Engineering Robust Server Software

Scalability

Duke UNIVERSITY

Andrew Hilton / Duke ECE

Impediments to Scalability

- Shared Hardware
  - Functional Units
  - Caches
  - Memory Bandwidth
  - IO Bandwidth
  - …
- Data Movement
  - From one core to another
- Blocking
  - Blocking IO
  - Locks (and other synchronization)

Let's talk about this now.

Locks + Synchronization

- Locks
  - Quick review of basics of locking
  - Non-obvious locks
  - Reader/writer locks
  - Locking granularity
- Memory Models/Memory Consistency [abbreviated version]
  - Compiler and/or hardware re-ordering
  - C++ atomics

Locks: Basic Review

- Need synchronization for correctness
- …but hate it from a performance standpoint
  - Why?

Locks: Basic Review

- Need synchronization for correctness
- ...but hate it from a performance standpoint
  - Why?
- Violates our rule of scalability
  - Contended lock = thread blocks waiting for it
- More data movement
  - Even if lock is uncontended, data must move through system

Synchronization

- Things you should already know
  - Mutexes:
    - pthread_mutex_lock
    - pthread_mutex_unlock
  - Condition variables:
    - pthread_cond_wait
    - pthread_cod_signal
  - Reader/writer locks:
    - pthread_rwlock_rdlock
    - pthread_rwlock_wrlock
    - pthread_rwlock_unlock

Synchronization Review (cont'd)

- Implementation: Atomic operations
  - Atomic CAS
  - Atomic TAS
- Likely want to test first, then do atomic
- Need to be aware of reordering (more on this later)
- Rusty? Review Aop Ch 28

Locking Overhead

- How long does this take
  - pthread_mutex_lock(&lock);
  - pthread_mutex_unlock(&lock);
- Assume lock is uncontended
- Lock variable is already in L1 cache
  - A: 15 cycles
  - B: 75 cycles
  - C: 300 cycles
  - D: 1200 cycles

Locking Overhead

- How long does this take
  - pthread_mutex_lock(&lock);
  - pthread_mutex_unlock(&lock);
- Assume lock is uncontended
- Lock variable is already in L1 cache
- Depends, but measured on an x86 core: about 75 cycles
- Rwlocks are worse: about 110 cycles

Analyze Locking Behavior

```
pthread_mutex_lock(&queue->lock);
while(queue->isEmpty()) {
   pthread_cond_wait(&queue->cv, &queue->lock);
}
req_t * r = queue->dequeue();
pthread_mutex_unlock(&queue->lock);
fprintf(logfile, "Completing request %ld\n", r->id);
delete r;
```

- Tell me about the synchronization behavior of this code
  - Where does it lock/unlock what?

Analyze Locking Behavior

```
pthread_mutex_lock(&queue->lock);
while(queue->isEmpty()) {
   pthread_cond_wait(&queue->cv, &queue->lock);
}
req_t * r = queue->dequeue();
pthread_mutex_unlock(&queue->lock);
fprintf(logfile, "Completing request %ld\n", r->id);
delete r;
```

- Ok, that one is obvious….

Duke UNIVERSITY

Analyze Locking Behavior

```
pthread_mutex_lock(&queue->lock);
while(queue->isEmpty()) {
   pthread_cond_wait(&queue->cv, &queue->lock);
}
req_t * r = queue->dequeue();
pthread_mutex_unlock(&queue->lock);
fprintf(logfile, "Completing request %ld\n", r->id);
delete r;
```

"The stdio functions are thread-safe. This is achieved by assigning to each FILE object a lockcount and (if the lockcount is nonzero) an owning thread. For each library call, these functions wait until the FILE object is no longer locked by a different thread, then lock it, do the requested I/O, and unlock the object again."

