CS302

Language-Level Consistency

Spring 2025

Arkaprava Basu & Babak Falsafi

parsa.epfl.ch/course-info/cs302

Adapted from slides originally developed by Profs. Falsafi, Patterson, Wenisch, Fatahalian of CMU/EPFL, UC Berkeley, Michigan, and CMU

Where are We?

M	T	W	T	F
17-Feb	18-Feb	19-Feb	20-Feb	21-Feb
24-Feb	25-Feb	26-Feb	27-Feb	28-Feb
3-Mar	4-Mar	5-Mar	6-Mar	7-Mar
10-Mar	11-Mar	12-Mar	13-Mar	14-Mar
17-Mar	18-Mar	19-Mar	20-Mar	21-Mar
	25-Mar	26-Mar	27-Mar	28-Mar
31-Mar	1-Apr	2-Apr	3-Apr	4-Apr
7-Apr	8-Apr	9-Apr	10-Apr	11-Apr
14-Apr	15-Apr	16-Apr	17-Apr	18-Apr
21-Apr	22-Apr	23-Apr	24-Apr	25-Apr
28-Apr	29-Apr	30-Apr	1-May	2-May
5-May	6-May	7-May	8-May	9-May
12-May	13-May	14-May	15-May	16-May
19-May	20-May	21-May	22-May	23-May
26-May	27-May	28-May	29-May	30-May

- Language-Level Consistency
  - Data Race Free
  - Case studies: Java, C++, Rust, and Go
- Thursday
  - Synchronization
- Friday lab session
  - MPI programming assignment

Assignment 2: Message Passing Programming

- Assignment 2 released!
  - Deadline: May 4th, 23:59
- Objectives:
  - Learn how to write programs using message passing
  - Compare and combine with shared memory
- Group list has been updated on Moodle
  - People whose partners left the course have been reassigned
    - Remains the same for others
  - Please check and get in touch with your new group partners!

Homework 4 and Sample Midterm Exam

- Homework 4 has been released
  - Deadline: March 30th, 23:59
- Homework 3 grades will be released this week
  - Feedback will be available on Moodle, ask TA if not found
- Sample midterm exam has been released
  - Midterm from CS307 (the predecessor to this course)
  - The coverage is not identical (CS307 was a 4-credit course)
  - Try to solve on your own first, solutions will be made available from Thursday
  - TA will go over some of the exam questions during Thursday's exercise session

”Grand Compromise” with Languages

- Programmers must write “correctly synchronized” or “well-labelled” code, containing no **data races**
  - Then, language, runtime, and hardware will **guarantee** SC

- This is called “Data Race Free = SC”
  - Removes burden from programmers to directly insert fences and barriers into their code

What is a Data Race?

- First, specify a minimal “memory operation”
  - Indivisibly accesses a certain address (i.e., not as two separate memory operations)
- Two operations conflict if they access the same location, and at least one is a write
- A data race is when:
  - Two conflicting operations from different threads occur simultaneously
  - Simultaneous operations are defined as back-to-back accesses in any sequentially consistent interleaving

Data Race (Free?) Examples

- Does our first example have a data race?

Thread 0

```
// A = r1 = 0

(S0) A = 1;
(L0) r1 = B;

print(r1);
```

Thread 1

```
// B = r2 = 0

(S1) B = 1;
(L1) r2 = A;

print(r2);
```

Data Race (Free?) Examples

- Does our first example have a data race?
  1. Identify the locations: { A, B }
  2. Identify the reads: **(L_1) (L_0)**
  3. Identify the writes: **(S_1) (S_0)**
  4. Are there SC executions where R/W to A or B are back to back?

Thread 0

```
// A = r1 = 0

(S0) A = 1;
(L0) r1 = B;

print(r1);
```

Thread 1

```
// B = r2 = 0

(S1) B = 1;
(L1) r2 = A;

print(r2);
```

Data Race (Free?) Examples

- Does our first example have a data race?
  - Are there SC executions where R/W to A or B are back to back?
    - Answer: Yes!

$A = 1$
$r_1 = B$
$B = 1$
$r_2 = A$

Thread 0

```
// A = r1 = 0

(S0) A = 1;
(L0) r1 = B;

print(r1);
```

Thread 1

```
// B = r2 = 0

(S1) B = 1;
(L1) r2 = A;

print(r2);
```

Data Race Free (DRF) Code: The Old-Fashioned Way

- Add inline fences, forcing the hardware to make the writes visible
  - Has same problems w. portability, readability, and behavior discussed before

Thread 0

```
// A = 0

(S0) A = 1;
__sync_synchronize();
(L0) r1 = B;
print(r1);
```

Thread 1

```
// B = 0

(S1) B = 1;
__sync_synchronize();
(L1) r2 = A;
print(r2);
```

DRF Code: The New Way

- Add a compiler directive declaring `A, B` as special variables (in Java, C++11)
  - The ”special type” depends on language (later this lecture!)
  - Compiler ensures atomic & program order **for you**
  - e.g., `atomic int`

Thread 0

```
// atomic int A = 0

(S0) A = 1;
(L0) r1 = B;

print(r1);
```

Thread 1

```
// atomic int B = 0

(S1) B = 1;
(L1) r2 = A;

print(r2);
```

DRF Implies SC: Before and After

- Before: needed to adjust our programs depending on consistency model in the ISA
- After: even on hardware that uses PC or WC, our program sees it as SC!
  - Programmer simply needs to obey the rules
- Approaches to implement synchronization in HW
  - We will discuss synchronization in the next lecture

Another DRF Example

- Does this code contain a data race?

Thread 0

```
// x = y = 0;
if( x ) {
  y = true;
}
```

Thread 1

```
// x = y = 0;
if( y ) {
  x = true;
}
```

Another DRF Example

- Does this code contain a data race?
  - Answer: No! (surprising, I know)
  - Executing the reads to x, y atomically means that the writes can never happen!

Thread 0

```
// x = y = 0;
if( x ) {
  y = true;
}
```

Thread 1

```
// x = y = 0;
if( y ) {
  x = true;
}
```

Implications of DRF Programming

- Sections of code containing no synch. operations appear to execute atomically to all other threads
  - Why? Otherwise, there would be data races!
  - Convince yourself this is true (draw examples)

- Calling an external library behaves as if it executes in one step, with no intermediate values
  - If stateless, no synchronization required in any form!
  - If the library has internal state, it also has to be DRF

Review: Java Memory Model

- Each threads T_i and T_j execute actions A_n:
  - Read or write to variables
  - Lock (acquire) or unlock (release) a monitor

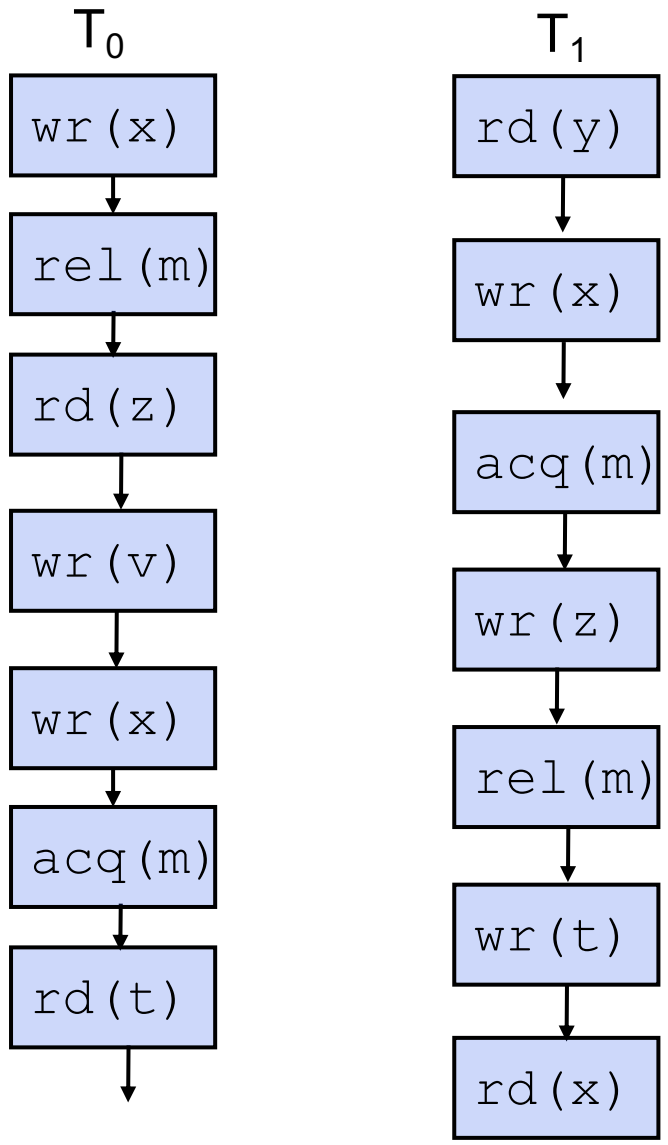
- Memory order in Java is defined by ”happens-before” (HB) relationships
  - These define which re-orderings are possible in both the **compiler** and the **JVM**
  - From now on, “Happens Before” = $\rightarrow$
- Given two threads’ actions, we can find their $\rightarrow$ relationships and therefore **check for data races**

