

Course Description

Lecture 18: Synchronization and Transactional Memory

This course covers the design and implementation of computer systems from the perspective of the hardware software interface. The purpose of this course is for students to understand the relationship between the operating system, software, and computer architecture. Students that complete the course will have learned operating system fundamentals, computer architecture fundamentals, compilation to hardware abstractions, and how software actually executes from the perspective of the hardware software/boundary. The course will focus especially on understanding the relationships between software and hardware, and how those relationships influence the design of a computer system's software and hardware. The course will convey these topics through a series of practical, implementation-oriented lab assignments.

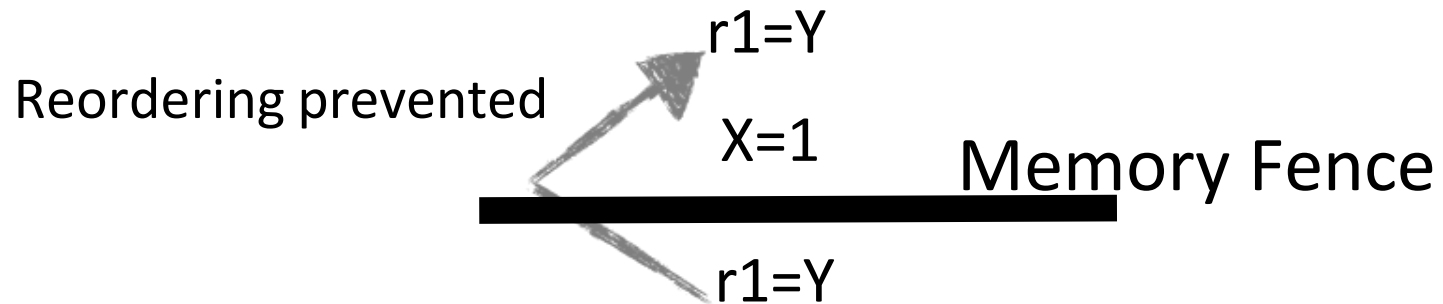
Credit: Brandon Lucia

Synchronization and its Implementation

Review:

Synchronization Can Prevent Operation Reordering

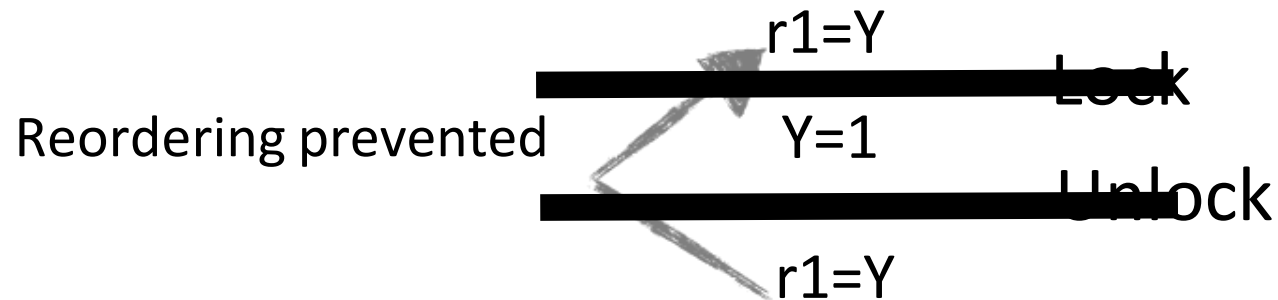
Memory fences are one type of synchronization



Fence implementation depends on reordering implementation

Review: Synchronization For Real Programmers

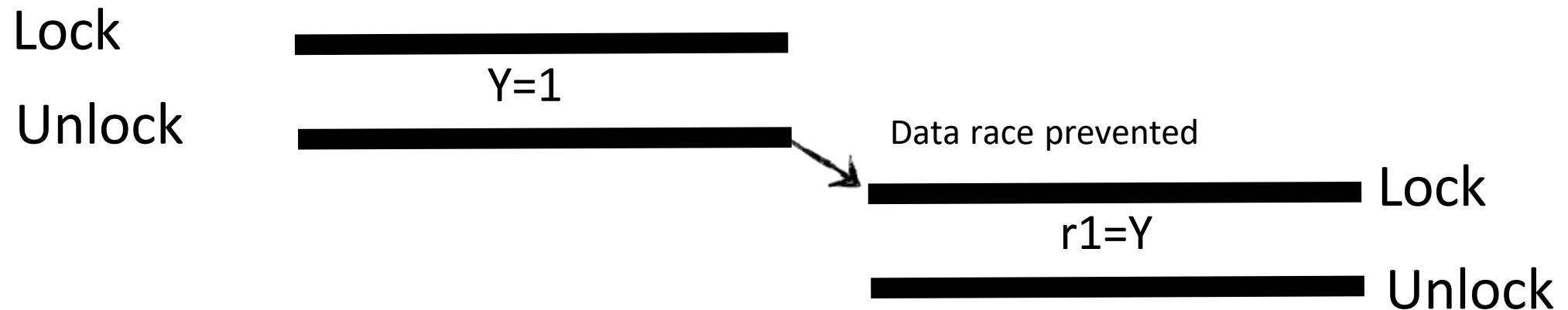
Memory fences are wrapped up in locks, etc.



Direct use of fences can be tricky and you will usually use a library

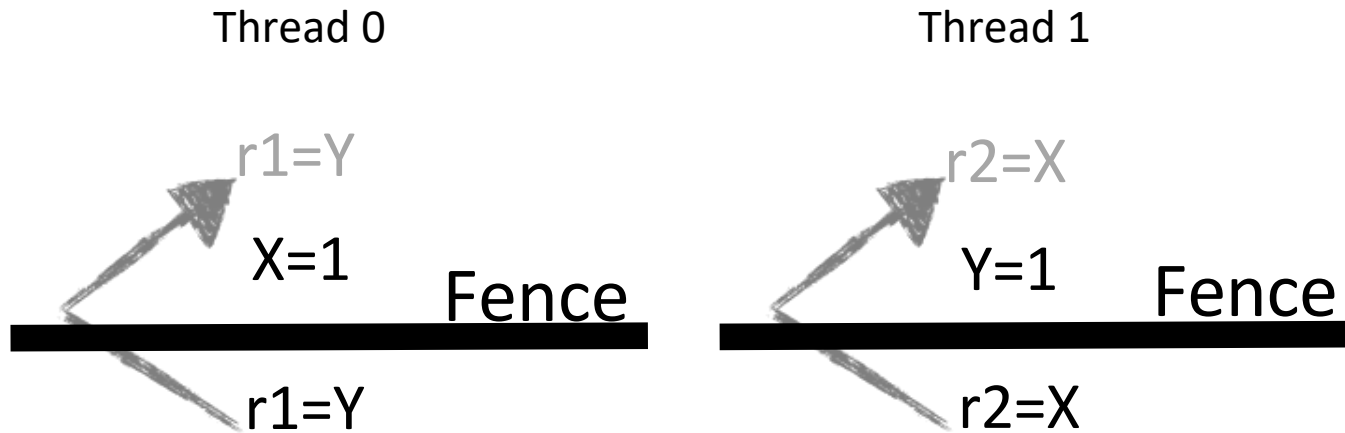
Data Races

Synchronization imposes happens-before on otherwise unordered operations



Data Race: Unordered operations to the same memory location, at least one write.

Fences are for (Preventing Re-)Ordering to Avoid Data Races & Ensure Correct Executions



We will see later that this program can produce very strange results if not synchronized

Fences are for (Preventing Re-)Ordering to Avoid Data Races & Ensure Correct Executions

Thread 0

r1=X

r1++

X=r1

Thread 1

r2=X

r2++

X=r2

What happens with this program? Where can we put the fence?

Fence



Fences are for (Preventing Re-)Ordering to Avoid Data Races & Ensure Correct Executions

Thread 0

r1=X Fence

r1++ Fence

X=r1

Thread 1

r2=X Fence

r2++ Fence

X=r2

How about fences everywhere? Does this fix our problem?

Some programs also require *atomicity*

Thread 0

```
r1=X  
r1++  
X=r1
```

Thread 1

```
r2=X  
r2++  
X=r2
```

Defining Atomicity:

Given a *critical region* that requires *atomic execution* by multiple different threads, all threads' executions of the region were atomic if the resulting execution is equivalent to some *serialization* of the atomic regions.

Fences don't provide atomicity,
but we have other primitives that
we can use for atomic operations

Fence



Some programs also require *atomicity*

Serialization #1

```
r1=X  
r1++  
X=r1
```

```
r2=X  
r2++  
X=r2
```

Serialization #2

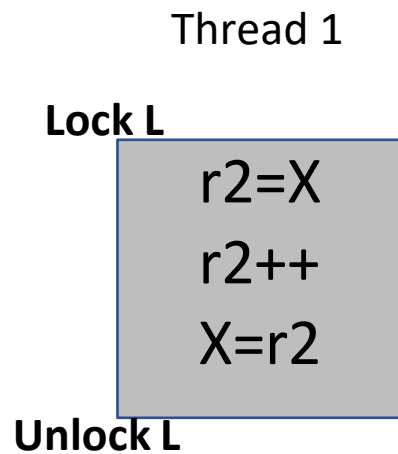
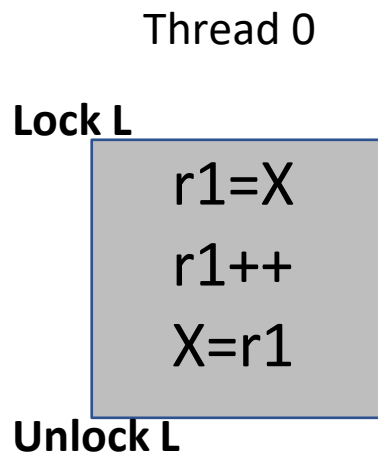
```
r2=X  
r2++  
X=r2
```

```
r1=X  
r1++  
X=r1
```

Defining Atomicity:

Given a *critical region* that requires *atomic execution* by multiple different threads, all threads' executions of the region were atomic if the resulting execution is equivalent to some *serialization* of the atomic regions.

Mutual exclusion (mutex) locks enforce *atomicity* (and ordering)

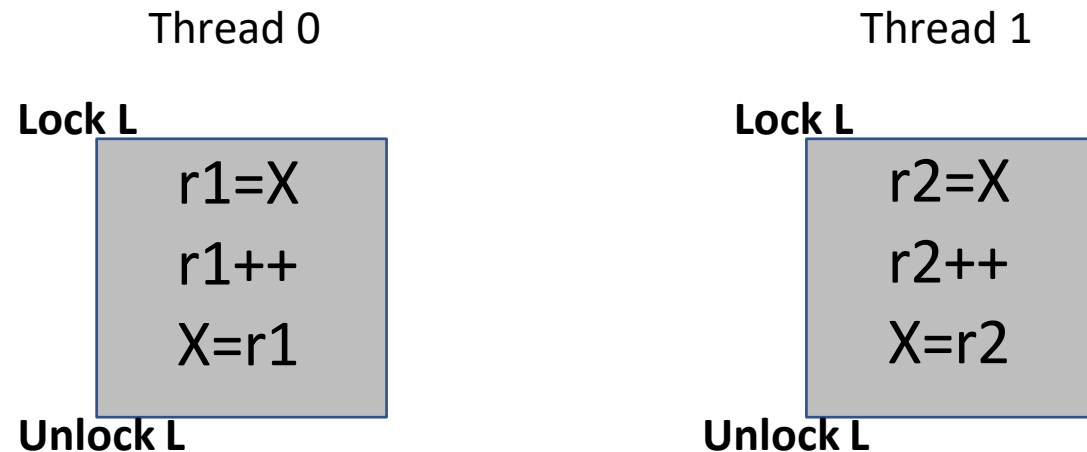


Lock Behavior:

A thread *acquires* a lock L, does stuff while *holding* L, and then *releases* lock L.

If a thread tries to acquire L while L is held, the thread keeps trying to acquire L until L is *unheld*, when its attempt to acquire succeeds.