— man flockfile

Duke UNIVERSITY

Stdio Locking

- Stdio locked by default
  - Generally good: want sane behavior writing to FILEs
- Can manually lock with flockfile
  - Guarantee multiple IO operations happen together
  - Can use _unlocked variants when holding a lock (or guaranteed no races)
- Hidden scalability dangers
  - Writing log file from multiple threads? Contending for a lock
  - Moving lock variable around system…
  - Waiting for IO operations can take a while
    - Small writes ~400 cycles -> /dev/null, ~2500 to a real file
    - Much worse if we force data out of OS cache to disk

Analyze Locking Behavior

```
pthread_mutex_lock(&queue->lock);
while(queue->isEmpty()) {
   pthread_cond_wait(&queue->cv, &queue->lock);
}
req_t * r = queue->dequeue();
pthread_mutex_unlock(&queue->lock);
fprintf(logfile, "Completing request %ld\n", r->id);
delete r;
```

- Memory allocator has to be thread safe (new/delete on any thread)
  - Delete locks the free list…
  - Contends with any other new/delete/malloc/free

Analyze Locking Behavior

```
pthread_mutex_lock(&queue->lock);
while(queue->isEmpty()) {
   pthread_cond_wait(&queue->cv, &queue->lock);
}
req_t * r = queue->dequeue();
pthread_mutex_unlock(&queue->lock);
fprintf(logfile, "Completing request %ld\n", r->id);
delete r;
```

- Probably some memory deallocation in here too
  - Also locks free list

Analyze Locking Behavior

```
pthread_mutex_lock(&queue->lock);
while(queue->isEmpty()) {
   pthread_cond_wait(&queue->cv, &queue->lock);
}
req_t * r = queue->dequeue();
pthread_mutex_unlock(&queue->lock);
fprintf(logfile, "Completing request %ld\n", r->id);
delete r;
```

- Probably some memory deallocation in here too
  - Also locks free list
  - Inside another critical section:
    - Waiting for free list -> hold queue's lock longer!

Duke University

Memory Allocation/Free Ubiquitous

- Memory allocation/deallocation happens all over the place:
  - Add to a vector?
  - Append to a string?
  - ….
- What can we do?
  - Simplest: use scalable malloc library, such as libtcmalloc
    - Easy: -ltcmalloc
    - Thread cached malloc: each thread keeps local pool (no lock for that)

Improving Scalability

- Three ideas to improve scalability
  - Reader/writer locks
  - Finer granularity locking
  - Get rid of locks

R/W Locks

- (Review): Reader/writer locks
  - Multiple readers
  - OR single writer
- Mostly reads?
  - Reads occur in parallel
  - Scalability improves
- Is that all there is to it?

R/W Lock Implementation?

- How do you make a r/w lock?
  - Everyone take a second to think about it…

Duke UNIVERSITY

Option 1: Mutex + Condition

```
struct rwlock_t {
   mutex_lock_t lock;
   cond_t cond;
   int readers;
   int anyWriter;
};
```

```
void read_lock(rwlock_t * rw) {
   mutex_lock(&rw->lock);
   while (rw->anyWriter) {
      cond_wait(&rw->cond);
   }
   rw->readers++;
   mutex_unlock(&rw->lock);
}
```

```
void write_lock(rwlock_t * rw) {
   mutex_lock(&rw->lock);
   while (rw->readers > 0 ||
          rw->anyWriter) {
      cond_wait(&rw->cond);
   }
   rw->anyWriter = true;
   mutex_unlock(&rw->lock);
}
```

Option 1: Mutex + Condition

```
struct rwlock_t {
   mutex_lock_t lock;
   cond_t cond;
   int readers;
   int anyWriter;
};
```

```
void unlock(rwlock_t * rw) {
   mutex_lock(&rw->lock);
   if (rw->anyWriter) {
      rw->anyWriter = false;
      cond_broadcast(&rw->cond);
   }
   else {
      rw->readers—;
      if (rw->readers == 0) {
           cond_signal(&rw->cond);
      }
   }
  mutex_unlock(&rw->lock);
}
```