JMM: “Happens Before”

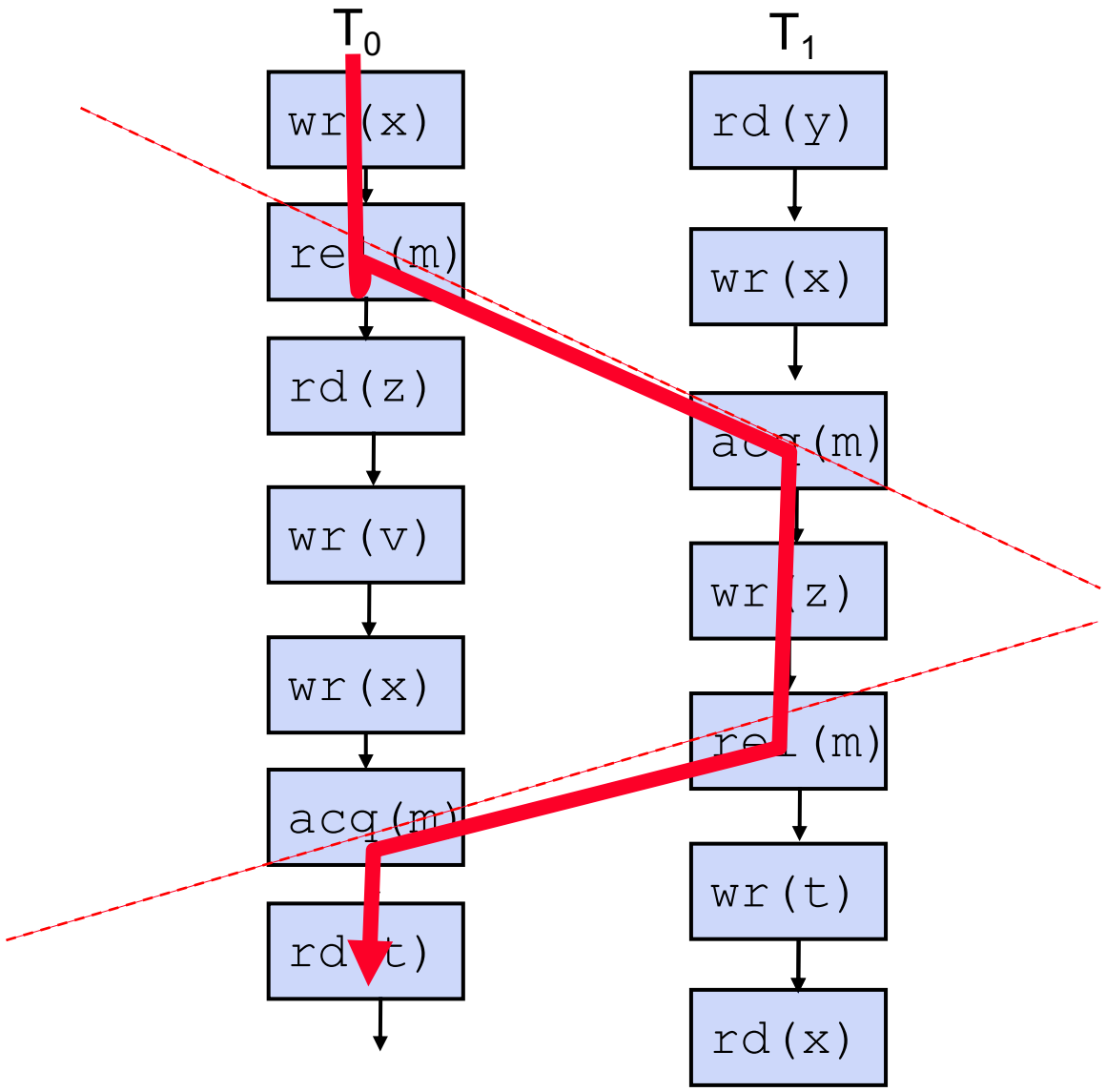
- → relationships you should care about:
  - Actions **in same thread** → each other **in program order**
  - Unlocking a `monitor` → all locking operations
  - Writing to a `volatile` → all reads of that field
  - All actions in a thread → a `join()` on that thread
  - Transitivity: If $A_x \rightarrow A_y$, and $A_y \rightarrow A_z$, then $A_x \rightarrow A_z$
- If we can find two actions A_x so that A_x **does not** → A_y , and A_y **does not** → A_x, we have found a **data race**

Exercise: Find the → Relationships

- Legend:
  - `wr()` and `rd()` are variable accesses
  - `acq()` is a monitor lock
  - `rel()` is a monitor unlock

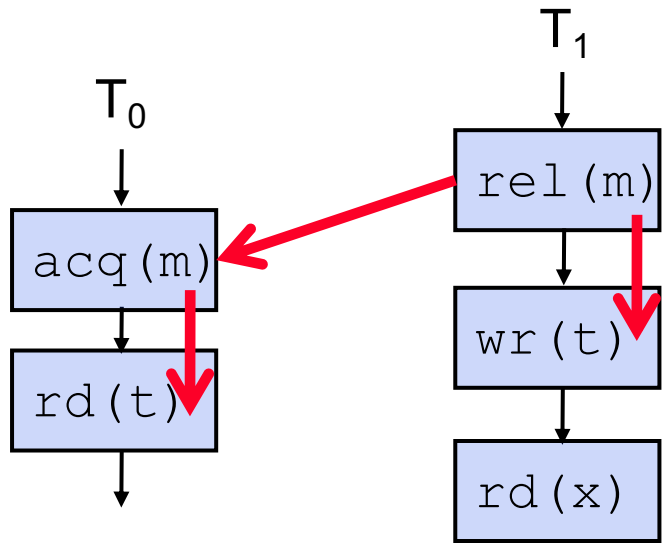


Exercise: Find the → Relationships



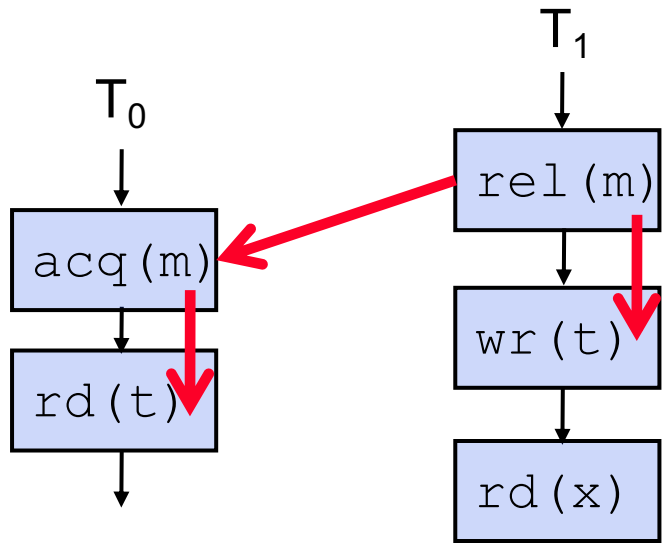
Writing DRF Code in Java

- Any action sees **all** of the actions that → it
  - But, it may or may not see other actions
- Go back to the previous example:
  - `rel(m)` → `acq(m)`
  - `acq(m)` → `rd(t)`
  - `rel(m)` → `wr(t)`
- Does `wr(t)` → `rd(t)`??



Writing DRF Code in Java

- Any action sees **all** of the actions that → it
  - But, it may or may not see other actions
- Go back to the previous example:
  - `rel(m)` → `acq(m)`
  - `acq(m)` → `rd(t)`
  - `rel(m)` → `wr(t)`



- Does `wr(t)` → `rd(t)`??
  - Answer: No!
    `rd(t)` will see all of T_1's actions that → `rel(m)`, but not guaranteed to see `wr(t)`!
  - And therefore, **this code has a data race**

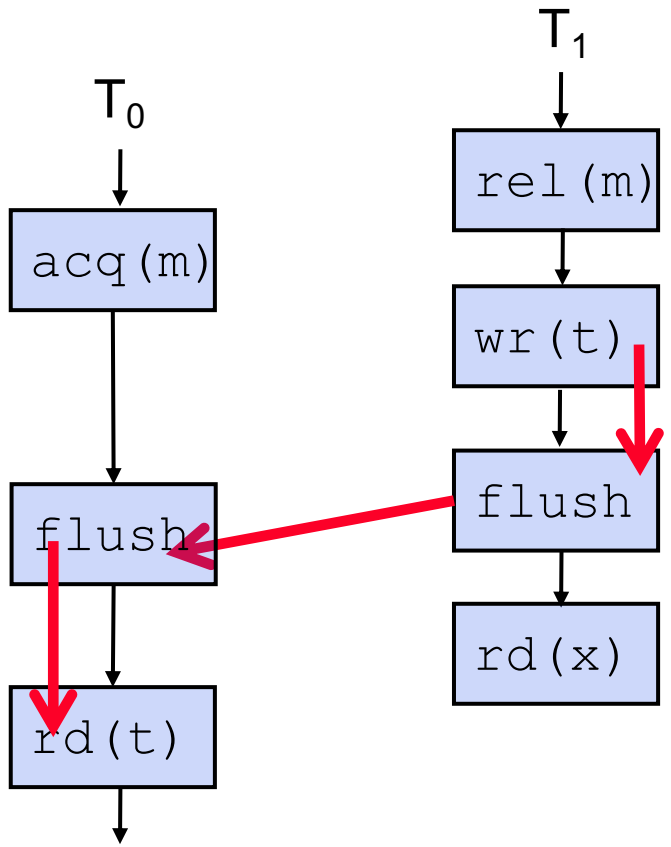
Fixing Data Races

- Conceptually, must add particular → relationships to eliminate all unordered accesses
- Can do that with monitors, `synchronized { }` blocks, or `@volatile` variables
  - All of them introduce → between conflicting accesses (R/W and W/W)