SpinLocks are one implementation of synchronization



```
spinlock(L) {  
    while( __sync_bool_compare_and_swap(&L, 0, 1) == 0 ) {  
        /*do nothing; pause here on some systems*/  
    }  
}  
  
unlock(L) { L = 0; __sync_synchronize(); /*mem fence*/ }
```

SpinLocks are one implementation of synchronization

```
spinlock(L) {  
    while( __sync_bool_compare_and_swap(&L,0,1) == 0 ) {  
        /*do nothing; pause here on some systems*/  
    }  
}  
unlock(L) { L = 0; __sync_synchronize(); /*mem fence*/ }
```

Turtles all the way down?

SpinLocks are one implementation of synchronization

```
spinlock(L) {  
    while( __sync_bool_compare_and_swap(&L,0,1) == 0 ) {  
        /*do nothing; pause here on some systems*/  
    }  
}  
unlock(L) { L = 0; __sync_synchronize(); /*mem fence*/ }
```

```
175b: 48 8b 02      mov    (%rdx),%rax //load L into %rax  
175e: 48 8d 48 01   lea   0x1(%rax),%rcx //add 1 to %rax, into %rcx  
1762: f0 48 0f b1 0a lock cmpxchg %rcx,(%rdx) //compare & exchange  
1767: 75 f2       jne   175b //loop to mov if cmpxchg fails
```

SpinLocks are one implementation of synchronization

```
spinlock(L) {  
    while( __sync_bool_compare_and_swap(&L,0,1) == 0 ) {  
        /*do nothing; pause here on some systems*/  
    }  
}  
unlock(L) { L = 0; __sync_synchronize(); /*mem fence*/ }
```

```
1762: f0 48 0f b1 0a    lock cmpxchg %rcx, (%rdx)  
//if (%rdx) == %rax{    (%rdx) = %rcx    }
```

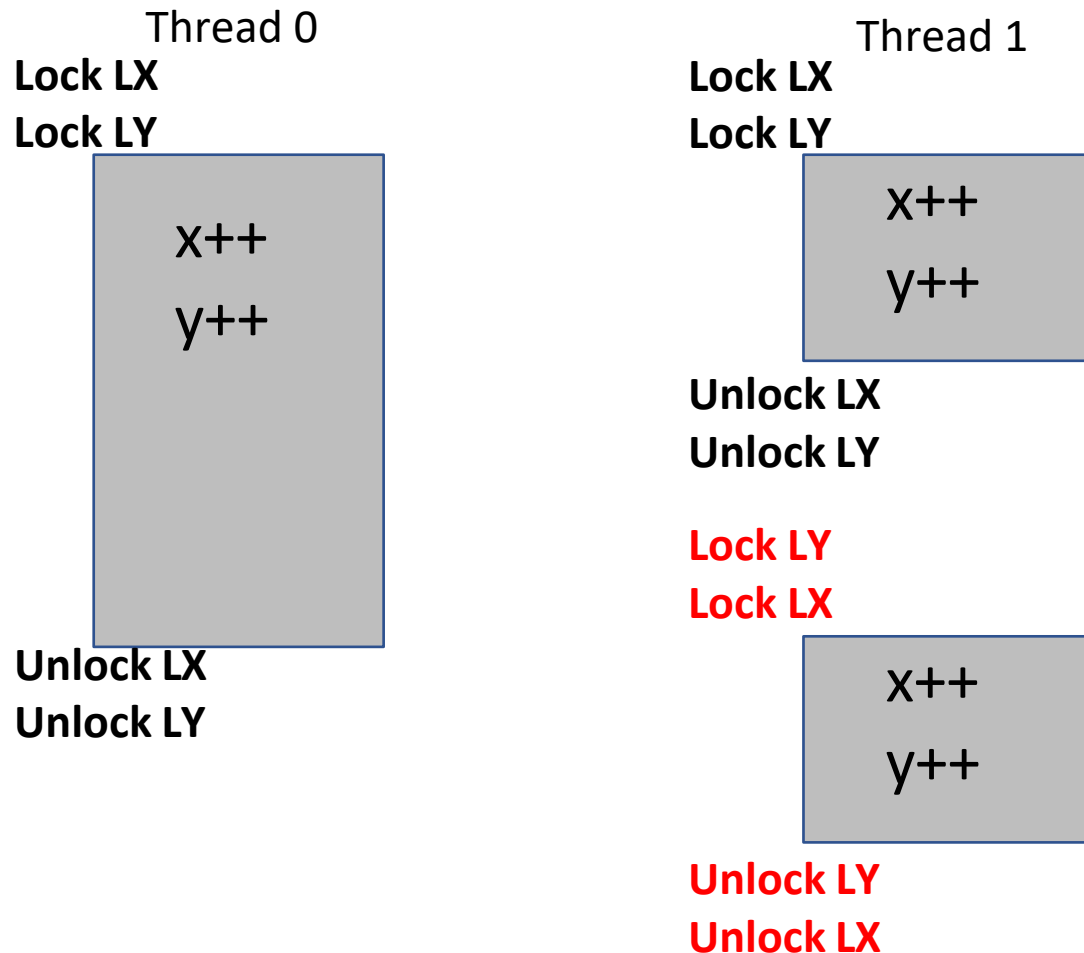
Implemented directly in the machine microarchitecture. Even if multiple threads executing, hardware guarantees *no inter-thread interactions*

SpinLocks are one implementation of synchronization

```
spinlock(L) {  
    while( __sync_bool_compare_and_swap(&L,0,1) == 0 ) {  
        /*do nothing; pause here on some systems*/  
    }  
}  
unlock(L) { L = 0; __sync_synchronize(); /*mem fence*/ }
```

1890: 0f ae f0 mfence Fence

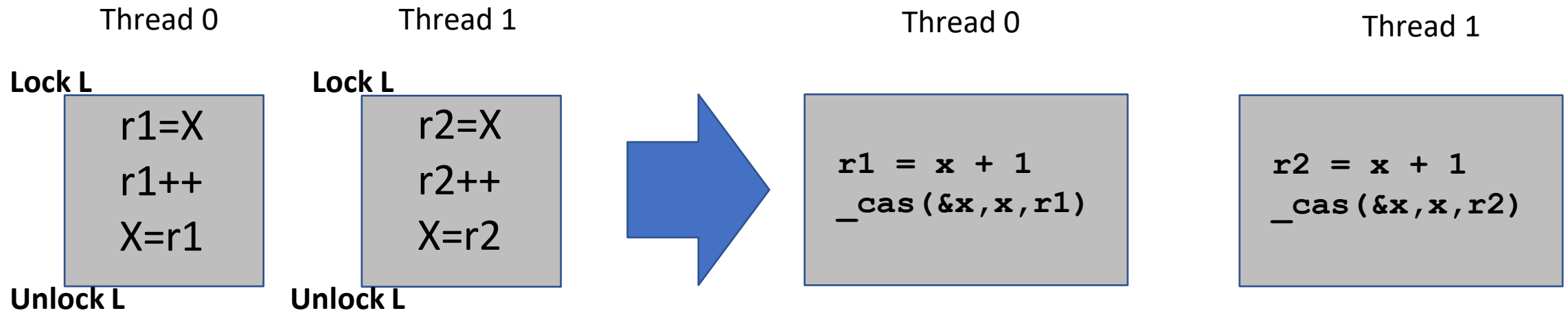
Lock ordering matters



Lock Ordering:

If you manipulate more than one piece of data in a critical region, you will need to acquire the locks in the same order for all critical regions or face **deadlock**

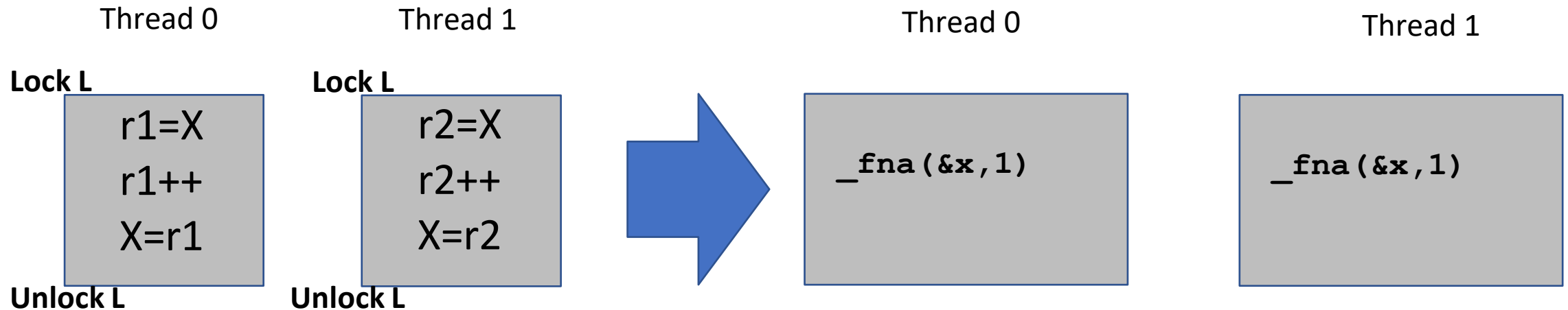
Directly Using Compare and Swap



How general is a CAS operation for implementing critical regions that need to execute atomically? What are the limitations on a CAS operation?

Fetch and Add – Further Specializing Atomics

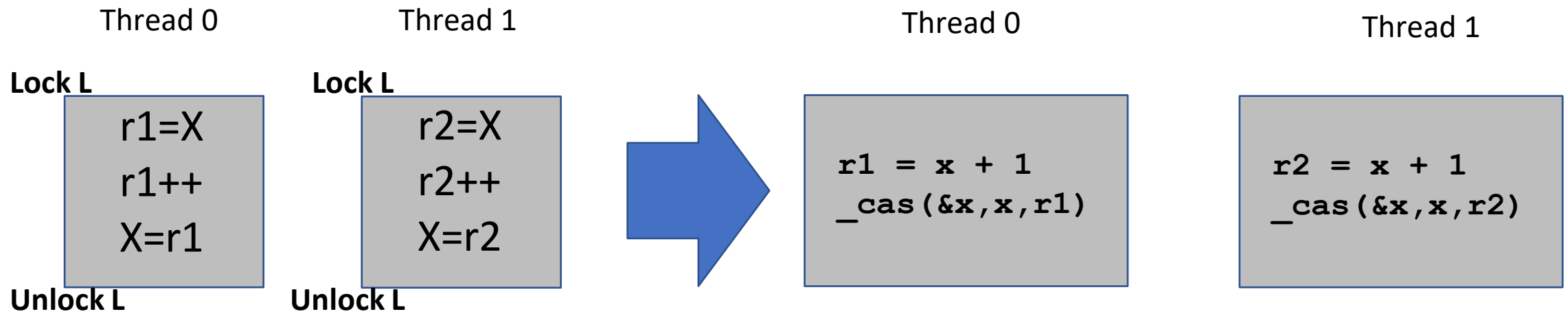
```
__sync_fetch_and_add(x, 1);
```



```
1707: f0 48 83 04 d0 01    lock addq $0x1, (%rax,%rdx,8)
```

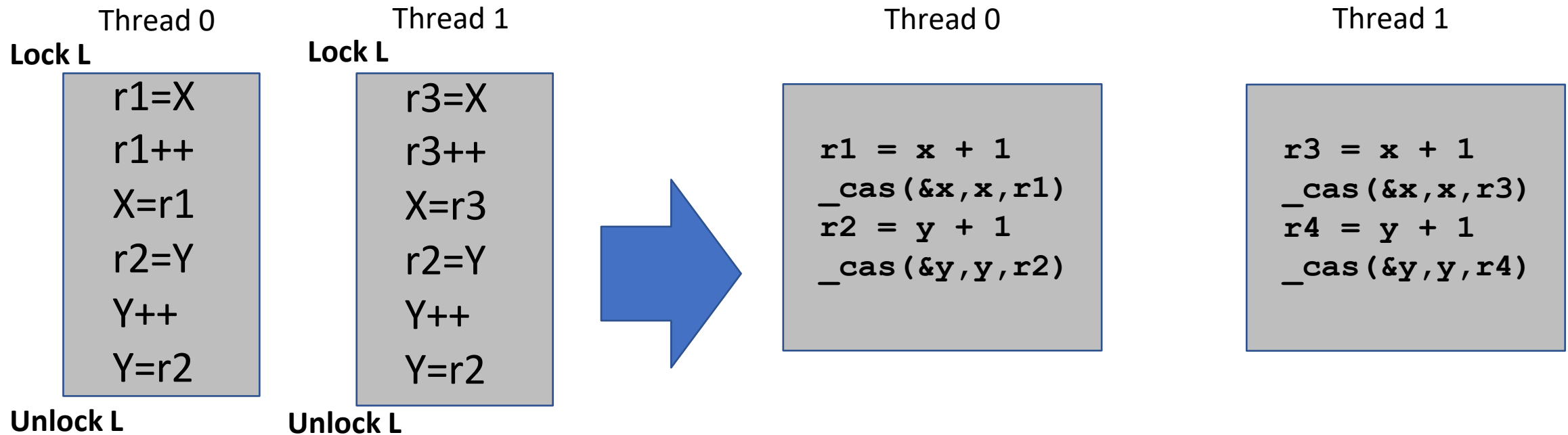
How much *less* general than compare and swap?

