Engineering Robust Server Software

Scalability

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
  - Compiler and/or hardware re-ordering
  - Happens-before
  - C++ atomics
- Lock free data structures
- SLE and HTM

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

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….

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

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!

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…

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);
}
```

Option 2: Two Mutexes

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

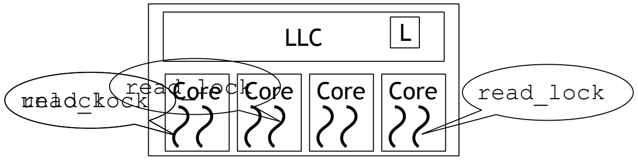
```
void read_lock(rwlock_t * rw) {
   mutex_lock(&rw->rlck);
   if (rw->readers == 0) {
      mutex_lock(&rw->wlck);
   }
   rw->readers++;
   mutex_unlock(&rw->rlck);
}
```

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

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);
}
```

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

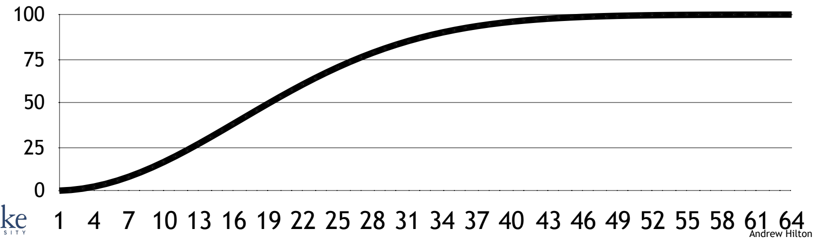
- Can use many locks instead of one
  - Lock guards smaller piece of data
  - Multiple threads hold different locks -> parallel
  - Data movement? Different locks = different data -> less movement
- Simple example
  - One lock per hashtable bucket
  - Add/remove/find: lock one lock
    - Different threads -> 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?
  - What if there are 64 threads?

Quick Math Problem

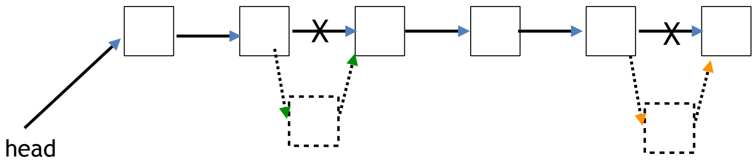
- Suppose I have 256 locks and 32 threads
  - Each thread acquires one lock (suppose random/uniform)
  - Probability that two threads try to acquire the same lock? **87%**
  - What if there are 64 threads? **99.98%**
- Probability all different (32 thr)= 256/256 * 255/256 * 254/256 * …. 225/256



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?

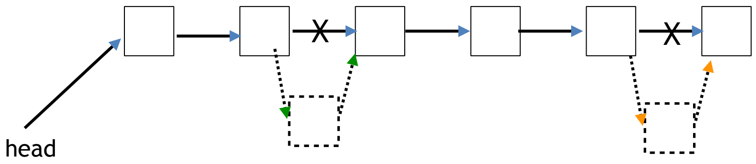
Hand Over Hand Locking



- 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?

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.

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

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        Thread 1

b = 1           a = 1
c = a           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        Thread 1

1  b = 1        2 a = 1
3  c = a        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?

What Can This Print

```
a = 0
b = 0

Thread 0        Thread 1

1  b = 1        3 a = 1
2  c = a        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?

What Can This Print

```
            a = 0
            b = 0
Thread 0              Thread 1
  b = 1                 a = 1
  c = a                 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?

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

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

---

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*

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

---

C++ Atomics

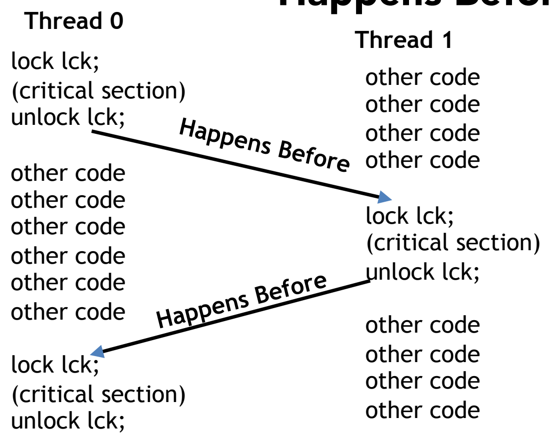
- In C++, use std::atomic<T> (use for anything no guarded by a lock)
  - Has .load and .store
  - These each require a std::memory_order to specify ordering

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

What Do These Mean?