Eliminating Data Race on `t`

To introduce → between the write and read to `t`, could declare `t` as `volatile`

- Language spec says that `t` will never be cached "thread locally", always written to memory
- `volatile` in Java guarantees a → between the write and read to `t`
- Specifics left up to JVM writers (i.e., what H/W op it uses)
- Warning! This is only for Java, not C, C++



Synchronized { … } versus Volatile

Synchronized

- Only works w. objects
- Thread can wait while others in `synch{…}`
- Can wrap a function or critical section

```
synchronized {
  a = b+c;
  x = foo();
}
```

Volatile

- Can use w. primitives (e.g., `int`)
- No synchronization/direct access
- Prevents specific memory re-orderings

```
@volatile int a,b;
a = b+c;
x = foo();
// can't reorder before wr(a)
```

Lack of a C++ Memory Model

- C++ threads did not formally exist until 2011 (C++ was born in 1979)
  - Libraries such as `pthreads`, `OpenMP` were layered on top of the old language specification
- Behavior depends on version, platform, and compiler, since memory model was not part of C++
  - Code is often riddled with data races, not portable, etc…

Changes in C++11

- Threads are now officially part of the language
  - `std::thread( )`, and `std::async( )`
  - Therefore, need to formalize a memory model

- Provides the “DRF implies SC” guarantee
  - Uses similar ”happens before” reasoning to Java

- **If your prog. has a data race, its behavior is completely undefined**
  - Known as “catch fire semantics”

C++11 Memory Model

- All threads can read/write to **any** location in the program scope – through references & pointers
  - More permissive than Java
- Data races are similarly defined:
  - Two conflicting memory operations that do not operate on “special” variables, or ordered by →

C++11 Synchronization Variables

- Reminder: a special variable is one that obeys an agreed upon memory order
- C++11 uses an `std::atomic<T>` template to designate a special variable of type `T`
  - Conflicting accesses to a synch. variable are **not** considered a data race
  - e.g., `std::atomic<int>`, `std::atomic<TrivialClass>`
    - Can use your own class if it's considered Trivial (which is well defined, you can read about it online)
- C++11 also defines `std::mutex( )`, which is the loose equivalent of a Java monitor

Use of Atomic Variables

- Quick example with an atomic integer:
  - Guaranteed that `count = 4` after all threads complete and are joined
  - Join omitted from the example

```
std::atomic<int> count = 0;

void main( ) {
  for(int i = 0; i<4;i++)
    std::thread( [&]{count++;} );
}
```

DRF Example on PC Hardware

- Assume `std::atomic<int>` variables translated to instructions: `S_Load, S_Store`
  - `S_Load` and `S_Store` can be implemented in a variety of ways (e.g., with fence instructions in ARM, atomic instructions in x86)

Thread 0

```
// A,B = 0

(S0) A = 1;
(L0) r1 = B;

print(r1);
```

pseudo-x86

```
// possible compiled code

S_Store 1,(A)
S_Load (B),%eax

// setup args for print
...
jmp _print
```

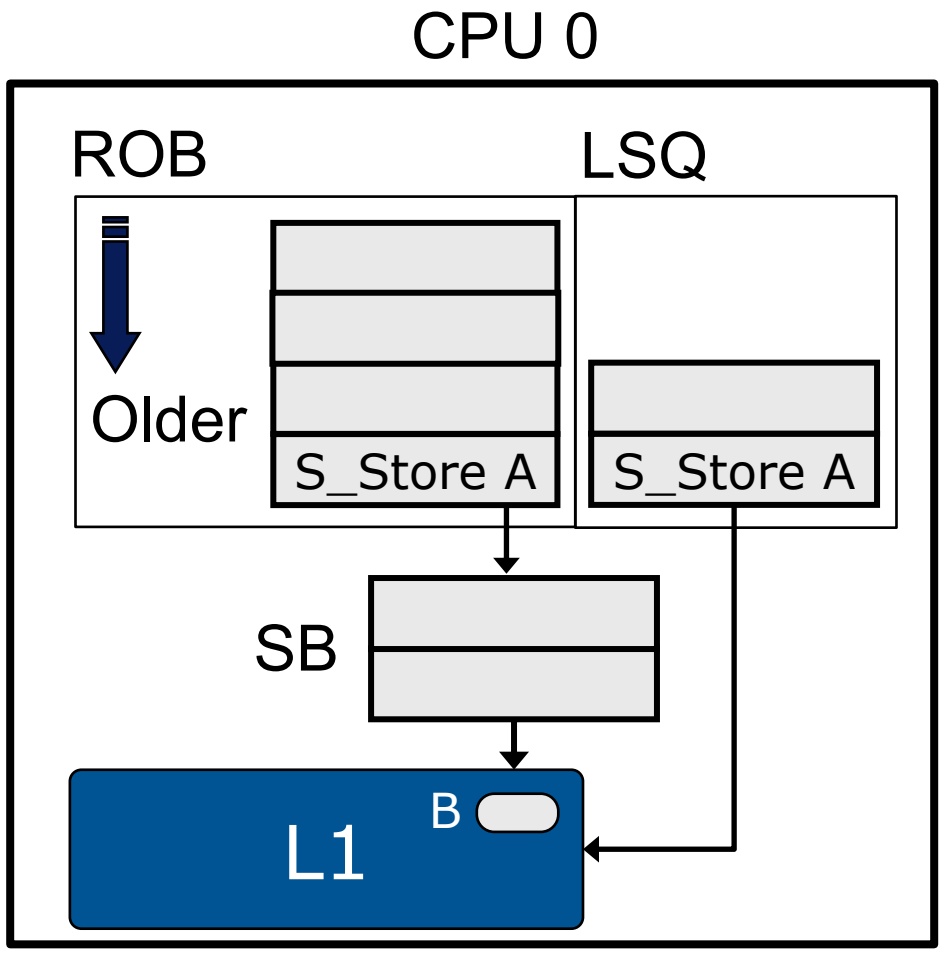
DRF Example on PC Hardware

- Now that our program is DRF,
  hardware must execute it as if it were SC

- What behavior **must** we disable if our processor is PC?
  (Recall Lecture 5.1)
  - Synch reads cannot bypass blocked synch writes
  - Need to hold operations in ROB, drain the store buffer

DRF Example on PC Hardware

1. `S_Store 1,(A)`
   - Resolves address (A)