Transactional Memory – Further *Generalizing* Atomics



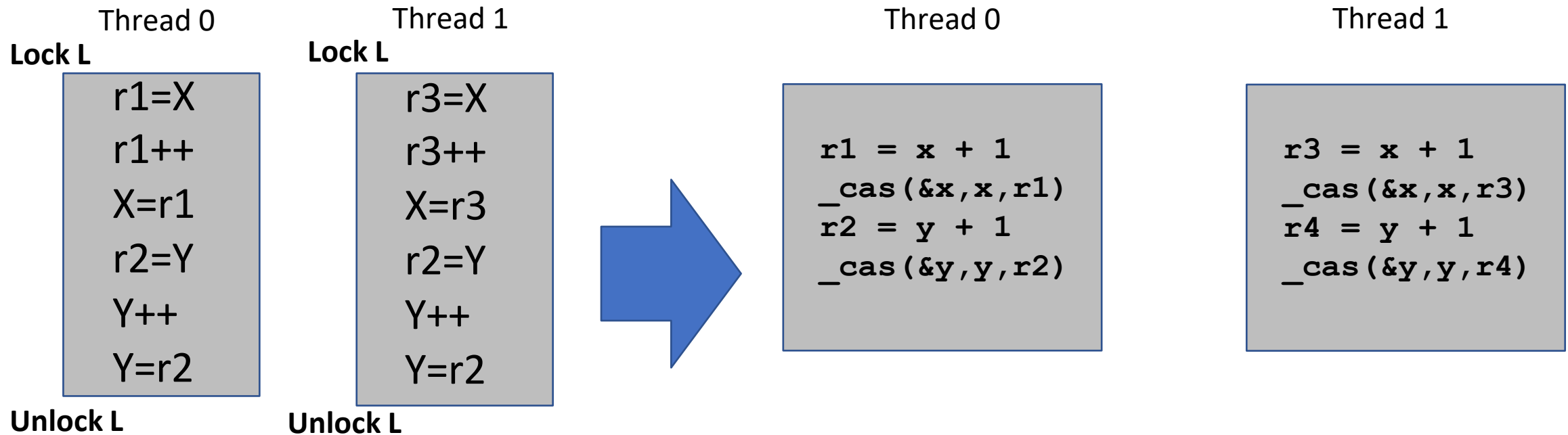
Limited by *single location* that can be updated using a CAS. What if we want to update 3 (or n) different locations (without using a lock)?

Transactional Memory – Further *Generalizing* Atomics



How about using multiple CAS operations?

Transactional Memory – Further *Generalizing* Atomics



How about using multiple CAS operations?

Problem: Need atomicity across CAS ops.

Transactional Memory: Atomicity for “n-CAS”

Thread 0

```
xbegin()  
r1 = x + 1  
r2 = y + 1  
x = r1  
y = r2  
xend()
```

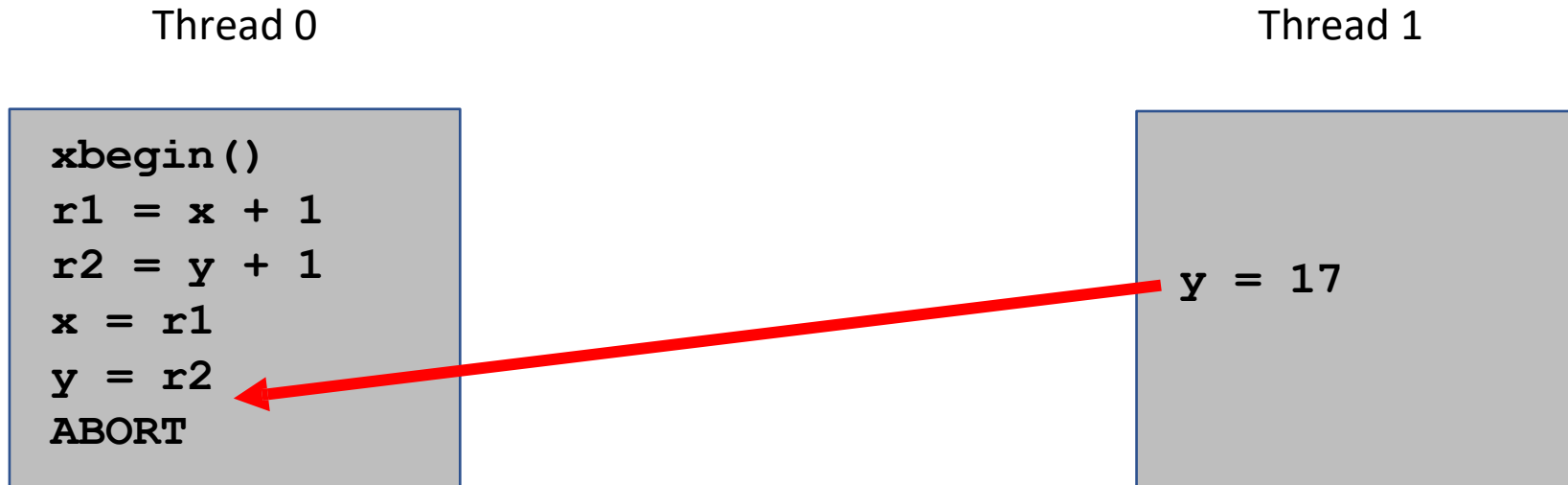
xbegin() starts a **transaction**
xend() ends the transaction
started by the most recent
xbegin()

Thread 1

```
xbegin()  
r1 = x + 1  
r2 = y + 1  
x = r1  
y = r2  
xend()
```

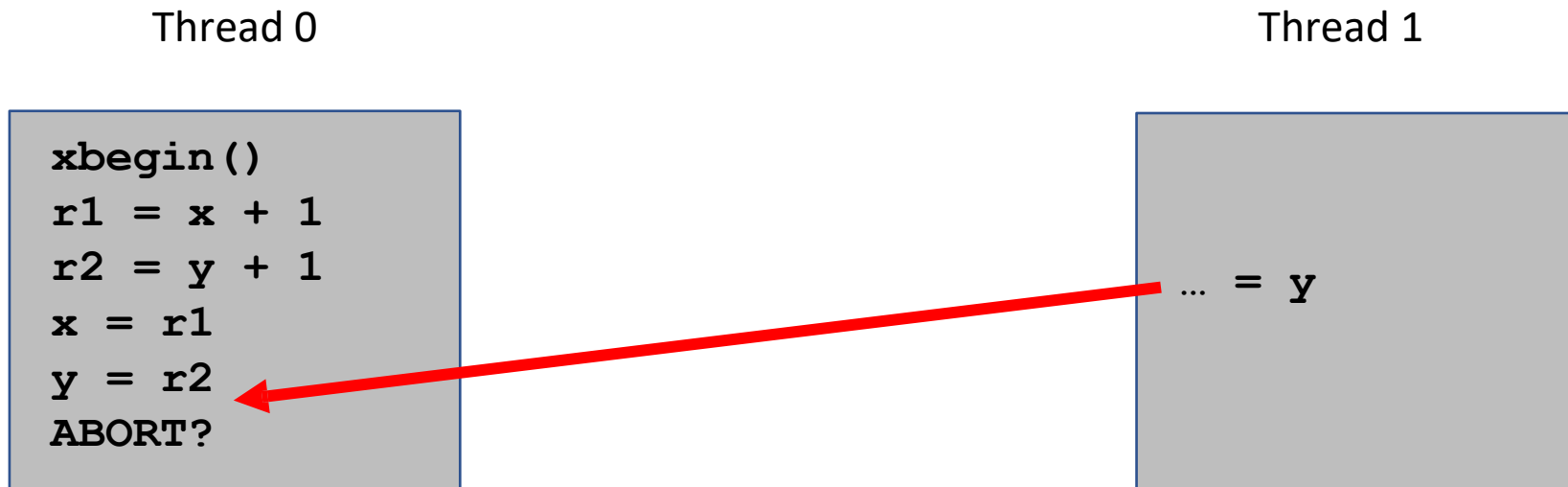
Transaction *attempts* to execute atomically,
as if protected by a lock

Transactional Memory: Atomicity for “n-CAS”



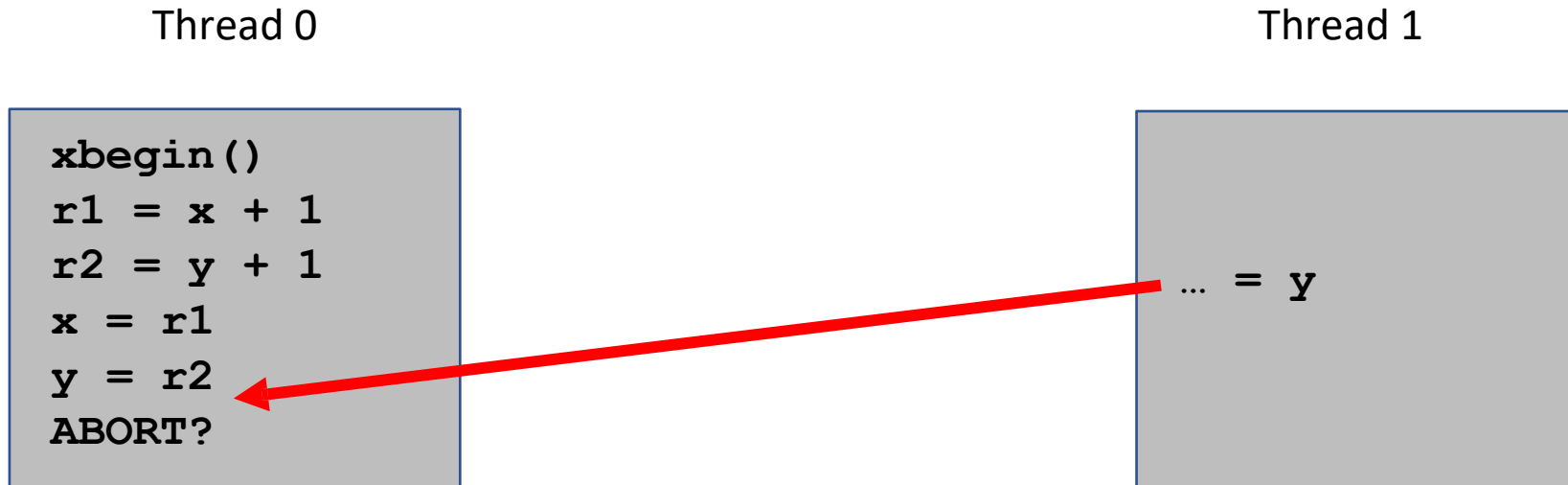
Transaction *aborts* if another thread accesses a location accessed in transaction (or if explicitly aborted)

Transactional Memory: Atomicity for “n-CAS”