Duke UNIVERSITY

Option 2: Two Mutexes

```
struct rwlock_t {
   mutex_lock_t rlck;
   mutex_lock_t wlck;
   int readers;
};
```

```
void read_lock(rwlock_t * rw) {
   mutex_lock(&rw->rlck);
   if (rw->readers == 0) {
      mutex_lock(&rw->wlck);
   }
   rw->readers++;
   mutex_unlock(&rw->rlck);
}
```

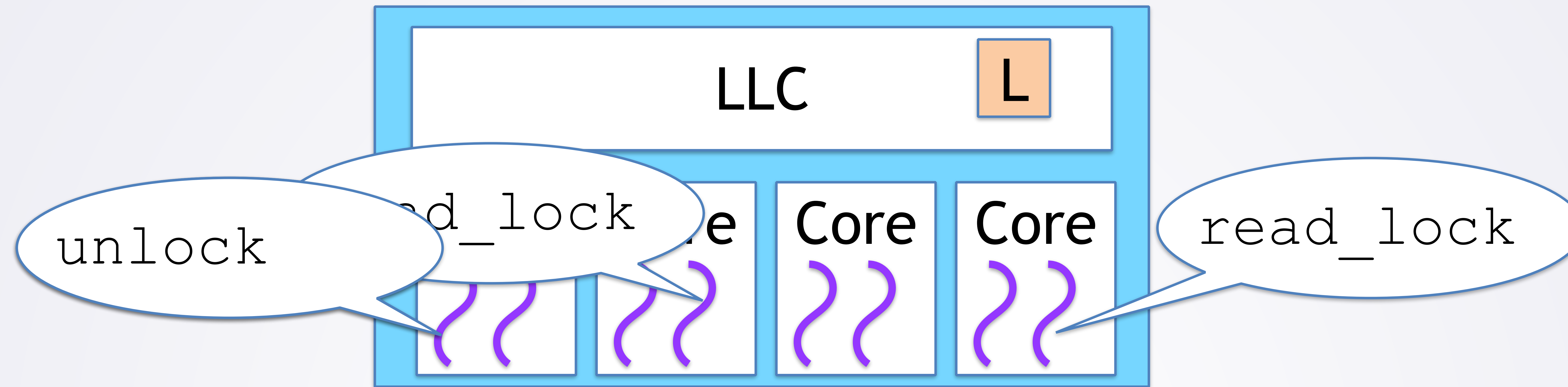
```
void write_lock(rwlock_t * rw)
   mutex_lock(&rw->wlck);
}
```

Duke UNIVERSITY

Other R/W Lock Issues

- These can both suffer from write starvation
  - If many readers, writes may **starve**
  - Can fix: implementation becomes more complex
- What about upgrading (hold read -> atomically switch to write)?
- What about performance?
  - We know un-contended locks have overhead…
  - What if many threads read at once?
    - Not truly "contended"—-r/w lock allows in parallel
    - …but how about overheads?

Either One: Data Movement To Read Lock




```
void read_lock(rwlock_t * rw) {
    mutex_lock(&rw->lock);
    while (rw->anyWriter) {
        cond_wait(&rw->cond);
    }
    rw->readers++;
    mutex_unlock(&rw->lock);
}
```

```
void read_lock(rwlock_t * rw) {
    mutex_lock(&rw->rlck);
    if (rw->readers == 0) {
        mutex_lock(&rw->wlck);
    }
    rw->readers++;
    mutex_unlock(&rw->rlck);
}
```

Duke UNIVERSITY

What Does This Mean?