- To understand these, need several new ideas
  - Happens-before relationship
  - Modification order: total ordering of memory operations on atomic
    - Note: does not exist for "regular" loads + stores

Happens Before



Thread 0

lock lck;
(critical section)
unlock lck;

other code
other code
other code
other code
other code
other code

lock lck;
(critical section)
unlock lck;

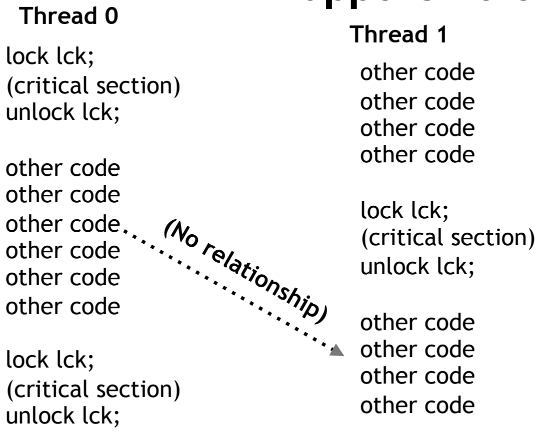
Thread 1

other code
other code
other code
other code

lock lck;
(critical section)
unlock lck;

other code
other code
other code
other code

Happens Before



Thread 0

lock lck;
(critical section)
unlock lck;

other code
other code
other code
other code
other code
other code

lock lck;
(critical section)
unlock lck;

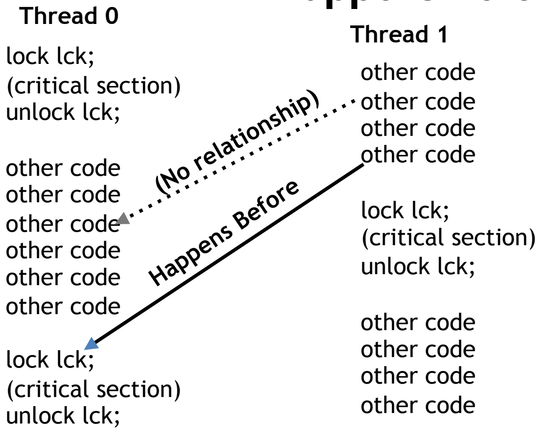
Thread 1

other code
other code
other code
other code

lock lck;
(critical section)
unlock lck;

other code
other code
other code
other code

Happens Before



Thread 0

lock lck;
(critical section)
unlock lck;

other code
other code
other code
other code
other code
other code

lock lck;
(critical section)
unlock lck;

Thread 1

other code
other code
other code
other code

lock lck;
(critical section)
unlock lck;

other code
other code
other code
other code

Happens Before

- Single-thread ordering rules
  - In C++ "sequenced before"
- Those created by memory order requirements
  - "synchronizes-with"
- Transitivity
  - A happens-before B & B happens-before C -> A happens-before C

Data Races/Undefined Behavior

- In C++: data race -> undefined behavior
  - Two (or more) accesses to same location
  - No happens-before relationship between them
  - At least one is not an atomic

Modification Order

- For atomics (not regular loads/stores): modification order
  - Total ordering of accesses to atomic
  - Who wants to remind us what a total order is? How about a partial order?
    - What have we seen recently that is a partial order?

Modification Order

- For atomics (not regular loads/stores): modification order
  - Total ordering of accesses to atomic
  - Who wants to remind us what a total order is? How about a partial order?
    - What have we seen recently that is a partial order?
- Total order: for any A and B, either A<=B, or B<=A,
- Partial order: May have A incomparable with B (neither A<=B, nor B<=A)
  - Happens before is a partial order
- Both are reflexive, anti-symmetric, and transitive

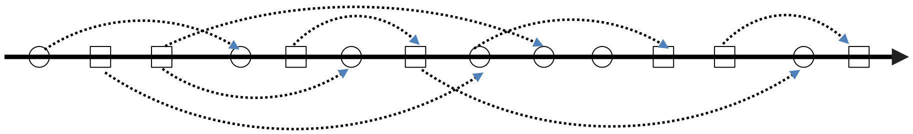
Modification Order

Total Order per atomic variable
- One ordering for x
- Another for y
...etc

○ Store
☐ Load
Color = Thread

Modification Order

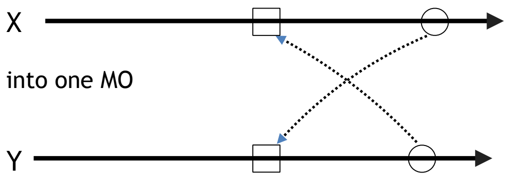
Respects happens-before
If A happens-before B, A must precede B in MO