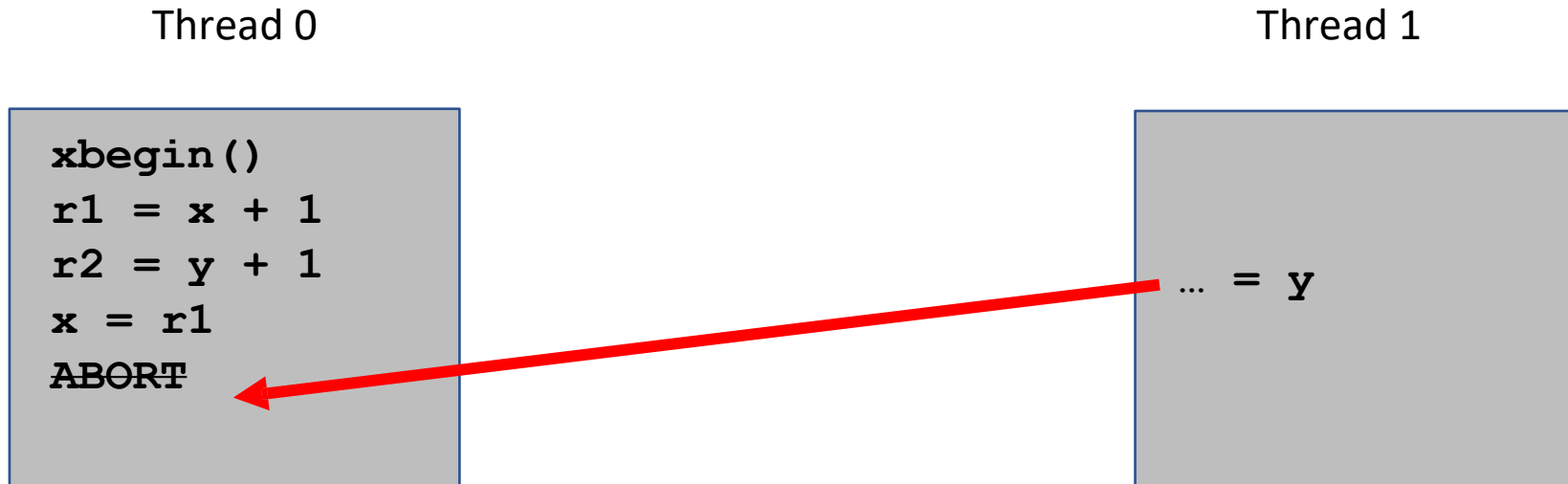
Transaction *aborts* if another thread **accesses** a location accessed in transaction (or if explicitly aborted)

Transactional Memory: Atomicity for “n-CAS”



Transaction *aborts* if another thread **reads** a location written by the transaction or **writes** a location accessed by the transaction (“Conflicting” accesses)

Transactional Memory: Atomicity for “n-CAS”



**Reads don't conflict and
transactions can read-share data**

What do we do if we have repeated aborts?

Thread 0	Thread 1
<pre>xbegin() r1 = x + 1 r2 = y + 1 x = r1 ABORT</pre>	<pre>xbegin() r1 = x + 1 r2 = y + 1 x = r1 ABORT</pre>
<pre>xbegin() r1 = x + 1 r2 = y + 1 x = r1 ABORT</pre>	<pre>xbegin() r1 = x + 1 r2 = y + 1 x = r1 ABORT</pre>
<pre>xbegin() r1 = x + 1 r2 = y + 1 x = r1</pre>	<pre>xbegin() r1 = x + 1 r2 = y + 1 x = r1</pre>

These threads are *contending for memory locations* causing repeated aborts.

How to deal with contention in a transactional memory system?

Lock-based Fallback Path

```
if(xbegin()==OK) {
    r1k =
    read_spinlock(L)
    r1 = x + 1
    r2 = y + 1
    x = r1
    xend()
}else{
//fallback
    lock(L)
    r1 = x+1
    r2 = y+1
    x = r1
    y = r2
    unlock(L)
}
```

Add a fallback path & abort handling code

Fallback should use spinlocks, not TM. **Why?**

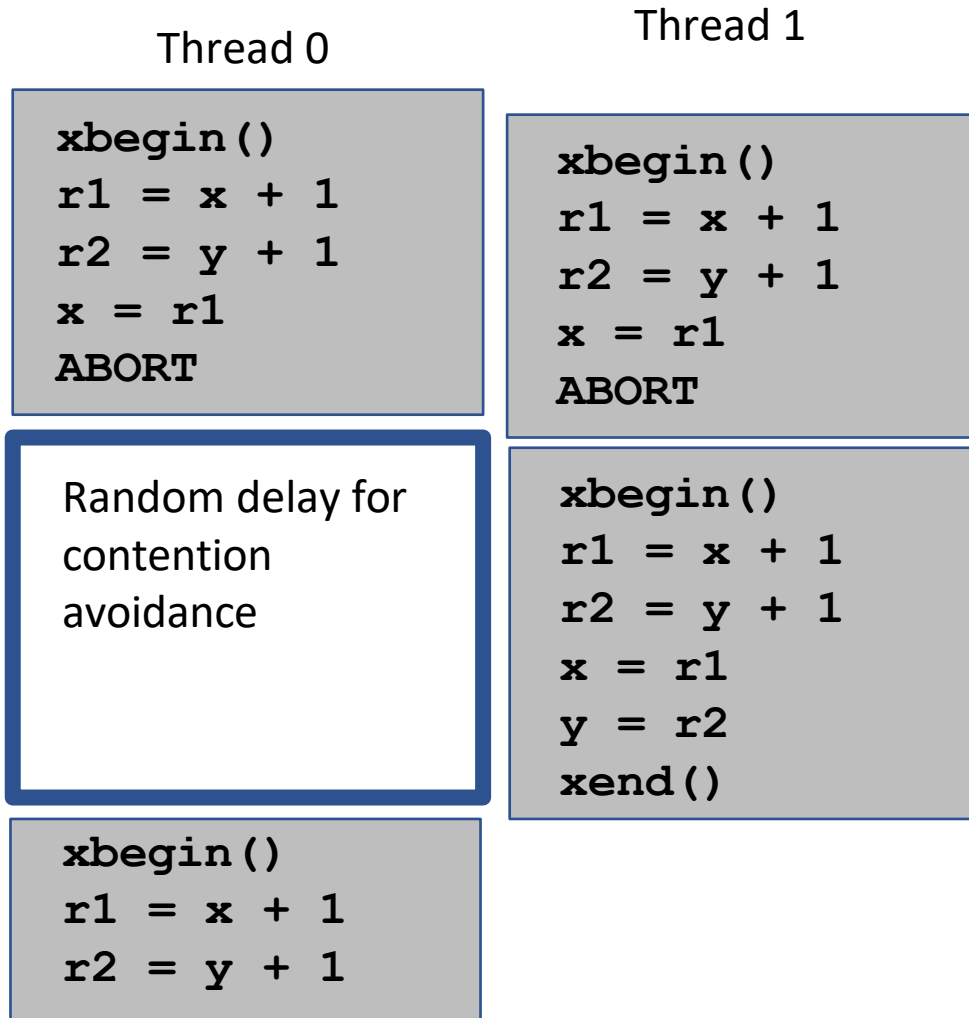
TM case needs to read spinlock lock word. **Why?**

In fallback, can do arbitrary code.

Can also retry TM version repeatedly before giving up and running fallback. Up to you the programmer what sequence to follow.

Precise Intel TSX syntax is available in the lab handout and tm.h in the lab release files.

What do we do if we have repeated aborts?



These threads are *contending for memory locations* causing repeated aborts.

How to deal with contention in a transactional memory system?

A Note About Lock-based Fallback Paths

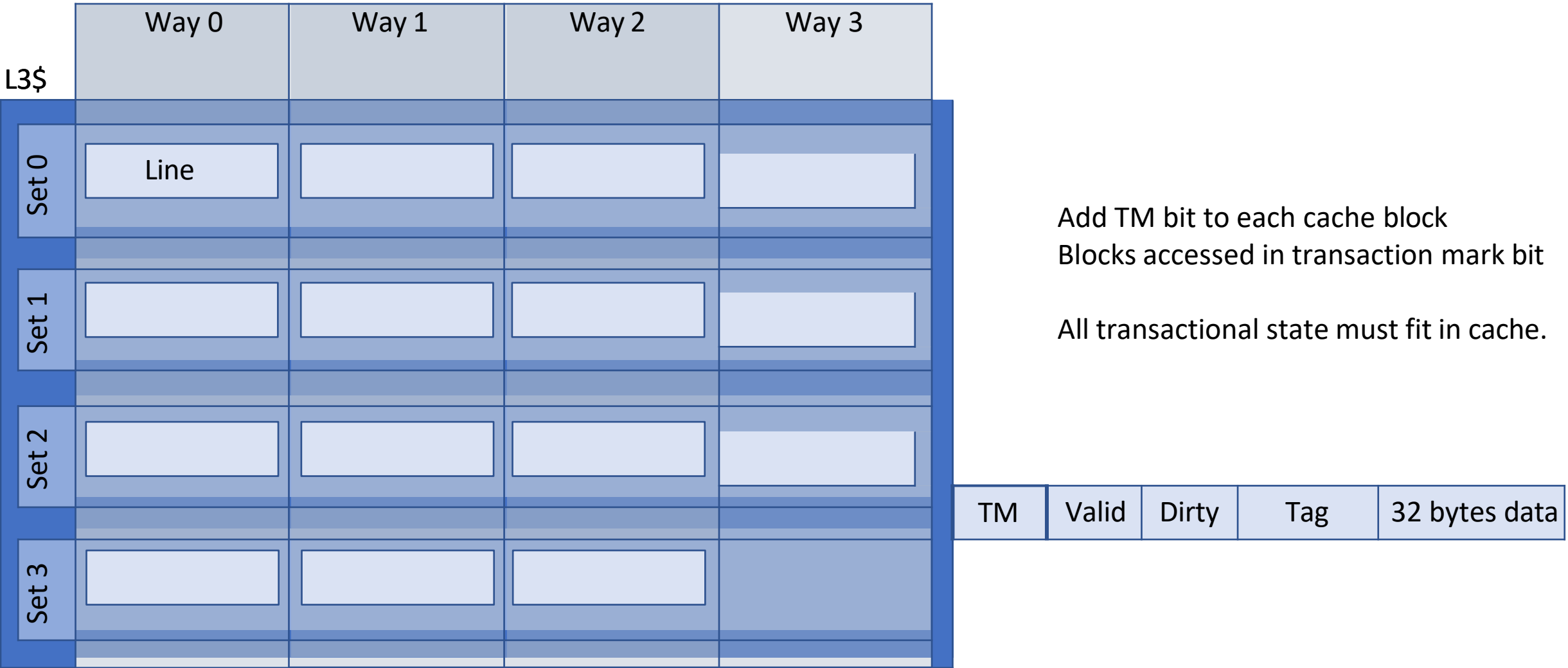
```
for(i = 0..MAX_TRIES){
  if(xbegin()){
    ...; xend(); goto done;
  } //abort code here
}
//Fallback code here
lock(Lx); lock(Ly);
r1 = x+1
r2 = y+1
x = r1
y = r2
unlock(Lx); unlock(Ly);
}

done:
//continue
```

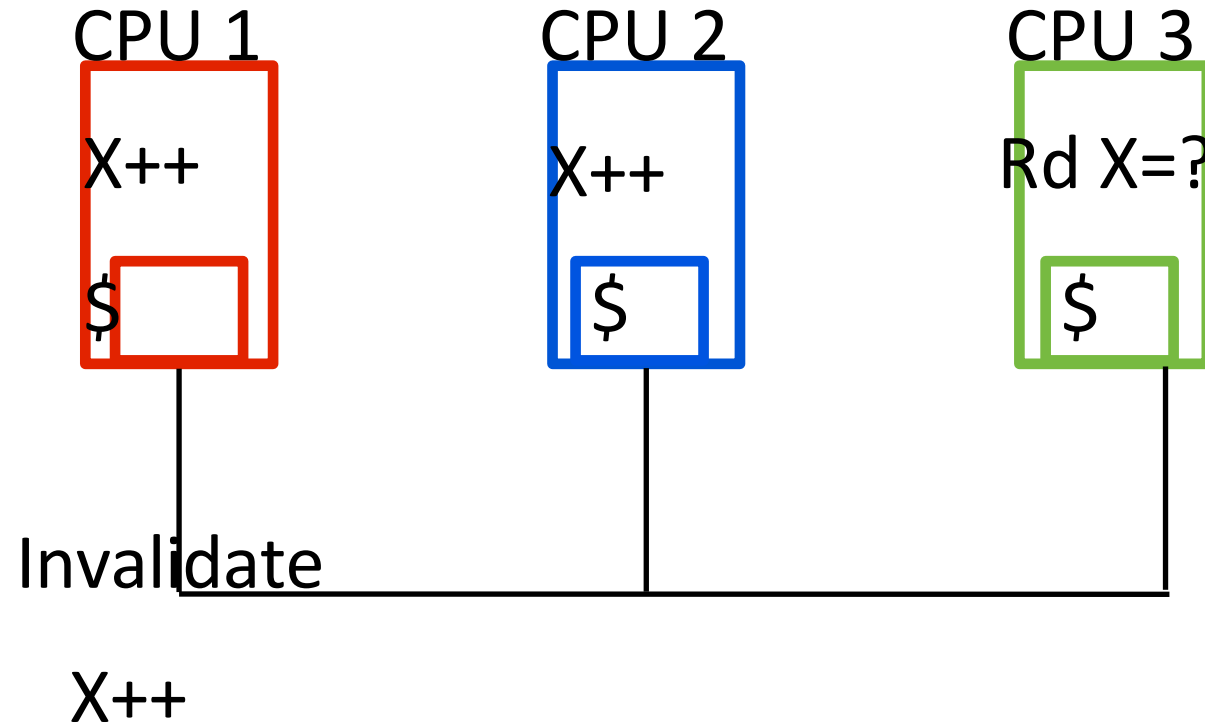
Run your transaction some number of times (MAX_TRIES)
If you commit once, skip past your fallback. Often use 'goto'...

