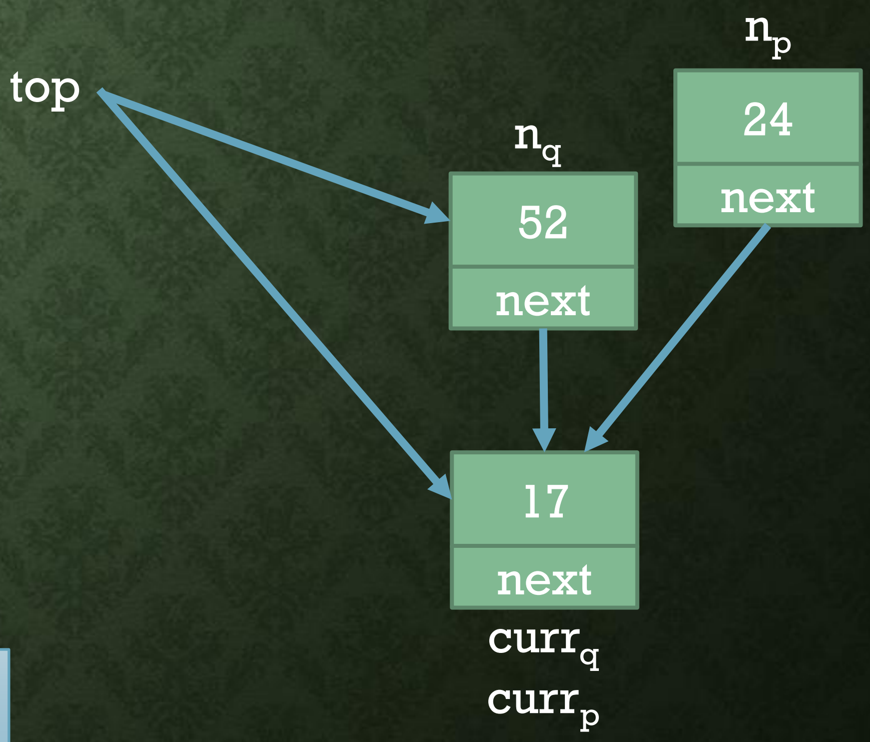
```
int stack::pop()
  while (true) {
    node * curr = top;
    if (curr == NULL) return EMPTY;
    node * next = curr->next;
    if (CAS(&top, curr, next)) {
      return curr->key;
    }
  }
```

Change top **<u>from curr</u>** to curr->next

HOW DOES THIS AVOID ERRORS?

Thread p will reread top and try again





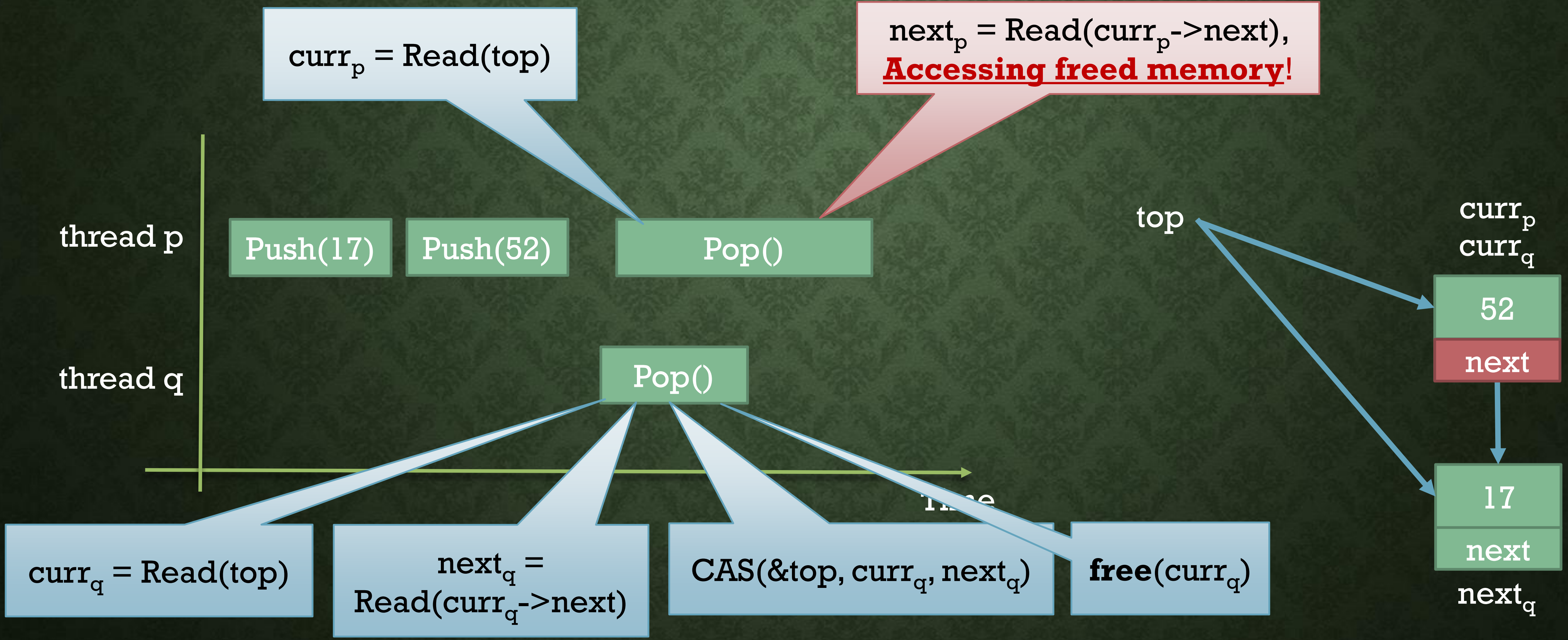
WHAT ABOUT FREEING MEMORY?

```
int stack::pop()
  while (true) {
    node * curr = top;
    if (curr == NULL) return EMPTY;
    node * next = curr->next;
    if (CAS(&top, curr, next)) {
      free(curr);
      auto retval = curr->key;
      return retval;
    }
  }
```