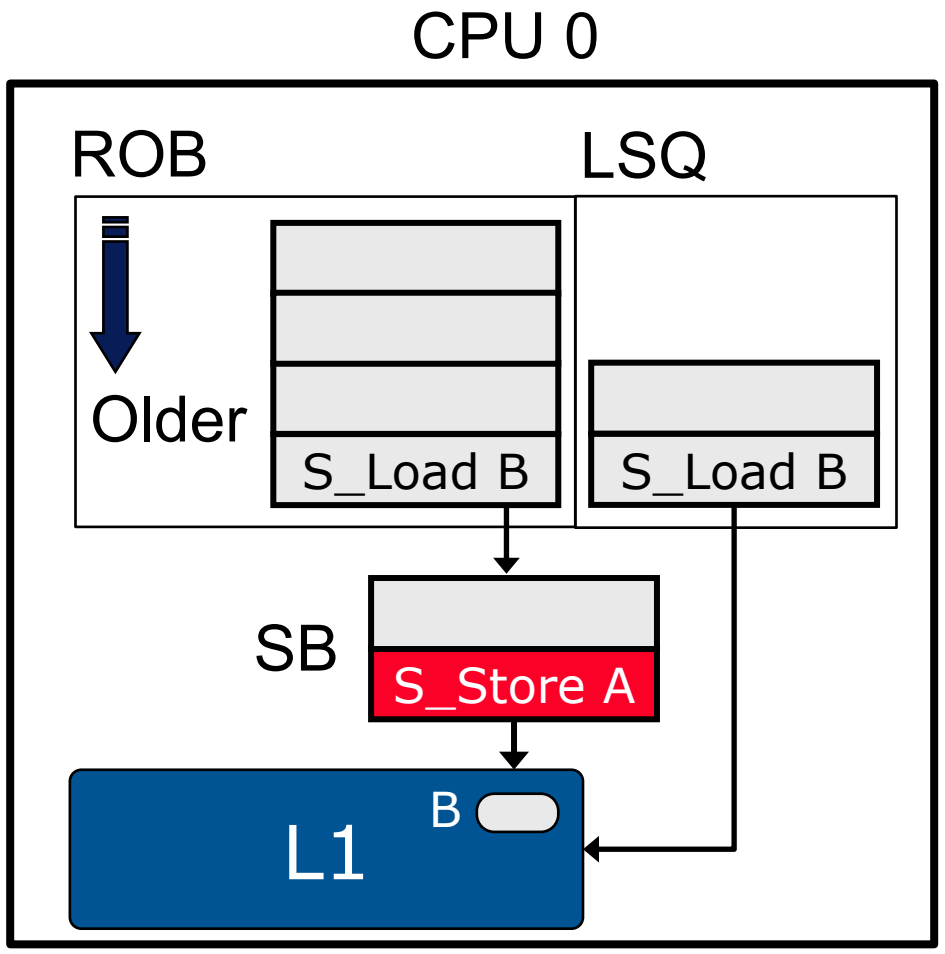


Program

```
S_Store 1,(A)
S_Load (B),%eax
...
jmp _print
```

DRF Example on PC Hardware

1. `S_Store 1,(A)`
   - Resolves address (A)
   - Retires to Store Buffer but incurs L1 cache miss (red)
2. `S_Load (B),%eax`
   - Resolves address (B)

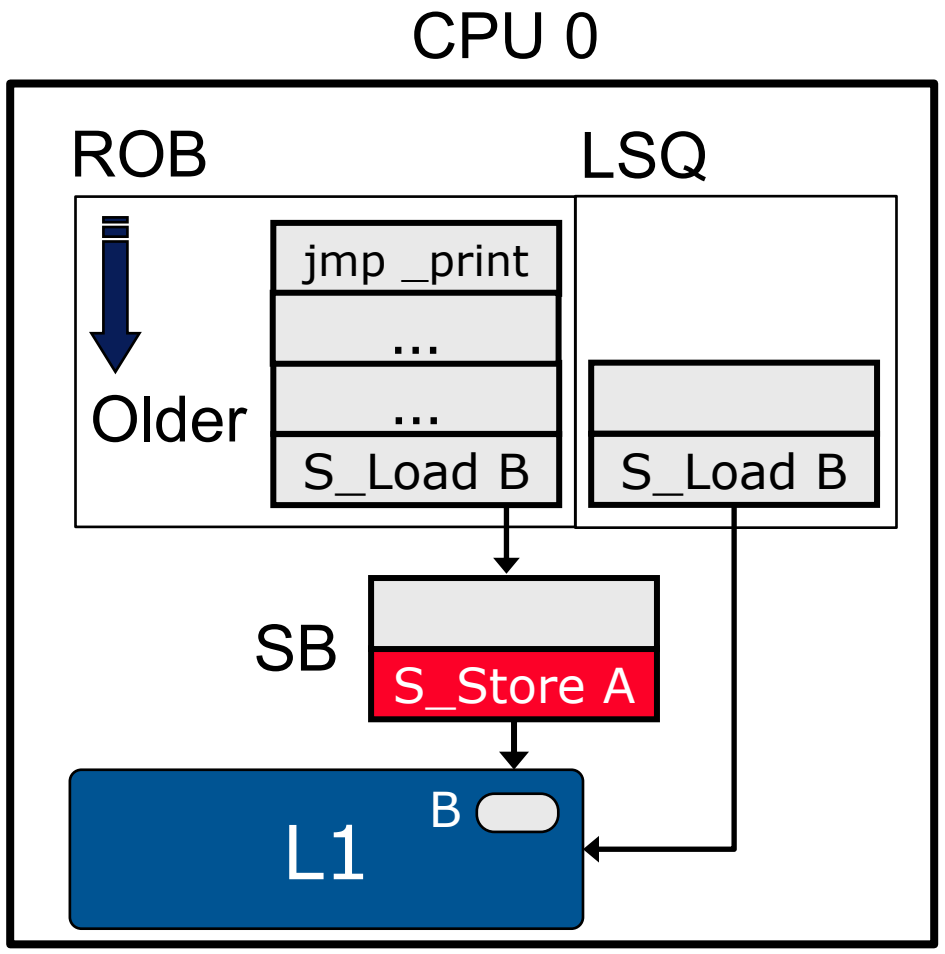


Program

```
S_Store 1,(A)
S_Load (B),%eax
...
jmp _print
```

DRF Example on PC Hardware

1. `S_Store 1,(A)`
   - Waiting for cache
2. `S_Load (B),%eax`
   - Resolves address (B)
   - A regular load (not S_Load) in PC, could issue and incur non-SC reordering (but in x86 all loads are the same, later)

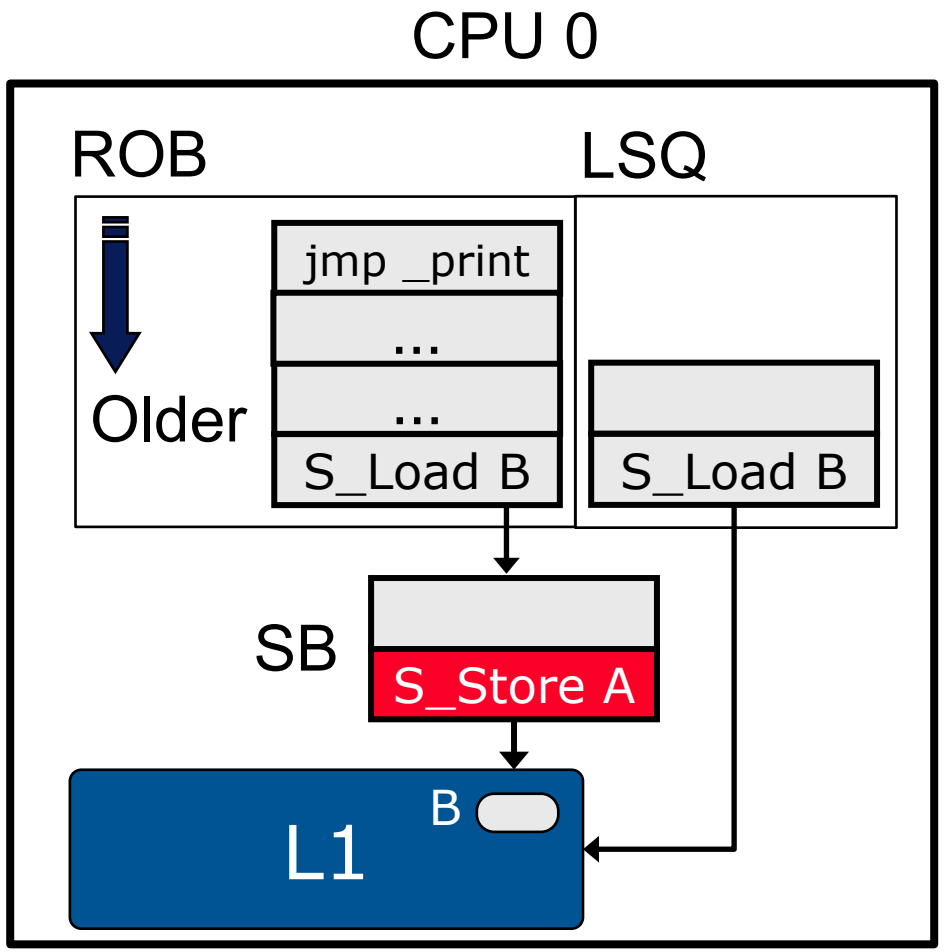


Program

```
S_Store 1,(A)
S_Load (B),%eax
...
jmp _print
```

DRF Example on PC Hardware

1. `S_Store 1,(A)`
   - Waiting for cache
2. `S_Load (B),%eax`
   - Resolves address (B)
   - But S_Load **must** wait for SB to drain (can peek to overlap latency)