Locks are tricky in code like this: which locks do you need to acquire? *Often need to acquire them all before you make accesses associated with locks.*

Implementation sketch of TM



Tracking TM conflicts using coherence msgs

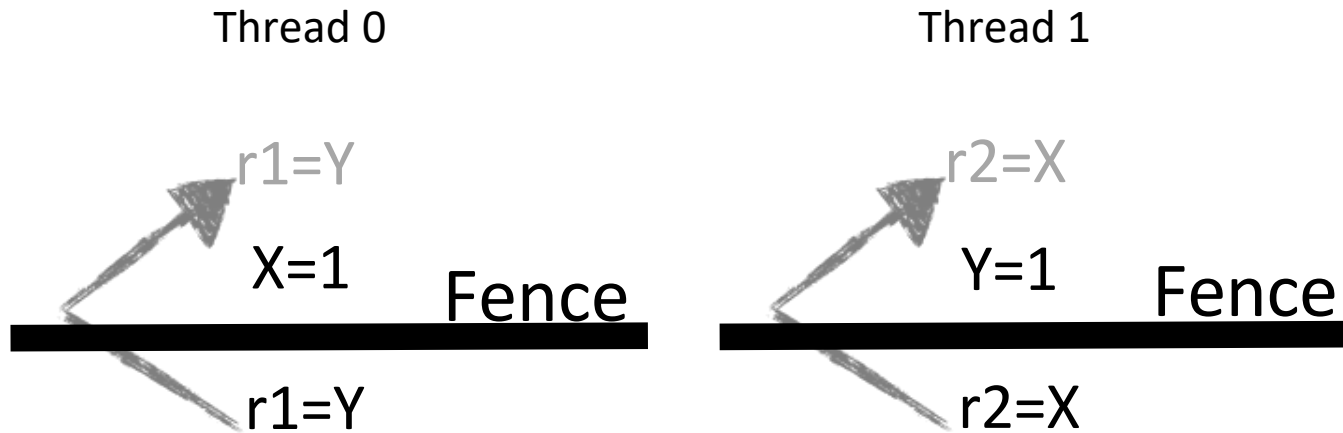


An incoming access request for a block with its TM bit set leads to a *conflict* and a transactional *abort*

Reasons a transaction might abort

- **Too many blocks with their TM bits set leaves no room for more TM blocks**
 - Too many defined as “more blocks w/ TM bits set than blocks in a way”
- **Conflict with another transaction *or non-transactional access***
 - identified through incoming coherence traffic
- **Explicit `xabort()` instruction when transactional code concludes transaction is not useful**
- **Other, unspecified, but arbitrary conditions left up to the microarchitects**
 - I speculate that these are related to internal buffers of fixed capacity

Why is this reordering situation a problem?



Reordering **independent** memory operations that access **different locations**



“computers execute operations in a **different order** than is specified by the program. A correct execution is achieved if the results produced are the same as would be produced by executing the program steps in order. For a multiprocessor computer, such a correct execution by each processor **does not guarantee the correct execution of the entire program.**”

Excerpt from “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Program”

LESLIE LAMPART, 1979



“The **memory consistency model** of a shared-memory system specifies the **order in which memory operations will appear to execute to the programmer**. The memory consistency model affects the process of writing parallel programs and forms an integral part of the entire system, including the architecture, the compiler, and the programming language.”

Excerpt from “Recent Advances in Memory Consistency Models for Hardware Shared-Memory Systems”
Sarita Adve, et al, 1999

Memory Consistency

Memory Consistency Model

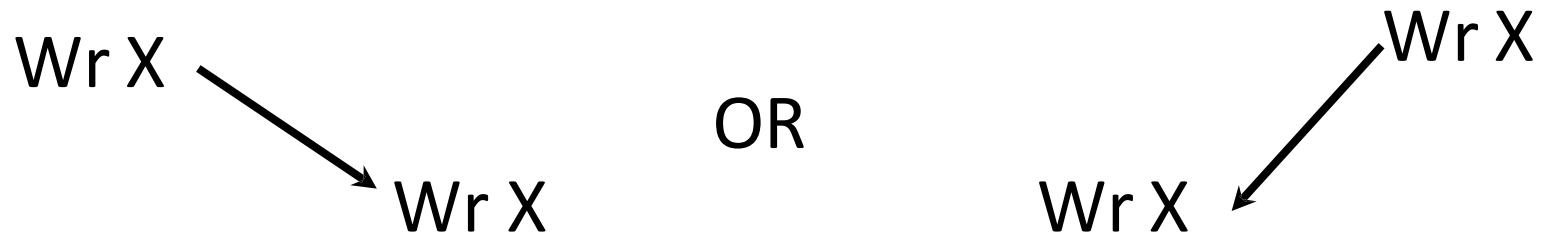
Informal Definition:

“Defines the value a read operation may read at each point during the execution”

“Defines the set of legal observable orders of memory operations during an execution”

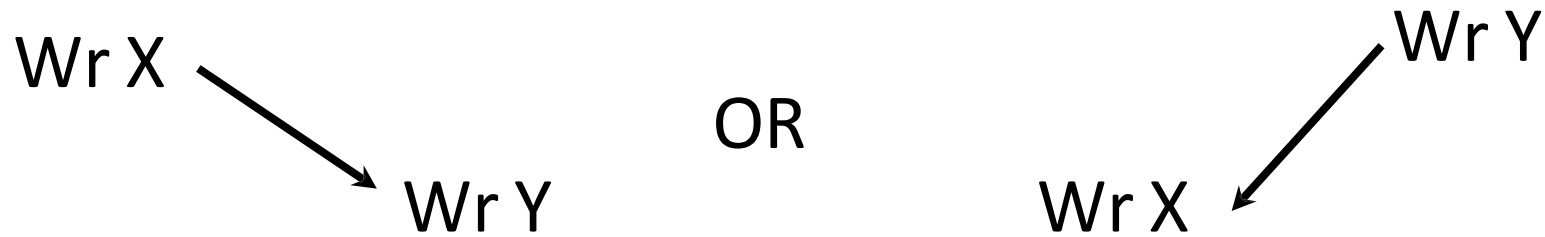
“Defines which reorderings of memory operations are permitted”

Coherence is Ordering



Coherence defines the set of legal orders of accesses to a **single** memory location

Consistency is Ordering



Consistency defines the set of legal orders of accesses to **multiple** memory locations

Sequential Consistency (SC)

The simplest, most intuitive memory consistency model

Two Invariants to SC:

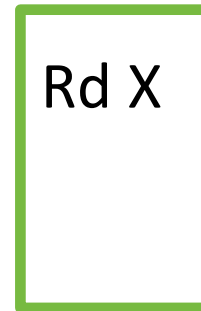
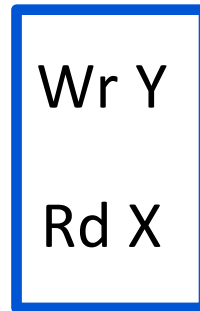
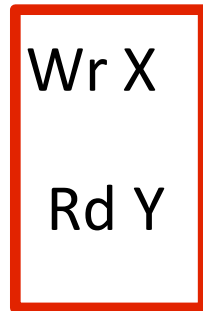
Invariant #1:

Instructions are
executed in program
order

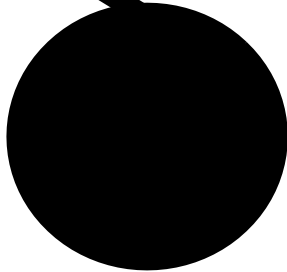
Invariant #2:

All processors agree
on a total order of
executed instructions

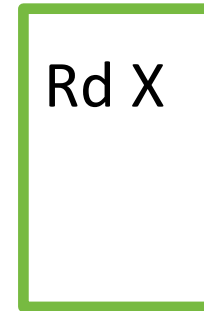
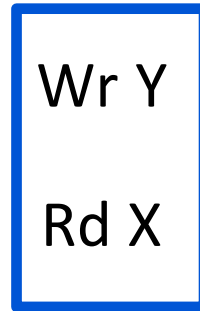
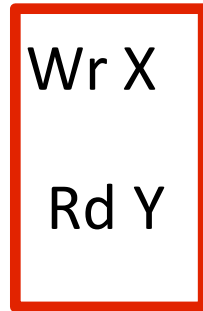
The SC "Switch"



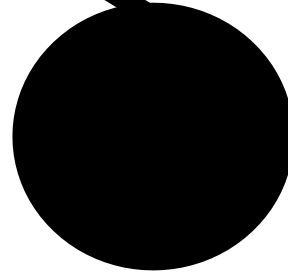
Execution



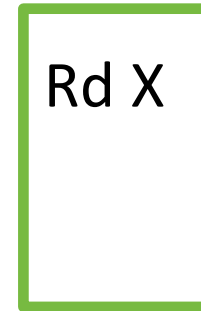
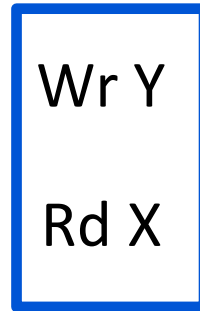
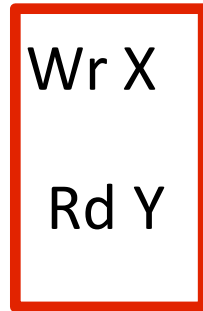
The SC "Switch"



Execution
Wr X

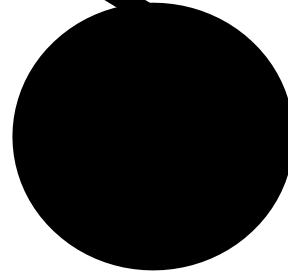


The SC "Switch"

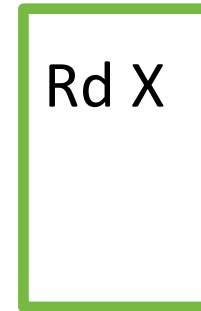
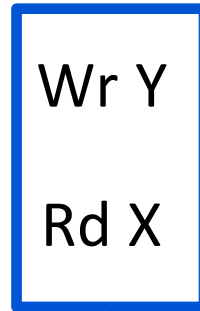
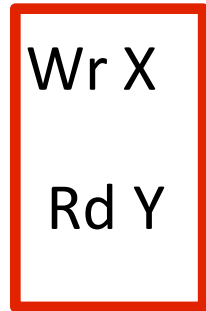


Execution

Wr X
Rd Y

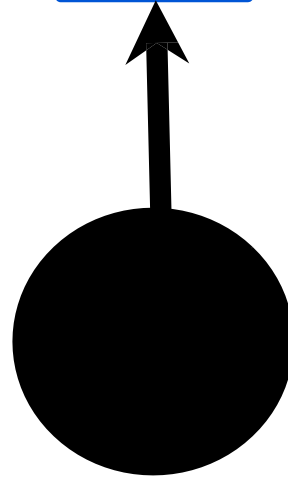


The SC "Switch"