Happens Before
○ Store
☐ Load
Color = Thread

Modification Order

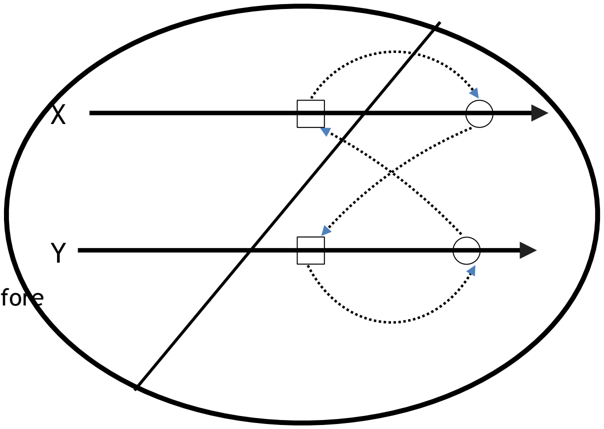
Two different variables:
  Two different MOs
  Do NOT need to be combinable into one MO



⇢ Happens Before
○ Store
☐ Load
Color = Thread

No Cycles in Happens Before

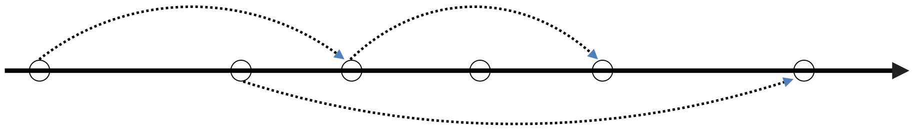
However, cycles are
NOT permissible in HB



⇢ Happens Before
○ Store
☐ Load
Color = Thread

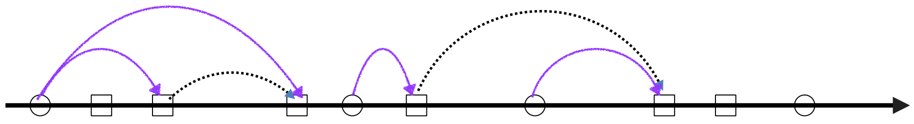
Modification Order: Write-Write

If W_1 happens before W_2, W_1 is before W_2 in the MO



⇢ Happens Before
○ Store
☐ Load
Color = Thread

Modification Order: Read-Read



→ Gets Value From
⇢ Happens Before
○ Store
☐ Load
Color = Thread

If R_1 happens before R_2,
and R_1 gets its value from W_1

Either:
  R_2 gets its value from W_1
OR
  R_2 gets its value from another write
  which is after W_1 in the MO

Modification Order: Read-Write

If R_1 happens before W_1, then R_1 gets its value from W_2
where W_2 is before W_1 in MO

Gets Value From

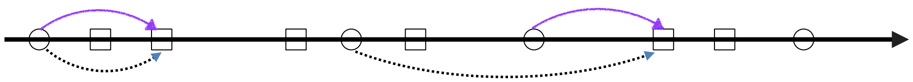
Happens Before

○ Store

☐ Load

Color = Thread

Modification Order: Write-Read



Gets Value From

Happens Before

○ Store

☐ Load

Color = Thread

If W_1 happens before R_1, then
Either

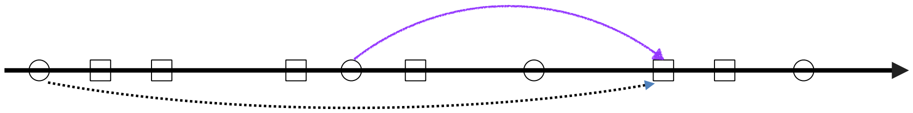
R_1 gets its value from W_1
OR
R_1 gets its value from W_2 where
W_2 is after W_1 in the MO

But Wait…

- I just gave you the intuitive pictures that interpreted those rules
  - But we can have non-intuitive outcomes that match the rules

---

Less Intuitive



Gets Value From

Happens Before

○ Store

☐ Load

Color = Thread

If W_1 happens before R_1, then
Either

R_1 gets its value from W_1
OR
R_1 gets its value from W_2 where
W_2 is after W_1 in the MO