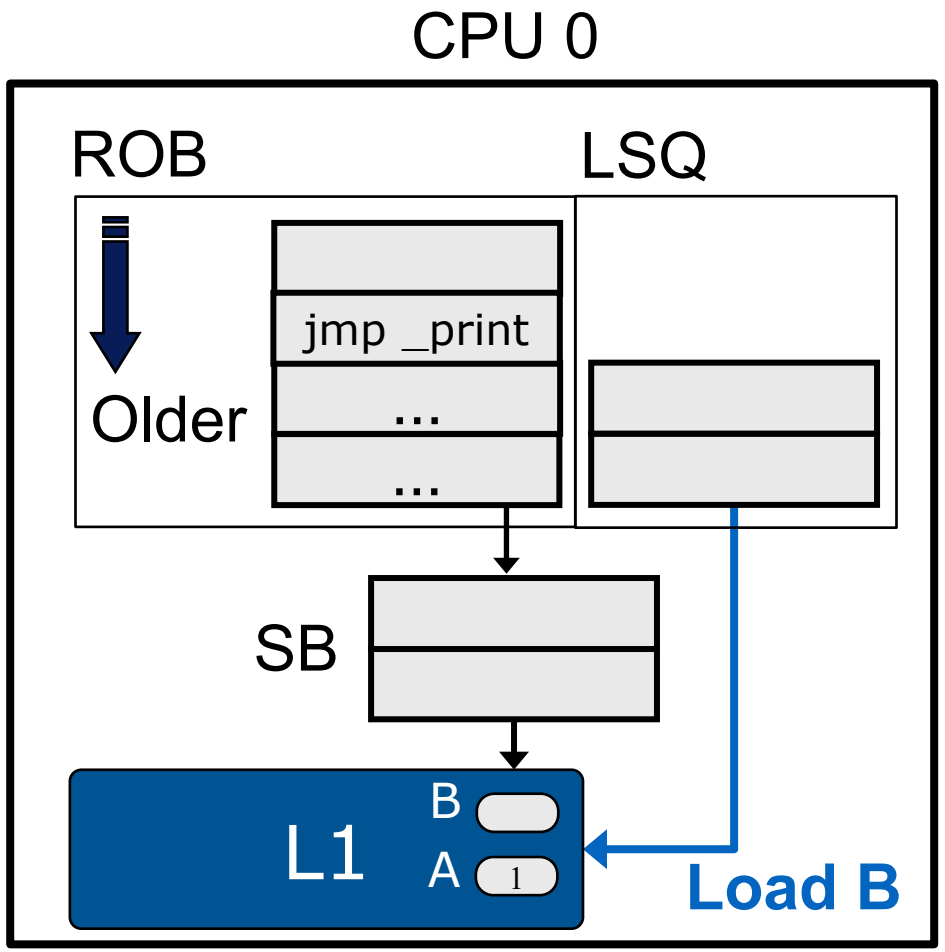


Program

```
S_Store 1,(A)
S_Load (B),%eax
...
jmp _print
```

DRF Example on PC Hardware

1. `S_Store 1,(A)`
   - Now in cache
2. `S_Load (B),%eax`
   - Resolves address (B)
   - **Issue Load B** → Hit
- When `S_Store 1,(A)` finishes, load executes



Program

```
S_Store 1,(A)
S_Load (B),%eax
...
jmp _print
```

Take a Break!

Recall What Happens with Two Threads

- Example with the code with variables swapped in two threads
- A, B are declared atomic

Thread 0

```
// atomic<int> A,B = 0

(S0) A = 1;
(L0) r1 = B;

print(r1);
```

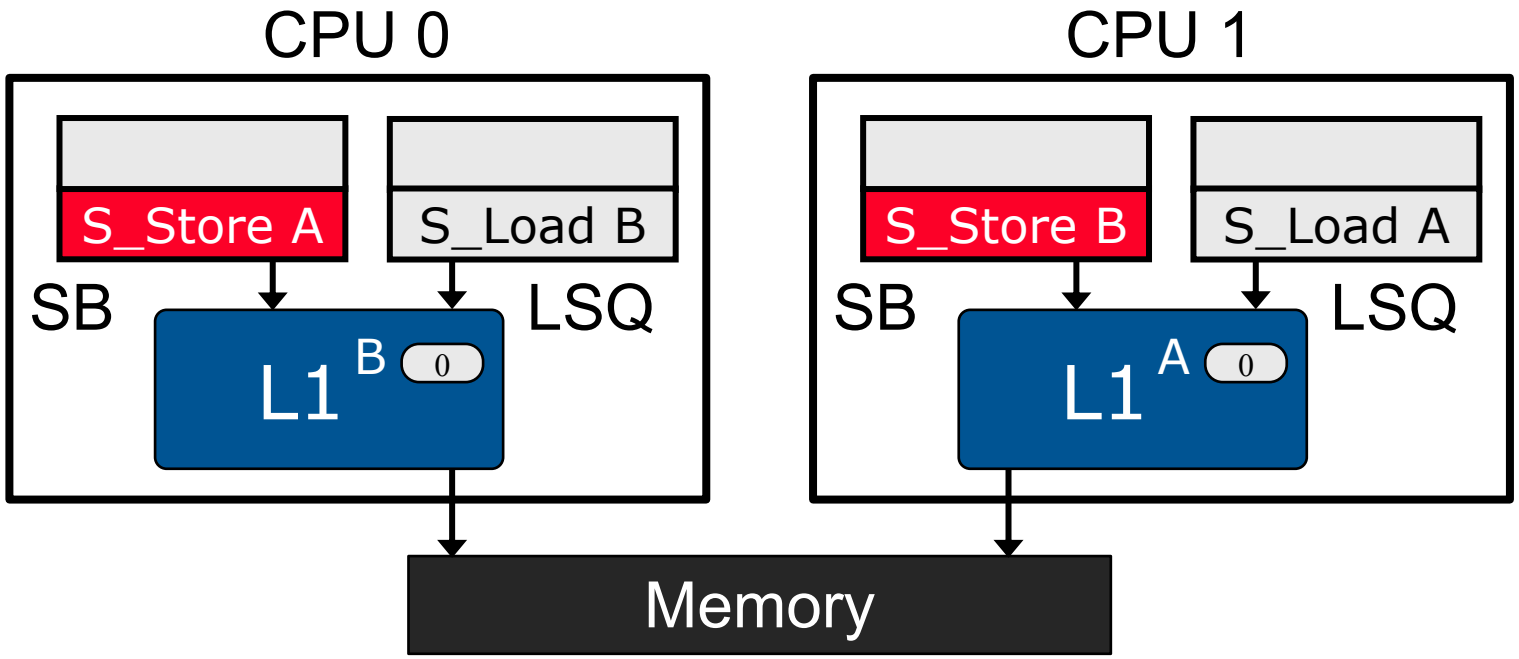
Thread 1

```
// atomic<int> A,B = 0

(S1) B = 1;
(L1) r2 = A;

print(r2);
```

DRF Example on PC Hardware

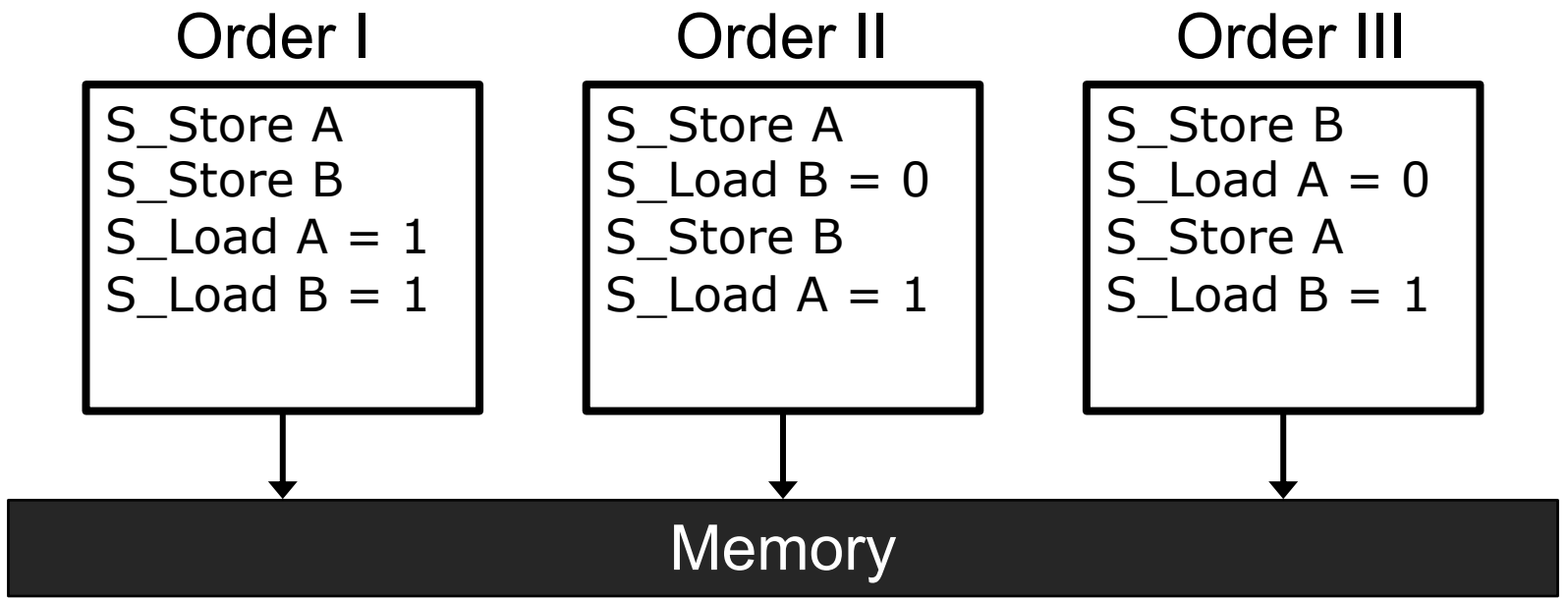
- If we show both processors' SBs/LSQs,

  we see how the hardware looks like as if it is SC
  - Issuing loads can lead to a non-SC (0,0) outcome
  - **But**, holding both means the memory behaves like our "switch" model, and therefore we get only SC outcomes!