- R/W lock is not a "magic bullet"
  - Data movement still hurts scalability
  - How much? Depends on size of critical section
- Could make lock more read-scalable
  - More scalable = more complex…

Locking Granularity

- Can use many locks instead of one
  - Lock guards smaller piece of data
  - Multiple threads hold different locks -> parallel
  - Data movement? Different locks = different data -> less movement
- Simple example
  - One lock per hashtable bucket
  - Add/remove/find: lock one lock
    - Different threads -> good odds of locking different locks
    - How good?…

Quick Math Problem

- Suppose I have 256 locks and 32 threads
  - Each thread acquires one lock (suppose random/uniform)
  - Probability that two threads try to acquire the same lock?
    - A: 25%
    - B: 40%
    - C: 87%
    - D: 99.9%

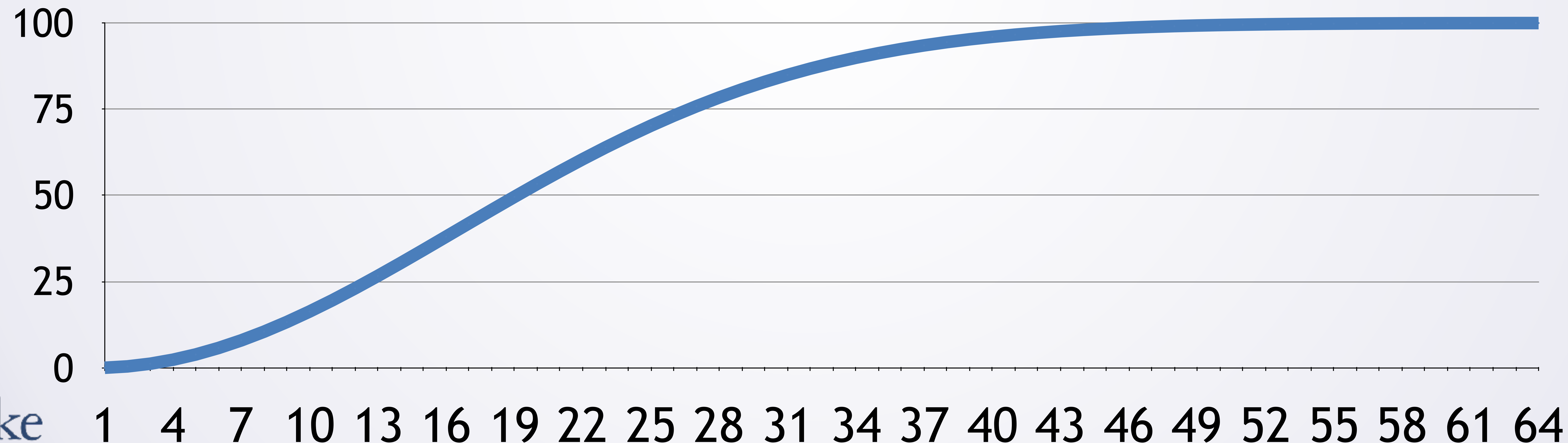
Quick Math Problem

- Suppose I have 256 locks and 32 threads
  - Each thread acquires one lock (suppose random/uniform)
  - Probability that two threads try to acquire the same lock?
  - What if there are 64 threads?
    - A: 93%
    - B: 95%
    - C: 99%
    - D: More than 99%

Duke University

Quick Math Problem

- Suppose I have 256 locks and 32 threads
  - Each thread acquires one lock (suppose random/uniform)
  - Probability that two threads try to acquire the same lock? **87%**
  - What if there are 64 threads? **99.98%**
- Probability all different (32 thr)= 256/256 * 255/256 * 254/256 * …. 225/256