We disconnected a node from the stack!
Why not call **free()** on it?

We are not locking before accessing nodes…
Multiple threads might be accessing curr!

PROBLEMS FREEING MEMORY



USING FREE CORRECTLY

- Must **delay** free(curr) until it is **safe**!
- When is it safe?
  - When **no other thread** has a pointer to curr
- There are memory reclamation algorithms designed to solve this
  - Hazard pointers, epoch-based reclamation, garbage collection (Java, C#)
  - Usually provide a **delayedFree** operation (plus some other operations)

Will see how to use such an algorithm in an assignment

- We won't worry about memory reclamation for now…

WHAT DOES THE STACK GUARANTEE?

- Correctness (safety): Linearizable
  - Every concurrent execution is equivalent to some valid execution of a sequential stack
  - (as long as we don’t screw up memory reclamation)

WHAT DOES THE STACK GUARANTEE?

- Progress (liveness): Lock-free
  - Some thread will always make progress,
    even if some threads can **crash**
    - A crashed thread stops taking steps forever
    - Crashed threads are indistinguishable from **very slow** threads
      (because threads can be **unboundedly** slow)
  - Allows some threads to **starve**
  - But **not all** threads will starve
  - Algorithms are usually designed so starvation is <u>rare in practice</u>

REASONING ABOUT PROGRESS

- What can prevent progress in the stack?
  - Unbounded while loops
- When do we break out of a loop?
  - When we do a successful CAS
  - What can prevent a successful CAS?
  - A **concurrent** successful CAS by another thread p

```
void stack::push(int key)

  node * n = new node(key);
  while (true) {
    node * curr = top;
    n->next = curr;
    if (CAS(&top, curr, n)) return;
  }
```

```
int stack::pop()

  while (true) {
    node * curr = top;
    if (curr == NULL) return EMPTY;
    node * next = curr->next;
    if (CAS(&top, curr, next)) {
      return curr->key;
    }
  }
```

MECHANICS OF PROVING PROGRESS

- Proof by contradiction
- Assume threads keep taking steps forever, but progress stops
- After some time t, no operation terminates, so everyone is stuck in a while loop
- To continue in their while loops,
  threads must continue to perform failed CASs forever after t
- Every failed CAS is concurrent with a successful CAS,
  which is performed by an operation that will not perform any more loop iterations
- Eventually, no running operation will perform loop iterations → contradiction!

RECAP

- Finishing up last class
  - Predicting performance on multicore machines revolves largely around the caches
- Instruction reordering
  - C++ atomics: relaxed memory orders --- fast but **dangerous**
  - Memory fences
- Important definition: lock-freedom
- Lock-free stack (implementation and progress proof sketch)