DRF Example on PC Hardware

- Potential orders when loads wait for **local** stores:
  - I - Both stores first, outcome (1,1)
  - II - CPU 0 first, outcome (0,1)
  - III - CPU 1 first, outcome (1,0)



Review: C++11 Memory Model

- C++11 provides the "DRF implies SC" guarantee
  - Uses similar "happens before" reasoning to Java

- C++11 uses `std::atomic<T>` to declare a special variable of type `T`
  - Conflicting accesses to a special variable are **not** considered a data race

DRF Example on PC Hardware

- Example: Thread 0 passes `data` to thread 1 with `flag` for readiness

- `data` and `flag` are `std::atomic<int>`'s initialized to 0
  - `std::atomic` ensures SC ordering by default
  - SC execution: thread 1 must return 1

Thread 0

```
// data = flag = 0

data = 1;
flag = 1;
```

Thread 1

```
while (!flag) {}

return data;
```

DRF Example on PC Hardware

- Assembly code on x86
  - `S_Load` becomes `mov [addr], reg` (move memory data to reg)
  - `S_Store` becomes `xchg reg, [addr]` (exchange reg and memory data)

Thread 0

```
mov  1, %eax
xchg %eax, [data]
xchg %eax, [flag]
```

Thread 1

```
loop:
    mov [flag], %eax
    cmp %eax, 0
    je  loop

mov [data], %eax
```

volatile int vs. std::atomic<int>

- data and flag can also be volatile int's
  - Prevents the compiler from removing or reordering memory accesses
  - In this case both S_Load and S_Store are normal mov's
  - This code runs correctly on PC hardware

Thread 0

```
mov 1, %eax
mov %eax, [data]
mov %eax, [flag]
```

Thread 1

```
loop:
    mov [flag], %eax
    cmp %eax, 0
    je  loop

mov [data], %eax
```

DRF Example on WC Hardware

- `std::atomic<int>` also works on WC hardware
  - `std::atomic` ensures SC ordering by default
  - SC execution: thread 1 must return 1, same as on PC hardware

Thread 0

```
// data = flag = 0

data = 1;
flag = 1;
```

Thread 1

```
while (!flag) {}

return data;
```

DRF Example on WC Hardware

- Assembly code on ARMv8
  - `dmb` is ARMv8's memory fence
  - `S_Load` becomes `dmb; ldr; dmb`
  - `S_Store` becomes `dmb; str; dmb`

Thread 0

```
mov x10, #1
dmb
str x10, [data]
dmb
dmb
str x10, [flag]
dmb
```

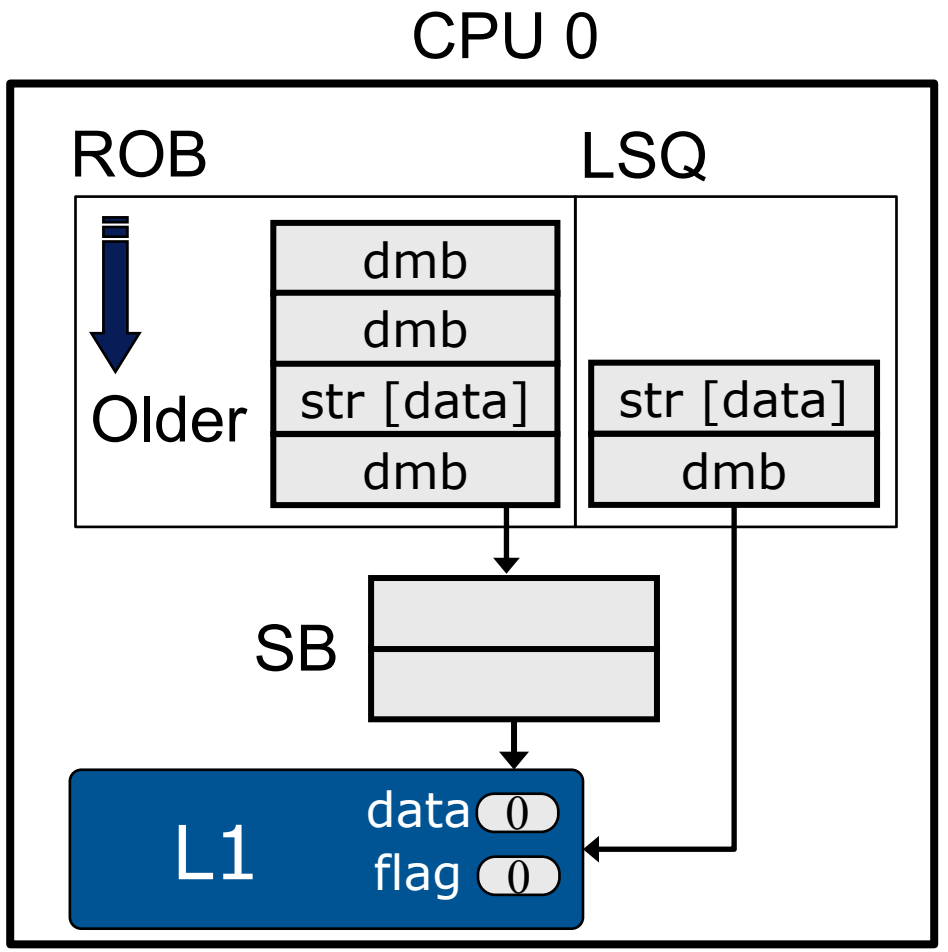
Thread 1

```
loop:
    dmb
    ldr x10, [flag]
    dmb
    cbz x10, loop
    dmb
    ldr x10, [data]
    dmb
```

Execution on ARMv8 (Thread 0)

- Commit `dmb`
  - Prevents reordering of younger loads/stores (e.g., `str [data]`)

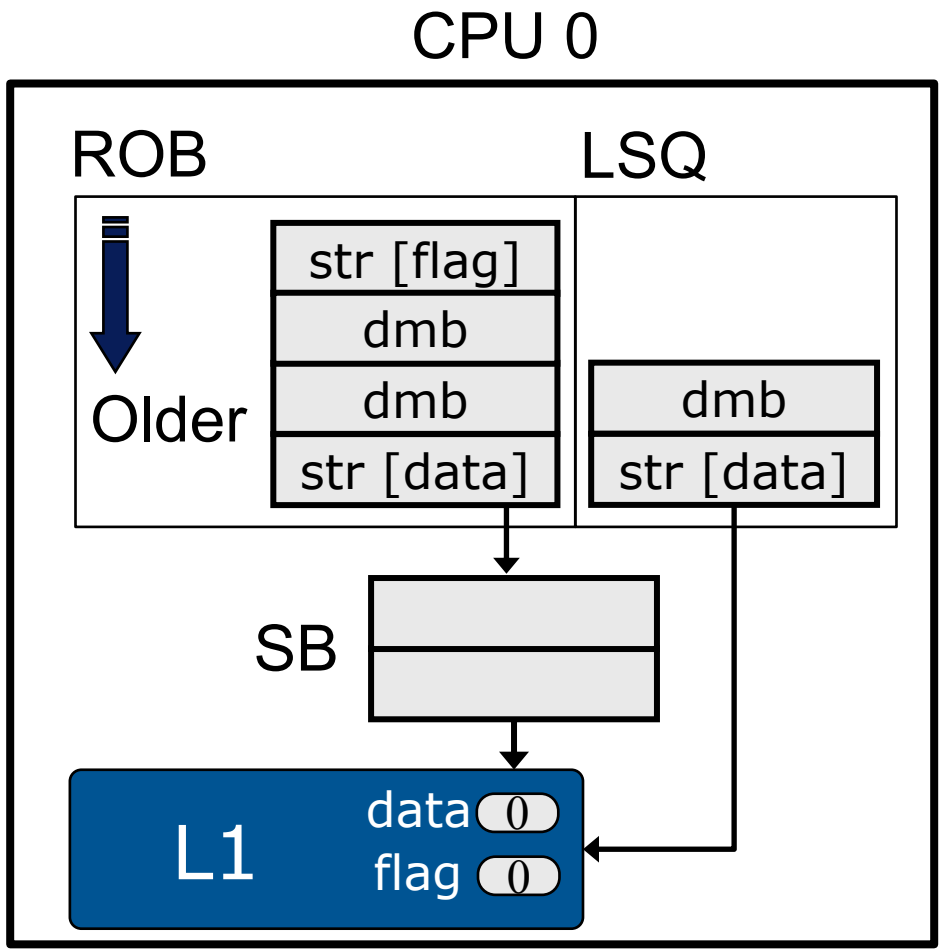
```
mov x10, #1
dmb
str x10, [data]
dmb
dmb
str x10, [flag]
dmb
```



Execution on ARMv8 (Thread 0)