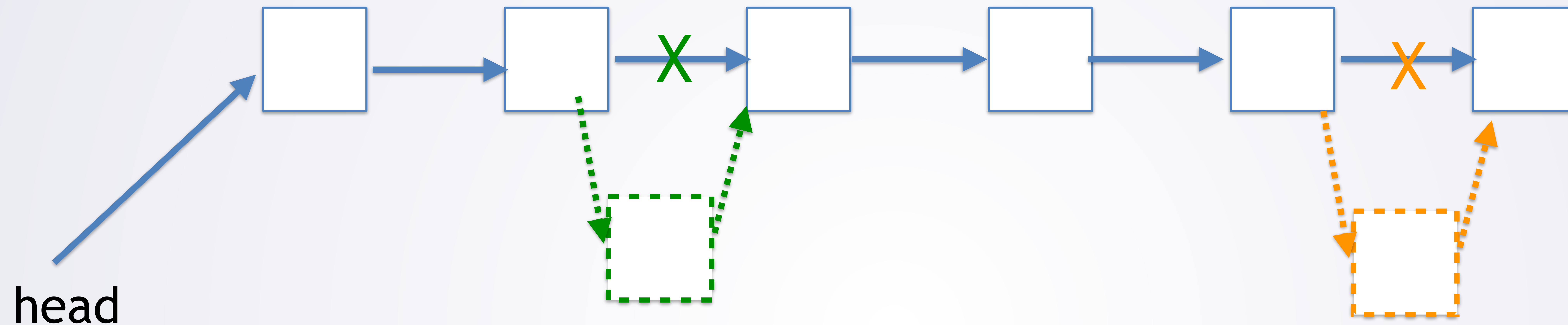
Duke UNIVERSITY

Birthday Paradox

- This is called the "birthday paradox"
  - If we have N people in a room, what are the odds 2 have the same bday?
  - Assume no Feb 29th
  - Assume uniform distribution (does not exactly hold)
- Comes up a lot in security also
  - Why?

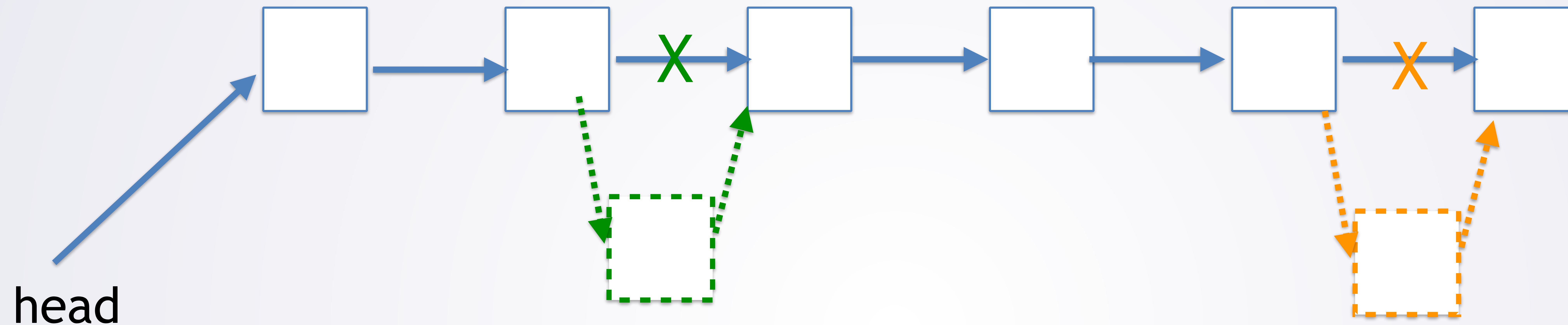
Duke UNIVERSITY

Hand Over Hand Locking



- Suppose I have an LL and need concurrency **within the list**
  - Different threads operating on different nodes in parallel

Hand Over Hand Locking



- I could have a bunch of locks
  - One for head
  - One for each node
- Acquire them "hand over head"
  - Lock next, then release current

Hand Over Hand Locking

```
void addSorted(int x) {
   pthread_mutex_t * m= &hdlck;
   pthread_mutex_lock(m);
   Node ** ptr = &head;
   while (*ptr != nullptr &&
          (*ptr)->data < x) {
     Node * c = *ptr;
     pthread_mutex_lock(&c->lck);
     pthread_mutex_unlock(m);
     m = &c->lck;
     ptr = &c->next;
   }
   *ptr = new Node(x, *ptr);
   pthread_mutex_unlock(m);
}
```

When is this a good idea?
When is this a bad idea?