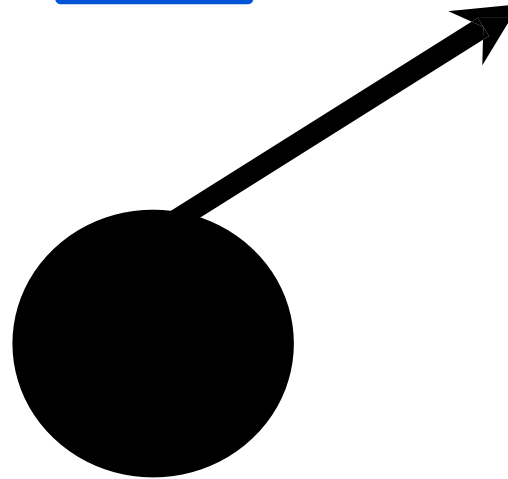
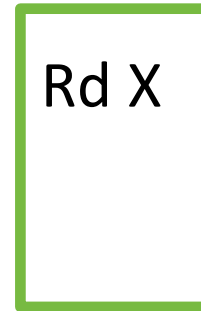
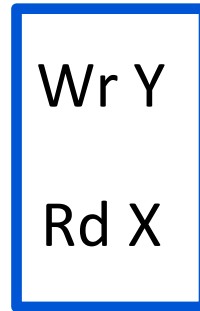
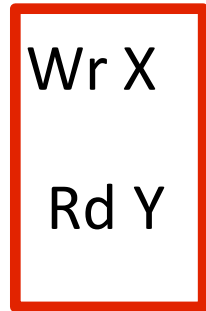


Execution

Wr X
Rd Y
Wr Y



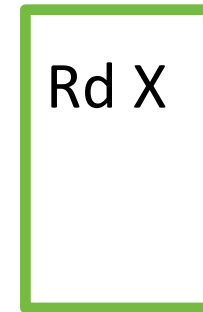
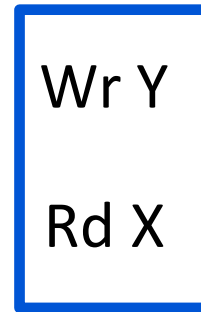
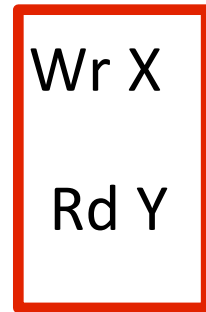
The SC "Switch"



Execution

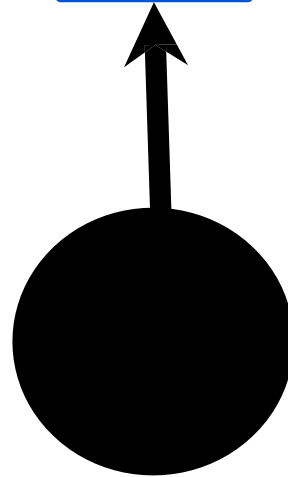
Wr X
Rd Y
Wr Y
Rd X

The SC "Switch"



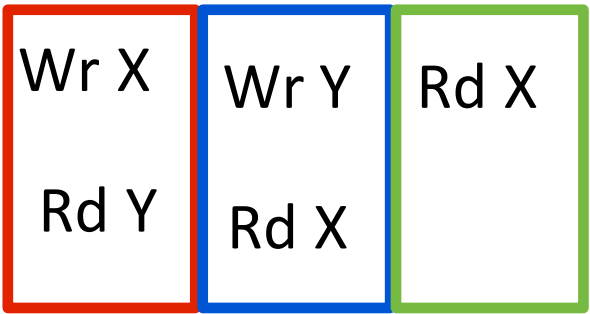
Execution

Wr X
Rd Y
Wr Y
Rd X
Rd X



Why is SC Important?

SC is the most complex model that we can ask **programmers** to think about.

	<u>Intuitive (SC)</u>	<u>Weird (not SC)</u>
	Wr X	Rd Y
	Rd Y	Wr X
	Wr Y	Rd X
	Rd X	Rd X
	Rd X	Wr Y

SC prohibits **all** reordering of instructions (Invariant 1)

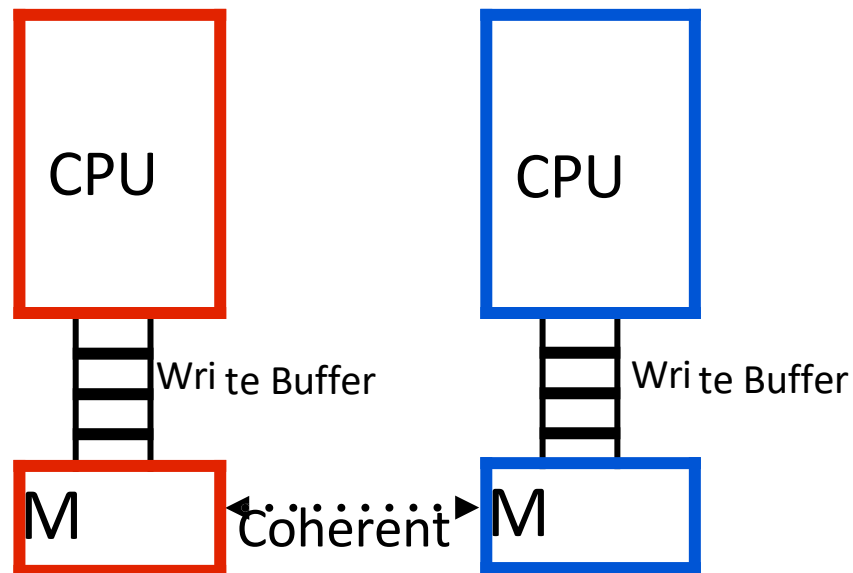
Real hardware does not enforce SC

The ARMv8 Memory Model:

The ARMv8 architecture employs a *weakly-ordered* model of memory. In general terms, this means that the order of memory accesses is not required to be the same as the program order for load and store operations. The processor is able to re-order memory read operations with respect to each other. Writes may also be re-ordered (for example, write combining). As a result, hardware optimizations, such as the use of cache and write buffer, function in a way that improves the performance of the processor, which means that the required bandwidth between the processor and external memory can be reduced and the long latencies associated with such external memory accesses are hidden.

<https://developer.arm.com/documentation/den0024/a/Memory-Ordering>

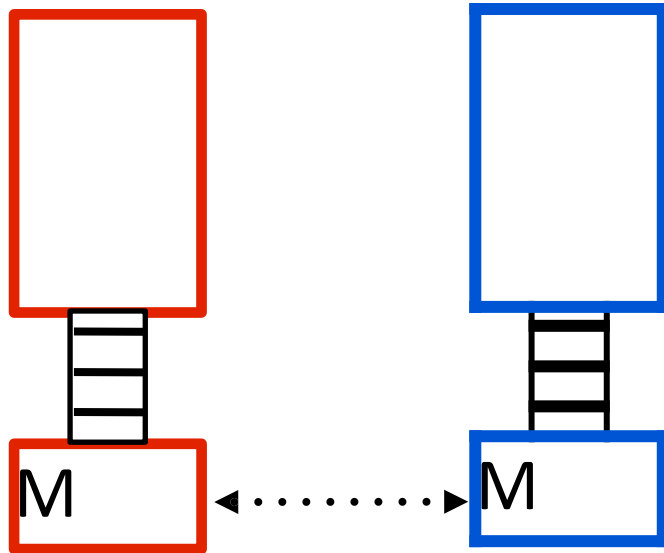
Reordering #1: Write Buffers



CPU can read its write buffer, but not others'

Buffered writes eventually end up in coherent shared memory

Reordering #1: Write Buffers



Program

Initially $X == Y == 0$

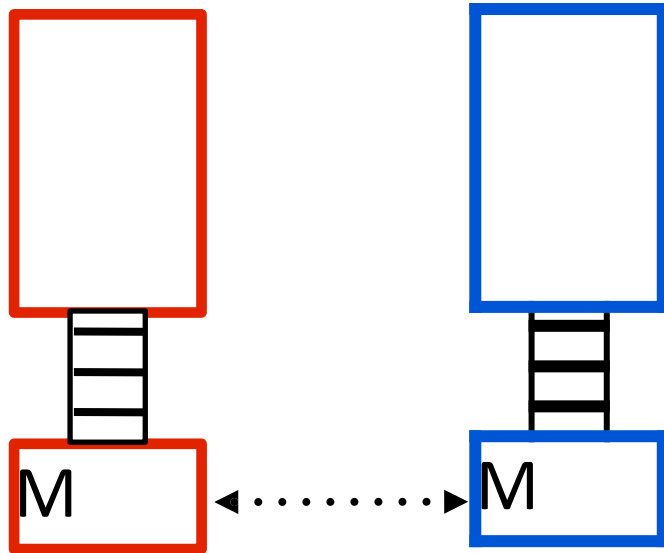
$X=1$ $Y=1$

$r1=Y$ $r2=X$

Is $r1==r2==0$

a valid result?

Reordering #1: Write Buffers



Program

Initially $X == Y == 0$

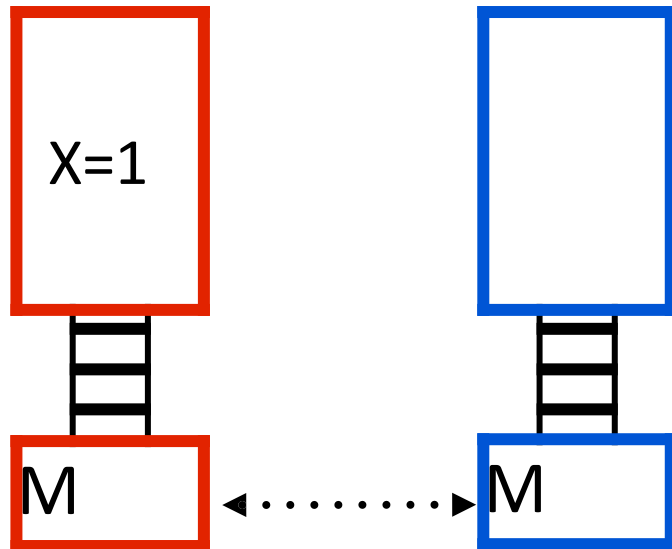
$X=1$ $Y=1$

$r1=Y$ $r2=X$

Is $r1 == r2 == 0$
a valid result?