- Commit `str [data]`

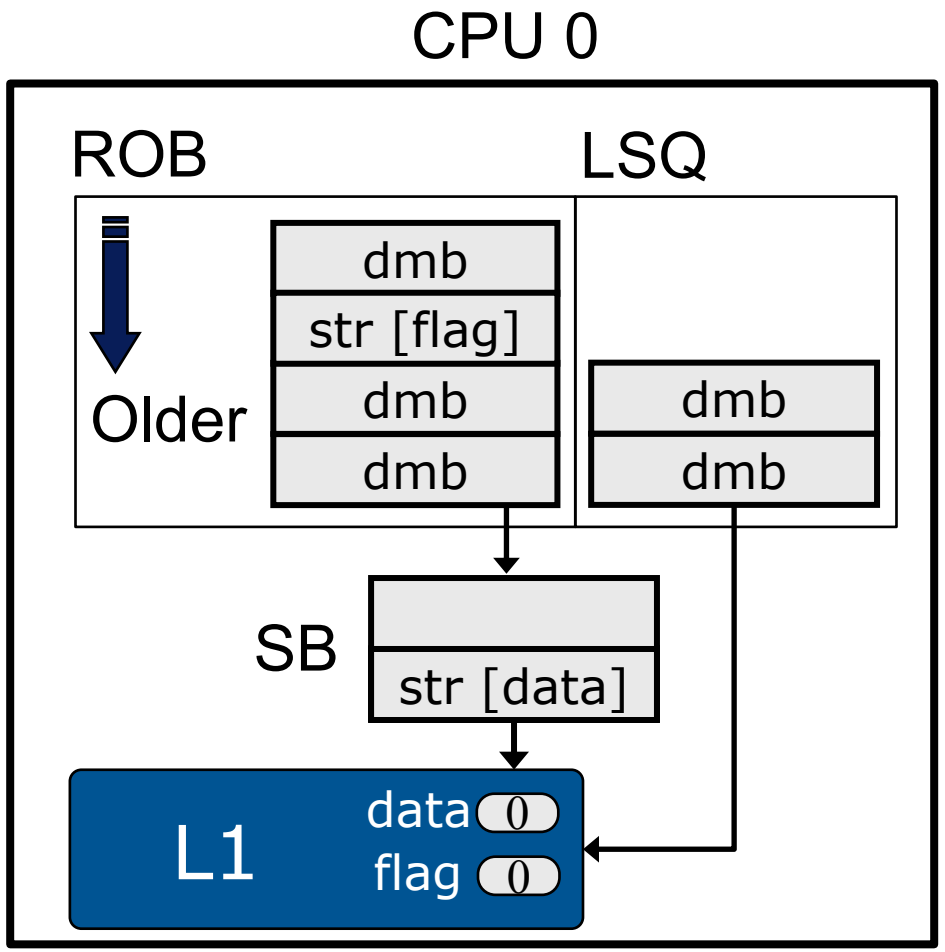
```
mov x10, #1
dmb
str x10, [data]
dmb
dmb
str x10, [flag]
dmb
```



Execution on ARMv8 (Thread 0)

- Commit `dmb`
  - `str [data]` enters into SB
  - `dmb` waits for SB to be drained

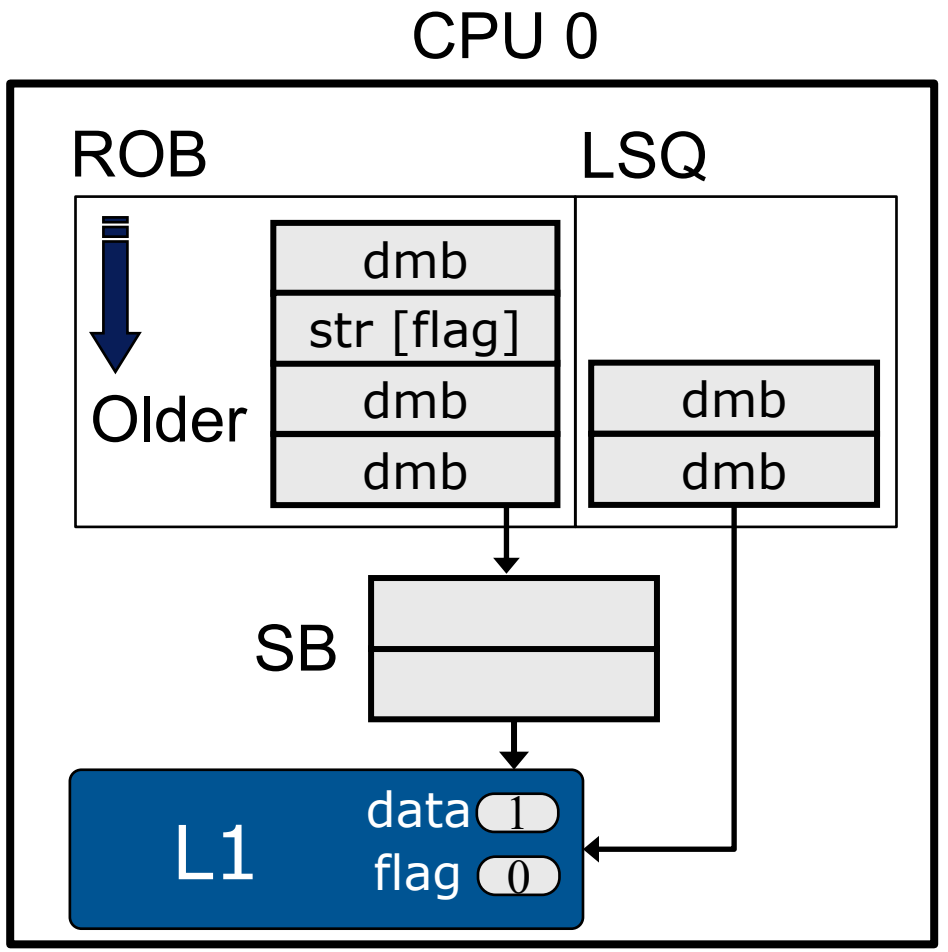
```
mov x10, #1
dmb
str x10, [data]
dmb
dmb
str x10, [flag]
dmb
```



Execution on ARMv8 (Thread 0)

- Commit `dmb`
  - SB drained
  - Prevents reordering of younger loads/stores (e.g., `str [flag]`)

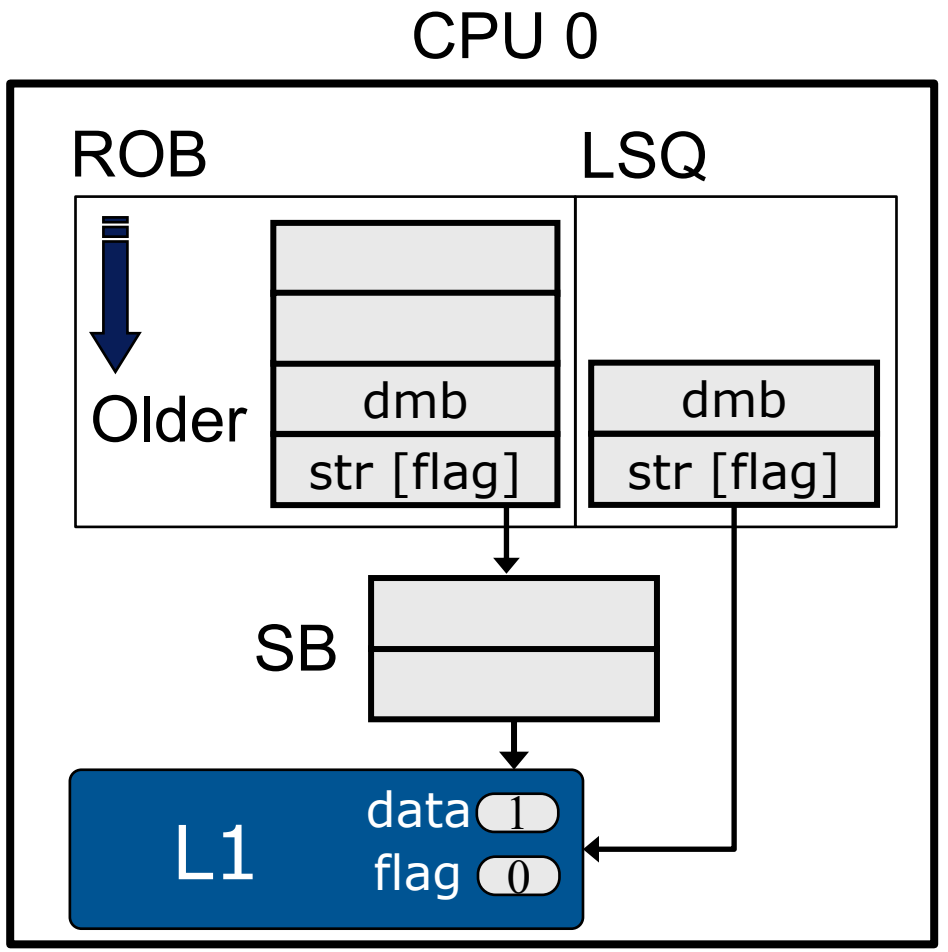
```
mov x10, #1
dmb
str x10, [data]
dmb
dmb
str x10, [flag]
dmb
```