Duke UNIVERSITY

Hand Over Hand Locking

- Locking overheads are huge
  - Lock/unlock per node.
  - Good if operations are slow + many threads at once
    - Increase parallelism, amortize cost of locking overheads
- How should we evaluate this?
  - Measure, graph
  - **Don't just make guesses.**

Duke UNIVERSITY

Fine Grained Locking

- Best: partition data
  - This is what we do in the HT example
  - Can we do it for other things?
    - Sure, but may need to redesign data structures
    - List? Multiple lists each holding ranges of values
      - Wrapped up in abstraction that LOOKS like regular list
- Other strategies:
  - Consider lock overheads
  - HoH would work better if we did locks for 100 nodes at a time
    - But really complicated

Duke
UNIVERSITY

So Why Not Just Get Rid Of Locks?

- Locks are bad for performance…
  - So let's just not use them!
- But how do we maintain correctness?

So Why Not Just Get Rid Of Locks?

- Locks are bad for performance…
  - So let's just not use them!
- But how do we maintain correctness?
  - Atomic operations (e.g., atomic increment, atomic CAS)
  - Lock free data structures
  - Awareness of reordering rules
    - And how to ensure the ordering you need

What Can This Print

a = 0

b = 0

Thread 0

b = 1
c = a

Thread 1

a = 1
d = b

Join

```
printf("c=%d\n", c);
printf("d=%d\n", d);
```

- What are the possible outcomes?

What Can This Print

a = 0

b = 0

Thread 0

1 b = 1
3 c = a

Thread 1

2 a = 1
4 d = b

Join

```
printf("c=%d\n", c);
printf("d=%d\n", d);
```

Possible?	*c*	*d*
Yes	*1*	*1*
	0	*1*
	1	*0*
	0	*0*

- What are the possible outcomes?

Duke UNIVERSITY

What Can This Print

```
a = 0
b = 0
```

Thread 0

```
1 b = 1
2 c = a
```

Thread 1

```
3 a = 1
4 d = b
```

Join

```
printf("c=%d\n", c);
printf("d=%d\n", d);
```

Possible?	*c*	*d*
Yes	*1*	*1*
Yes	*0*	*1*
	1	*0*
	0	*0*

- What are the possible outcomes?

Duke University

What Can This Print

```
a = 0
b = 0
```

Thread 0

```
b = 1
c = a
```

Thread 1

```
a = 1
d = b
```

Join

```
printf("c=%d\n", c);
printf("d=%d\n", d);
```

Possible?	*c*	*d*
Yes	*1*	*1*
Yes	*0*	*1*
Yes	*1*	*0*
Depends	*0*	*0*

- What are the possible outcomes?

How is c=0, d=0 possible?

- First: compiler might re-order instructions
  - Why? Performance
  - But what if the actual assembly is in this order?

Duke
UNIVERSITY

How is c=0, d=0 possible?

- First: compiler might re-order instructions
  - Why? Performance
  - But what if the actual assembly is in this order?
- Hardware may be allowed to **observably** reorder memory operations
  - Rules for this are the memory consistency model, part of the ISA

Duke
UNIVERSITY

Memory Consistency Models

	Sequential Consistency	x86	POWER
Ld ; Ld	In Order	In Order	Reorderable (unless dependent)
Ld ; St	In Order	In Order	Reorderable
St ; St	In Order	In Order	Reorderable
St; Ld	In Order	Reorderable	Reorderable

Why Reordering/Why Restrict It?

- Hardware designers: **Reordering is great!**
  - Higher performance
    - Do other operations while waiting for stalled instructions

Duke UNIVERSITY

Why Reordering/Why Restrict It?

- Hardware designers: Reordering is great!
  - Higher performance
    - Do other operations while waiting for stalled instructions
- Software writers: Reordering is painful!
  - Already hard to reason about code
  - Now may be even harder: not in the order you wrote it
  - Surprising behaviors -> bugs
    - If you don't understand what your code does, it isn't right

How to Write Code?