$r1 == r2 == 0$ is **not** SC, but it can happen with write buffers

Reordering #1: Write Buffers



Program

Initially $X == Y == 0$

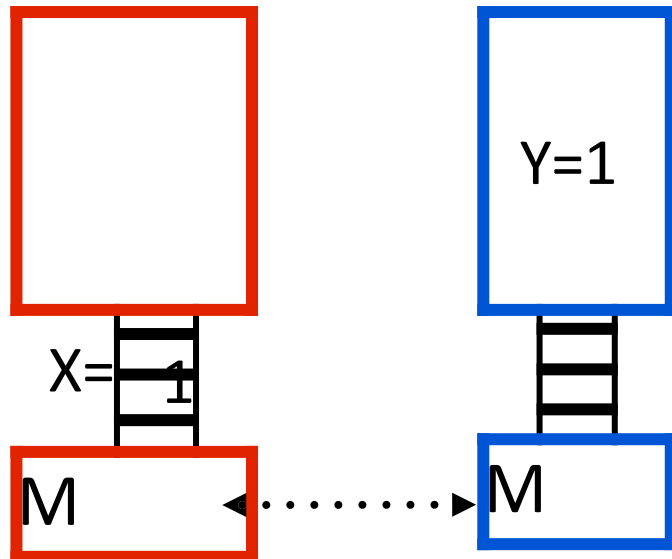
$Y = 1$

$r1 = Y$

$r2 = X$

Execution

Reordering #1: Write Buffers



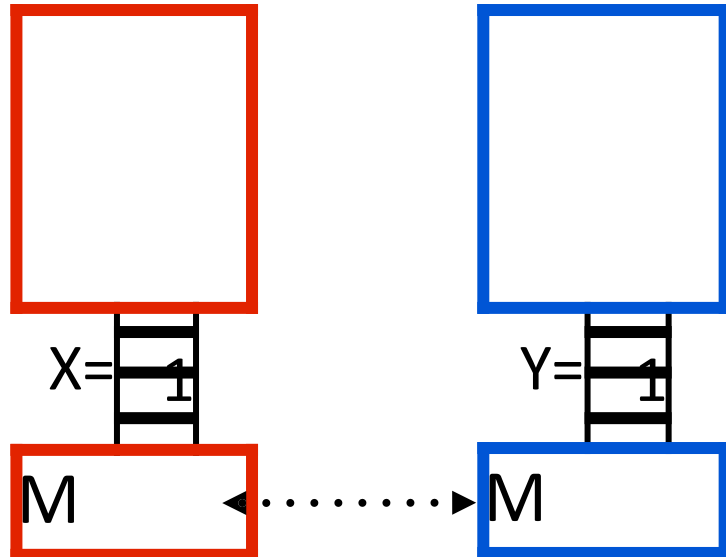
Program

Initially $X == Y == 0$

$r1=Y$ $r2=X$

Execution

Reordering #1: Write Buffers



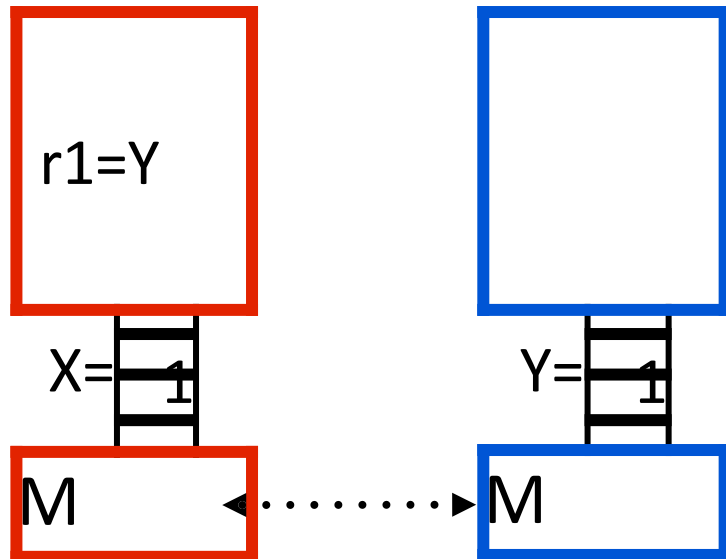
Program

Initially $X == Y == 0$

$r1=Y$ $r2=X$

Execution

Reordering #1: Write Buffers



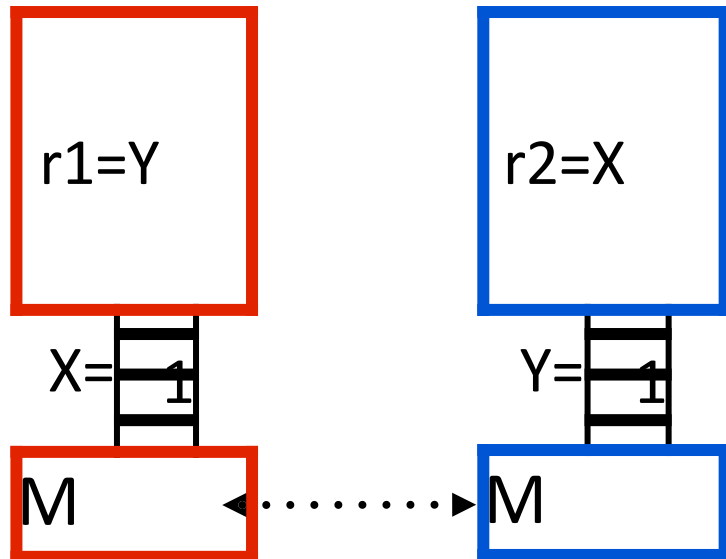
Program

Initially $X == Y == 0$

$r2=X$

Execution

Reordering #1: Write Buffers

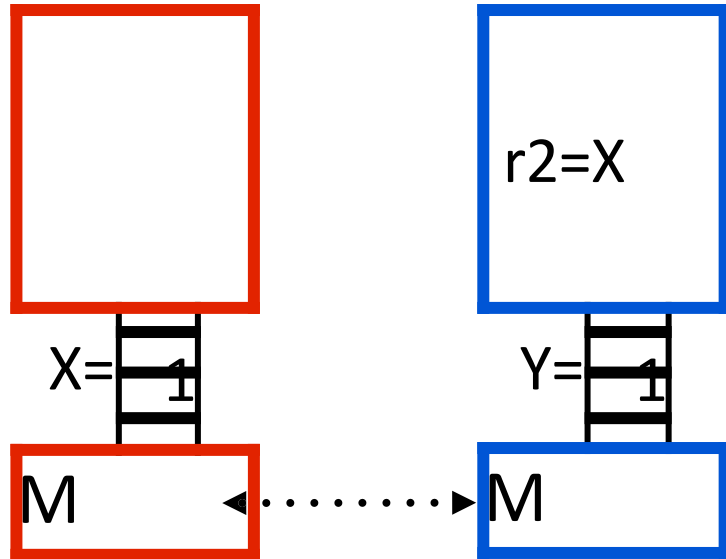


Program

Initially $X == Y == 0$

Execution

Reordering #1: Write Buffers



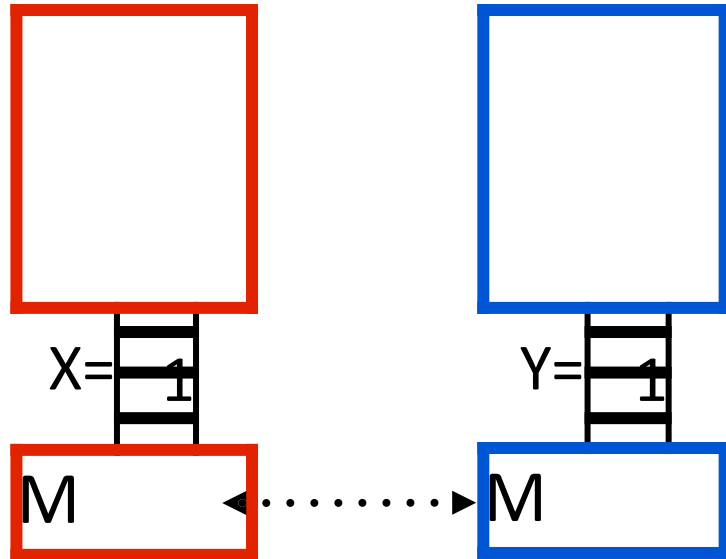
Program

Initially $X == Y == 0$

Execution

$r1=Y$ [$r1 \leftarrow 0$]

Reordering #1: Write Buffers



Program

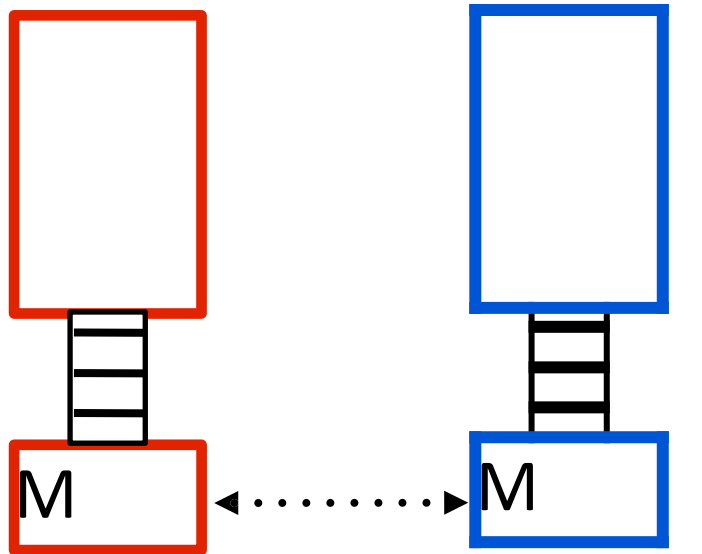
Initially $X == Y == 0$

Execution

$r1=Y$ [$r1 \leftarrow 0$]

$r2=X$ [$r2 \leftarrow 0$]

Reordering #1: Write Buffers



WBs let reads finish
before older writes

Program

Initially $X == Y == 0$

Execution

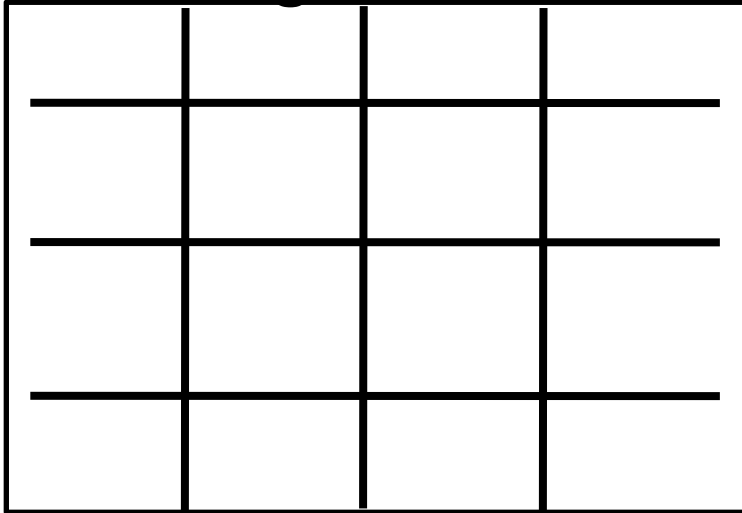
$r1=Y$ [$r1 \leftarrow 0$]

$r2=X$ [$r2 \leftarrow 0$]