Note: atomic RMWs are required to get most recent

Example Revisited

Thread 0

A a = 0;

B b = 0;

Spawn T1

Thread 1

a = 1; Y

d = b; Z

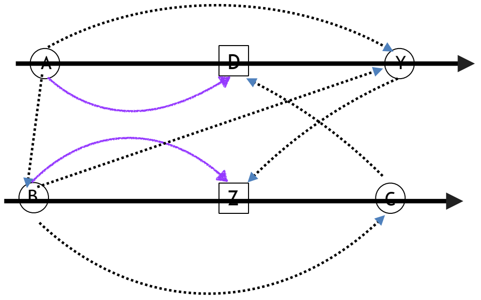
C b = 1;

D c = a;

Join

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

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



C++ Atomics

- Back to our orderings.
    - What we just saw is memory_order_relaxed
    - Guarantees Modification Ordering, but nothing else

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

Another Example

Thread 0

A a = 0;

B b = 0;

Spawn T1

Thread 1

Y while (b == 0) ;

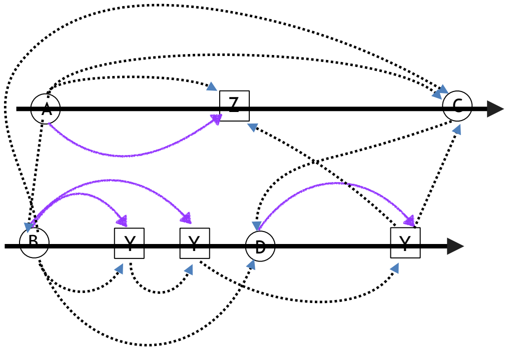
Z c = a

C a = 1

D b = 1;

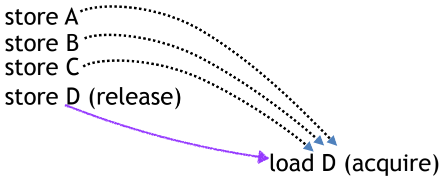
Join

printf("c=%d\n", c); c = 0



Acquire/Release Semantics

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



store A

store B

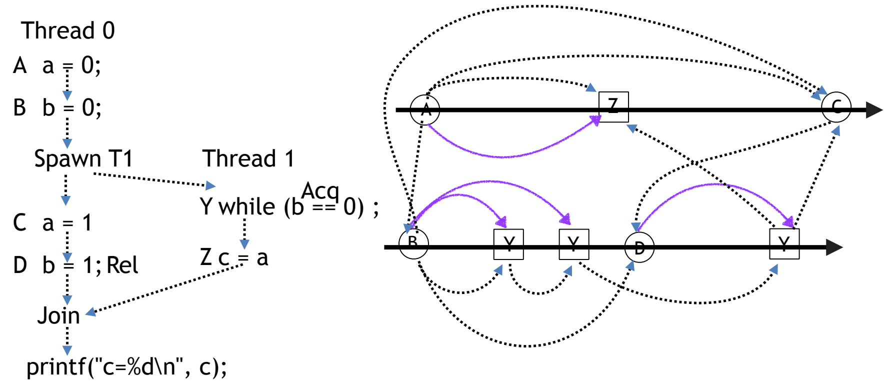
store C

store D (release)

load D (acquire)

- When load (acquire) receives value from store (release)
    - All prior stores in the releasing thread become visible-side effects
    - In acquiring thread (only)
    - Effectively: establishes happens-before for all these stores

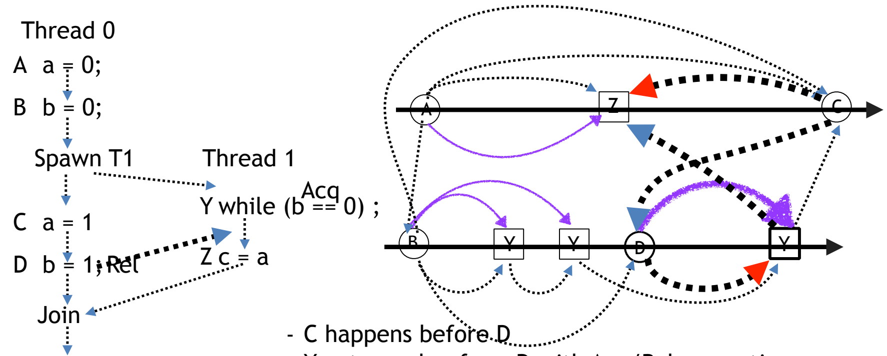
Another Example



Thread 0
A a = 0;
B b = 0;
Spawn T1
C a = 1
D b = 1; Rel
Join
printf("c=%d\n", c);