- How to handle correct/high performance code?
  - Different hw->different rules
- Sometimes we **need** order
  - E.g., lock; (critical section); unlock;

	Sequential Consistency	x86	POWER
Ld ; Ld	In Order	In Order	Reorderable (unless dependent)
Ld ; St	In Order	In Order	Reorderable
St ; St	In Order	In Order	Reorderable
St; Ld	In Order	Reorderable	Reorderable

How to Write Code?

- How to handle correct/high performance code?
  - Different hw->different rules
- Sometimes we **need** order
  - E.g., lock; (critical section); unlock;
- Hardware has instructions to force ordering ("fences")
  - Use when needed
  - Give correctness
  - Cost performance

	Sequential Consistency	*x86*	*POWER*
Ld ; Ld	*In Order*	*In Order*	*Reorderable (unless dependent)*
Ld ; St	*In Order*	*In Order*	*Reorderable*
St ; St	*In Order*	*In Order*	*Reorderable*
St; Ld	*In Order*	*Reorderable*	*Reorderable*

Duke UNIVERSITY

C++ Atomics

- In C++, use std::atomic<T> (use for anything no guarded by a lock)
  - Has .load and .store
  - These each require a std::memory_order to specify ordering

Duke UNIVERSITY

C++ Atomics

- In C++, use std::atomic<T> (use for anything no guarded by a lock)
  - Has .load and .store
  - These each require a std::memory_order to specify ordering

Load	Store
std::memory_order_seq_cst	std::memory_order_seq_cst
std::memory_order_acquire	std::memory_order_release
std::memory_order_consume	
std::memory_order_relaxed	std::memory_order_relaxed

C++ Atomics

- What do all these mean?
  - Formal, precise definition: Complex (ECE 565)
  - Basic ideas here.

Load	Store
std::memory_order_seq_cst	std::memory_order_seq_cst
std::memory_order_acquire	std::memory_order_release
std::memory_order_consume	
std::memory_order_relaxed	std::memory_order_relaxed