$X=1$ (Not SC!)
 $Y=1$

Reordering #2: Write Combining

Coalescing Write Buffer



4 word cache line

Program

X,Z in same \$ line

X=1

Y=1

Z=1

Reordering #2: Write Combining

Coalescing Write Buffer

X=1			

Program

X,Z in same \$ line

X=1

Y=1

Z=1

Reordering #2: Write Combining

Coalescing Write Buffer

X=1			
			Y=1

Program

X,Z in same \$ line

X=1

Y=1

Z=1

Reordering #2: Write Combining

Coalescing Write Buffer

X=1			
			Y=1
	Z=1		

Program

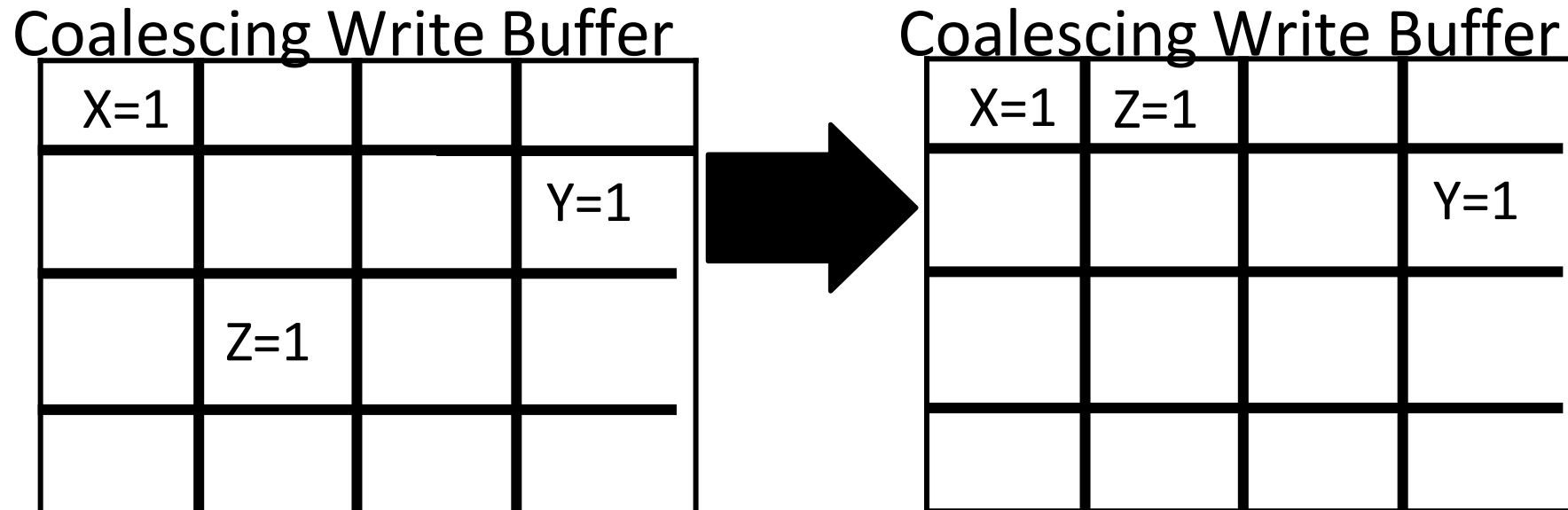
X,Z in same \$ line

X=1

Y=1

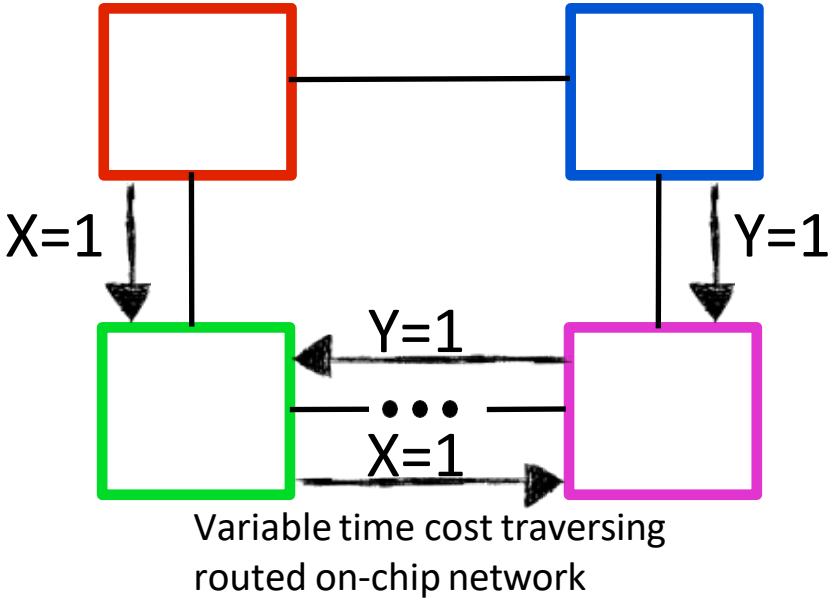
Z=1

Reordering #2: Write Combining



Combining the write to X & Z saves bandwidth,
but **reorders** Z=1 and Y=1

Reordering #3: Interconnect



X=1

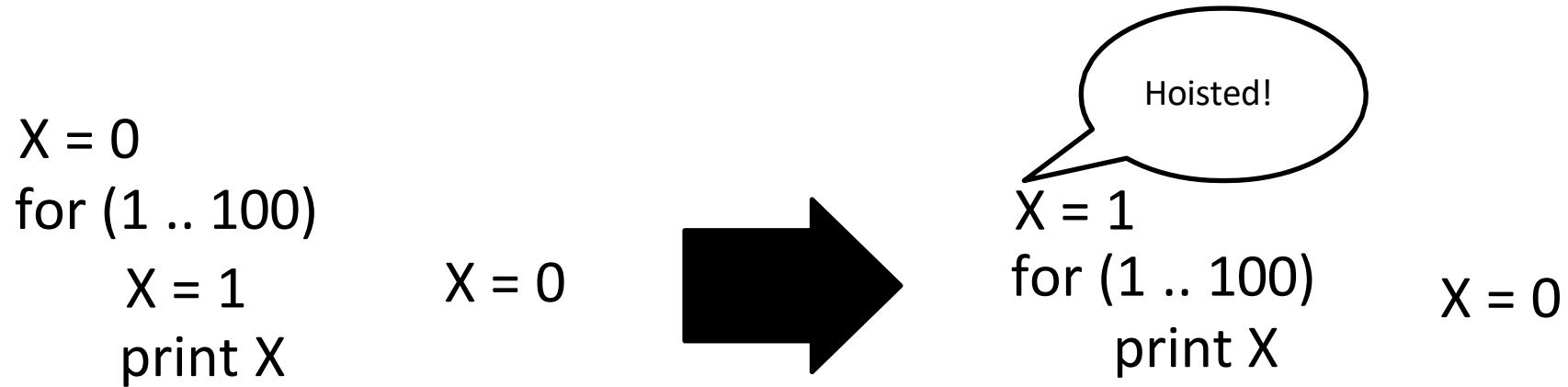
Program

r1=X Y=1 r3=Y
r2=Y r4=X

Execution

X=1
Y=1
r1=X [r1 <- 1]
r2=Y [r2 <- 0]
r3=Y [r3 <- 1]
r4=X [r4 <- 0]

Reordering #4: Compilers



The compiler hoists the write out of the loop, permitting new (non-SC) results (e.g., “1 0 0 0 0 0 0...”)

When is an Execution Not SC?

When a memory operation happens before itself

Execution

r1=Y [r1 <- 0]

r2=X [r2 <- 0]

X=1

Y=1

Happens-Before Graph

X=1

Y=1

r1=Y

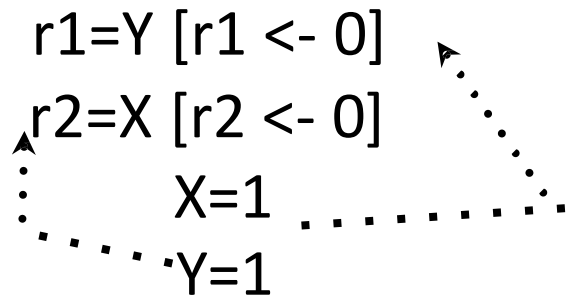
r2=X

When is an Execution Not SC?

When a memory operation *happens before* itself

Execution

r1=Y [r1 <- 0]
r2=X [r2 <- 0]
X=1
Y=1



Happens-Before Graph

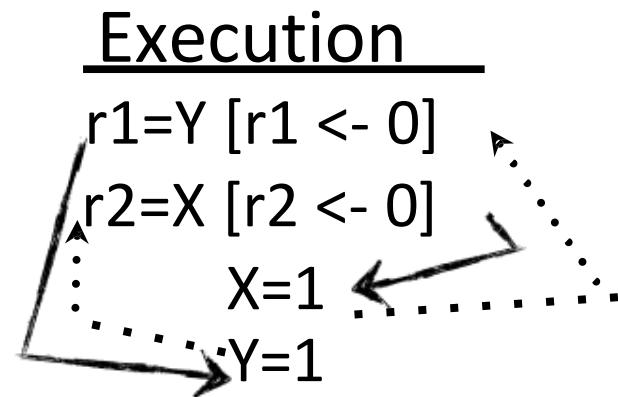
X=1	Y=1
⋮	⋮
↓	↓
r1=Y	r2=X

Program Order HB Edge

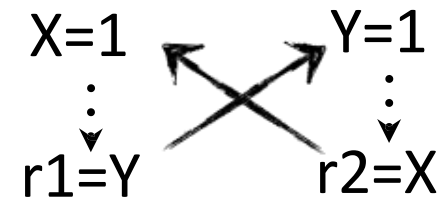


When is an Execution Not SC?

When a memory operation happens before itself



Happens-Before Graph

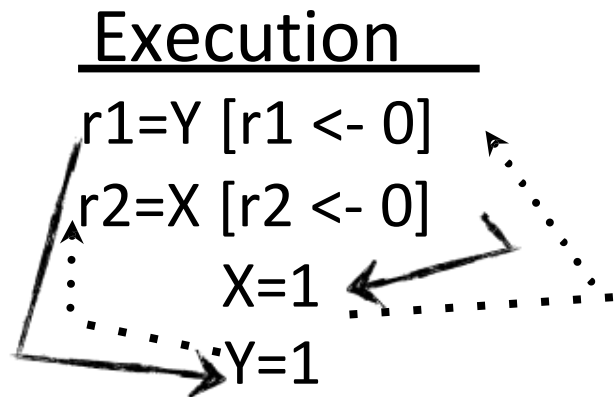


⋮ Program Order HB Edge

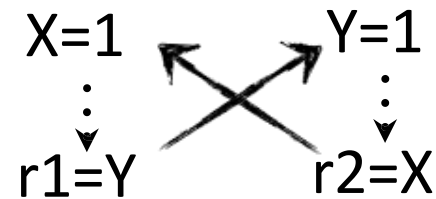
↓ Causal Order HB Edge

When is an Execution Not SC?

When a memory operation happens before itself



Happens-Before Graph



If there is a cycle in the happens-before graph, the execution is not SC

Two Design Constraints at Odds

SC is how **programmers** think, but restricts all reordering

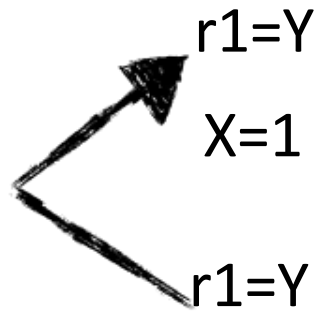
Reordering allows **optimization**, but leads to unintuitive non-SC behavior.

Relaxed Memory Consistency

Relaxed Memory Models permit reorderings, unlike SC

x86-TSO (intel x86s)

“The Write Buffer Memory Model”

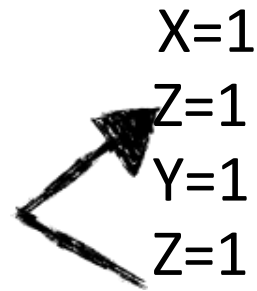


Relaxes W->R
order

Total Store Order - loads may complete before older stores to different locations complete.

PSO_(SPARC)

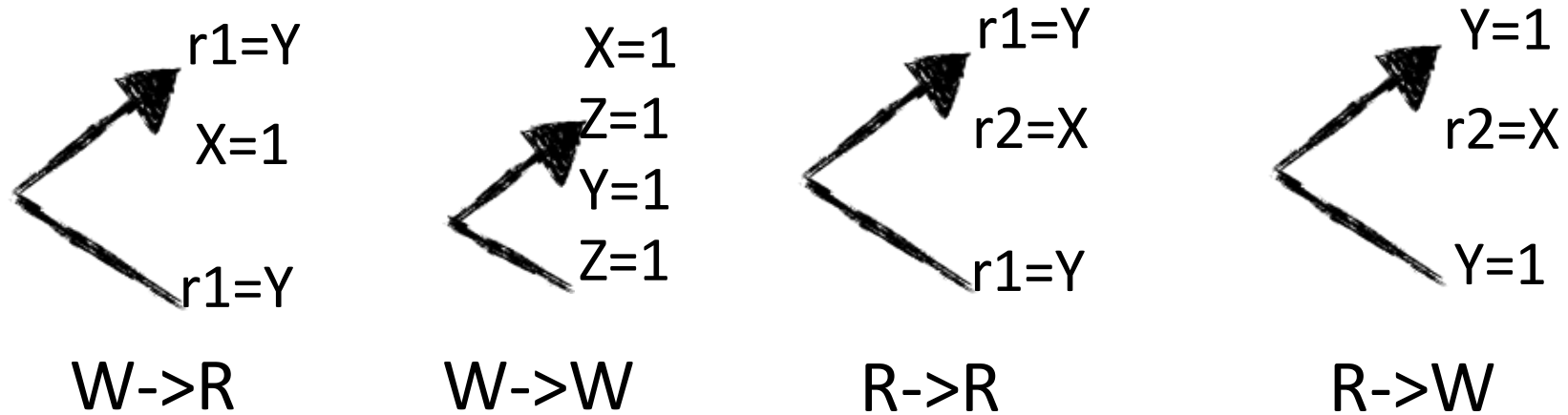
“The Write Combining Memory Model”



Relaxes W->W
order

Partial Store Order - loads and stores may complete before older stores to different locations complete.

In General



Starting with PSO and relaxing R->R and R->W yields
Weak Ordering or Release Consistency (alpha)

Depending on the implementation

Implementing Synchronization for Weak Memory Models

- What does synchronization have to do to prevent SC violations?
 - Flush WB, prevent coalescing/bypassing, impose ordering in network, prevent compiler reorderings
- What does synchronization have to do to prevent other kinds of problems?
 - Enforce mutually exclusive execution by different threads of critical region, force threads to wait at barriers, enforce wait/notify discipline

SC and Relaxed Consistency

SC is required for correctness and programmer sanity

+

Reordering is required* for performance

Goal: Ensure SC executions while permitting
Relaxed Consistency reorderings

*Usually; MIPS memory model is **SC**

Memory Models across the System Stack

Language

Java/C++: SC
for data-race-
free programs

Compiler

Conservative
with reordering
when d-r-f can't
be proved

Architecture

Usually very weak for
max optimization
(lots of reordering)

Note: fences from
“above” ensure SC

What did we just learn?

- Concurrency and parallelism, from the bottom to the top
- Coherence and consistency are both memory ordering principles
- Synchronization exists to spare you data-races and non-SC executions
- Transactional memory is a powerful sync primitive in many x86 CPUs