Thread 1
Acq
Y while (b == 0) ;
Z c = a

Another Example



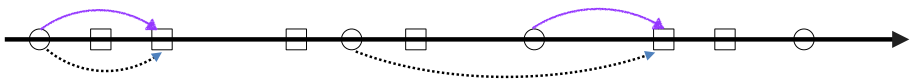
Thread 0
A a = 0;
B b = 0;
Spawn T1
C a = 1
D b = 1; Rel
Join
printf("c=%d\n", c);

Thread 1
Acq
Y while (b == 0) ;
Z c = a

- C happens before D
- Y gets a value from D with Acq/Rel semantics
  - Establishes D happens before Y
- Y happens before Z
- Now C happens before Z

Modification Order: Write-Read

Remember this rule?

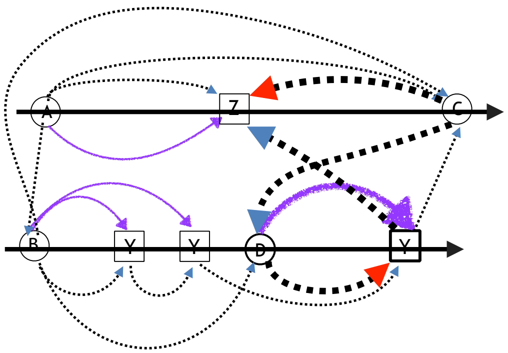


Gets Value From
Happens Before
○ Store
☐ Load
Color = Thread

If W_1 happens before R_1, then
Either

R_1 gets its value from W_1
OR
R_1 gets its value from W_2 where
W_2 is after W_1 in the MO

Another Example



If C happens before Z, then
Either

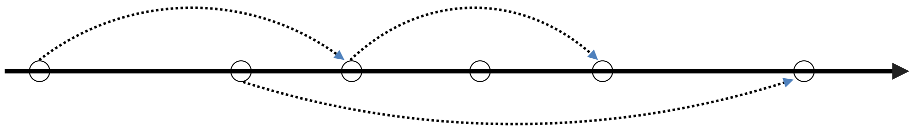
Z gets its value from C
OR
Z gets its value from A where
A is after C in the MO

Neither is true here

Can we put A after C?

Modification Order: Write-Write

If W_1 happens before W_2, W_1 is before W_2 in the MO



Happens Before

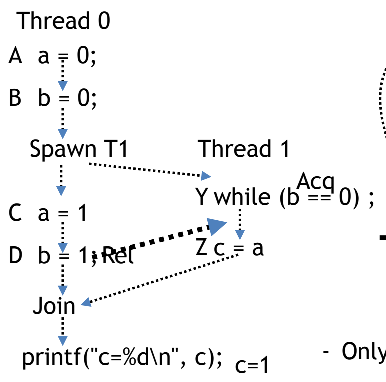
○ Store

☐ Load

Color = Thread

Cannot put A after C because of this rule

Another Example



Thread 0

A a = 0;

B b = 0;

Spawn T1

C a = 1

D b = 1; Rel

Join

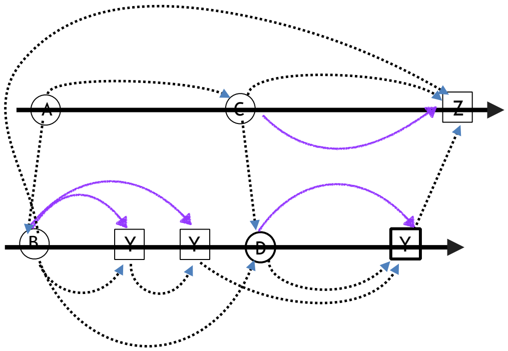
printf("c=%d\n", c);

Thread 1

Y while (b == 0) ; Acq

Z c = a

c=1



- Only way: Z gets value from C

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
- Effectively makes one total modification order of all SC variables
  - Loads observe most recent values in one total ordering
  - Total ordering respects all happens-before relationships
- Also gives all guarantees of acquire/release

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?

Atomics: Things We Can Do Without Locks

Broken

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

//…


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

- 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

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

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?]

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()