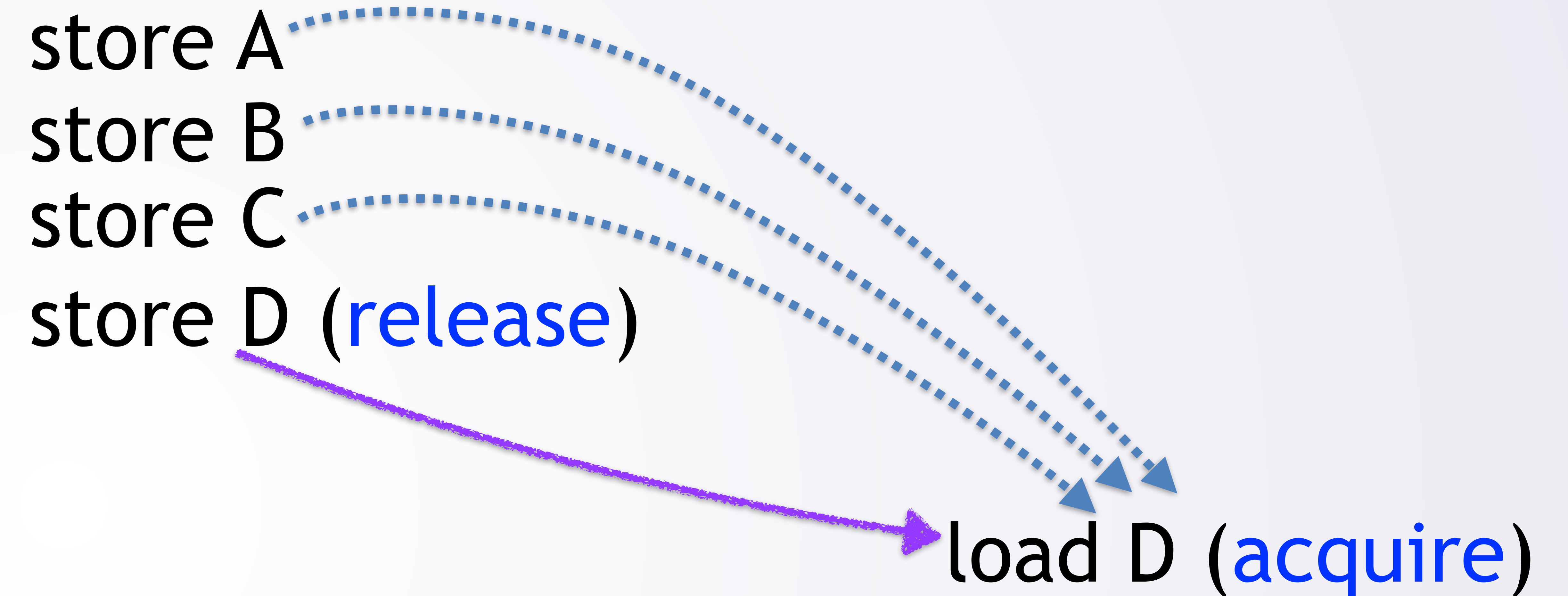
Duke UNIVERSITY

Relaxed Semantics

- Relaxed ordering: minimal guarantees
  - Disallows some really weird behavior
  - …but does not ask hardware to enforce anything special

Acquire/Release Semantics

- What we want is acquire/release semantics
  - Load: acquire
  - Store: release



- When load (acquire) receives value from store (release)
  - All prior stores in the releasing thread become **visible side-effects**
  - In acquiring thread (only)

Uses of Acquire/Release

- Locks: where the name comes from
- Store data, indicate it is ready
  - Write data;
  - Store (release) ready = 1
  - Load (acquire) to check if read
  - Read data

Sequential Consistency

- memory_order_seq_cst: sequentially consistent operations
  - With respect to other memory_order_seq_cst operations
  - May not be SC with respect to other operations
- Sequential Consistency = What you would hope for as programmer
  - Do loads/stores from each thread with some interleaving
  - That respects ordering in each thread

Atomics: Things We Can Do Without Locks

```
std::atomic<int> counter(0);

//...


int x = counter.load(/* some memory order*/);
x++;
counter.store(x, /* some memory order */);
```

- Suppose we wanted to increment a counter w/o a lock
  - Does this work?
  - Does the memory order we pick matter?

Duke
UNIVERSITY

Atomics: Things We Can Do Without Locks

```
std::atomic<int> counter(0);

//...


int x = counter.load(std::memory_order_seq_cst);
x++;
counter.store(x, std::memory_order_seq_cst);
```

Broken

- Suppose we wanted to increment a counter w/o a lock
  - Does this work? No
  - Does the memory order we pick matter? Broken even if we use SC

Duke UNIVERSITY

Atomics: Things We Can Do Without Locks

```
std::atomic<int> counter(0);

//......

counter.fetch_add(1, /* what memory order? */);
```

- We need load-add-store to be **atomic**
  - Fortunately, C++ atomics support this
  - Use hardware atomic operations

Duke UNIVERSITY

Atomics: Things We Can Do Without Locks

```
std::atomic<int> counter(0);

//......

counter.fetch_add(1, std::memory_order_relaxed);
```

- We need load-add-store to be **atomic**
  - Fortunately, C++ atomics support this
  - Use hardware atomic operations
  - For counters, generally relaxed memory ordering is fine [why?]

Duke UNIVERSITY

Other Atomic Operations

- C++ Atomics support
  - load/store
  - fetch_add/fetch_sub/fetch_and/fetch_or/fetch_xor
  - exchange
  - compare_exchange
    - weak: May fail spuriously
    - strong: Won't fail spuriously
- RMW operations, may want memory_order_acq_rel
- Note that for some T, atomic<T> may use locks
  - Can check with is_lock_free()