Execution on ARMv8 (Thread 0)

- Commit `str [flag]`

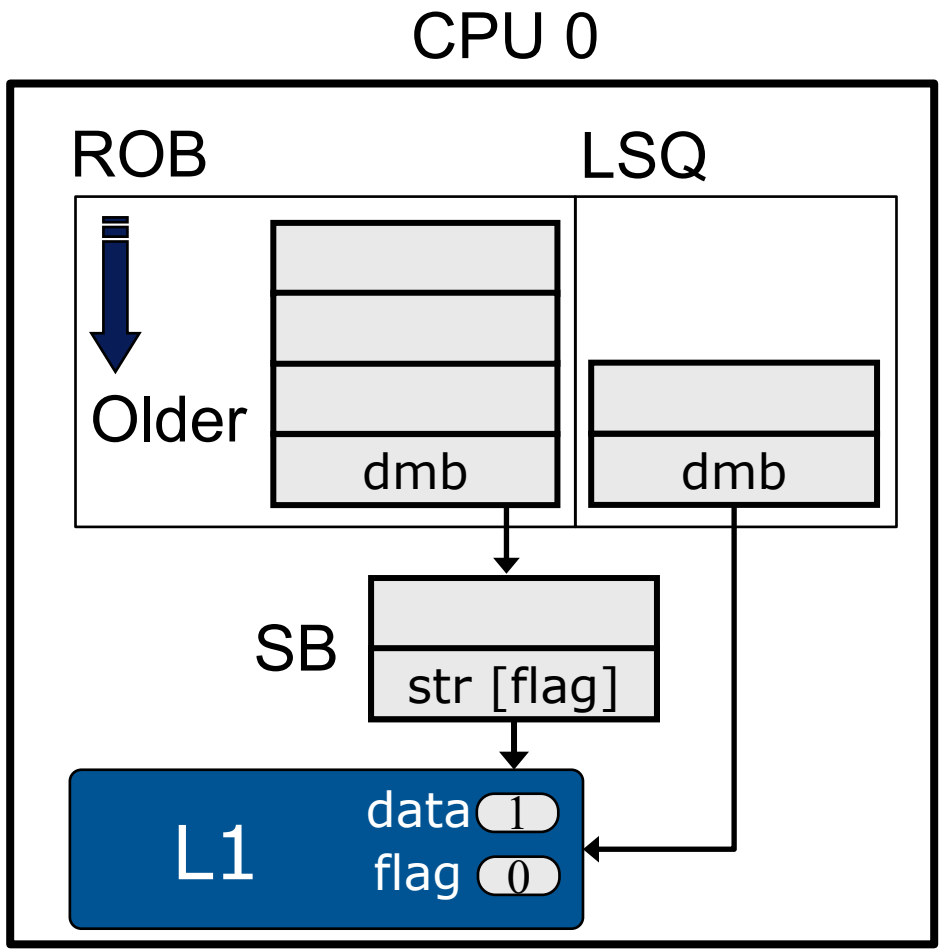
```
mov x10, #1
dmb
str x10, [data]
dmb
dmb
str x10, [flag]
dmb
```



Execution on ARMv8 (Thread 0)

- Commit `dmb`
  - `str [flag]` enters into SB
  - `dmb` waits for SB to be drained

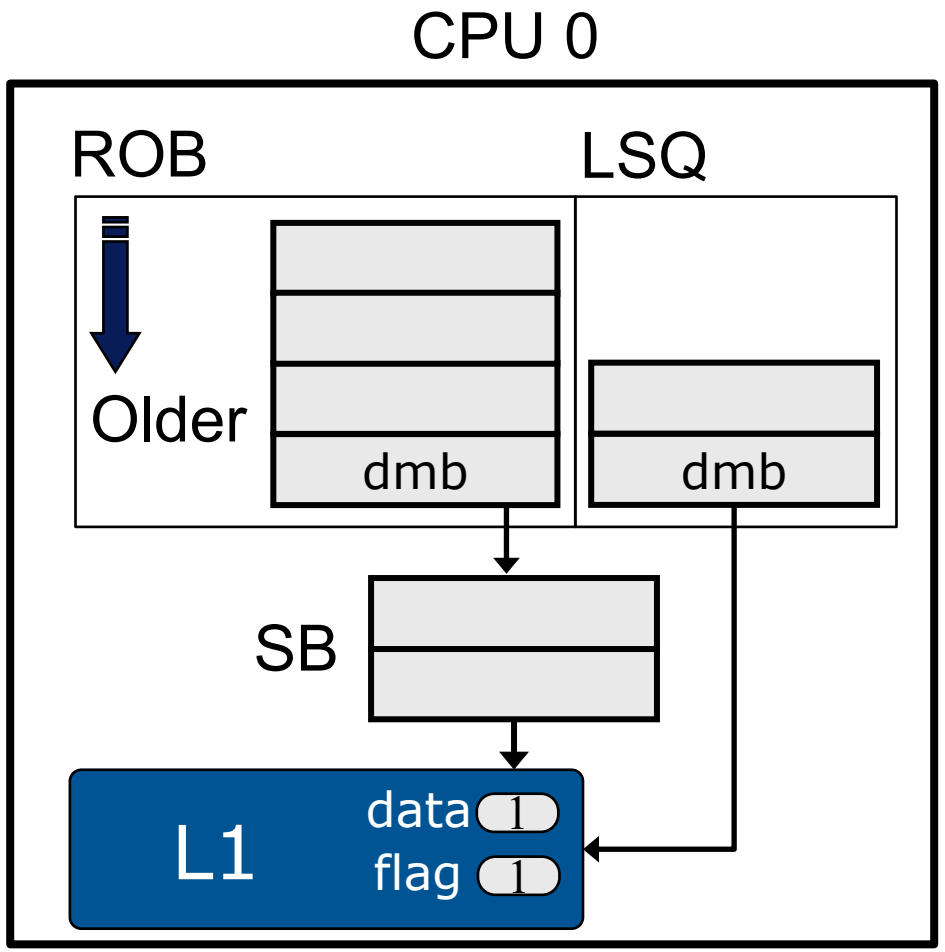
```
mov x10, #1
dmb
str x10, [data]
dmb
dmb
str x10, [flag]
dmb
```



Execution on ARMv8 (Thread 0)

- Commit `dmb`
  - SB drained

```
mov x10, #1
dmb
str x10, [data]
dmb
dmb
str x10, [flag]
dmb
```



`volatile int` vs. `std::atomic<int>`

- Using volatile `int`'s does not work anymore!

Thread 0

```
// data = flag = 0

data = 1;
flag = 1;
```

Thread 1

```
while (!flag) {}

return data;
```

`volatile int` vs. `std::atomic<int>`

- Assembly code on ARMv8
  - Two `str`'s can be reordered because of WC hardware
  - This code does not run correctly!

Thread 0

```
mov x10, #1
str x10, [data]
str x10, [flag]
```

Thread 1

```
loop:
    ldr x10, [flag]
    cbz x10, loop
    ldr x10, [data]
```

Advanced Usage of C++ Atomics

- Explicit load/store, assuming `std::atomic<int> x`
  - `x.load(order)`
  - `x.store(value, order)`
- For a specific load/store `order` can be directly specified
- The default order is `std::memory_order_seq_cst`

Advanced Usage of C++ Atomics

For example, `std::atomic<int> x`

`x = 1` is the same as

```
x.store(1, std::memory_order_seq_cst)
```

`… = x` is the same as

```
… = x.load(std::memory_order_seq_cst)
```

Advanced Usage of C++ Atomics

- Other memory orders options
  - `std::memory_order_relaxed`
  - `std::memory_order_acquire` (for more advanced hardware)
  - `std::memory_order_release` (for more advanced hardware)
- Specifying other memory orders does not make a difference on PC
- More on this in the MS course

Advanced Usage of C++ Atomics

- For example, to minimize memory fences on ARMv8
  - Leveraging various memory order options
  - `flag` is still `std::atomic<int>` (special), but `data` can be `int` (normal)

Thread 0

```
// data = flag = 0

data = 1;
flag.store(1,
std::memory_order_relaxed);
```

Thread 1

```
while
(!flag.load(std::memory_ord
er_relaxed)) {
}

return data;
```

Advanced Usage of C++ Atomics

- Assembly code on ARMv8
  - `stlr`: store-release: no load/store before it can be reordered after it
  - `ldar`: load-acquire: no load/store after it can be reordered before it
  - ARMv8 provides direct support for C++ semantics

Thread 0

```
mov  x10, #1
str  x10, [data]
stlr x10, [flag]
```

Thread 1

```
loop:
    ldar x10, [flag]
    cbz  x10, loop
    ldr  x10, [data]
    ret
```

How Compiler Enforces the Ordering Constraints

- At the language level
  - Compiler issues load/store instructions in the same order as in the program
- At the ISA level
  - Compiler issues ISA-specific instructions with the required ordering constraints
  - E.g., in ARMv8, the compiler inserts `dmb`'s (memory fences)
  - In x86 (PC), the compiler just emits normal `mov`'s or atomic instructions for locks (covered on Thursday)

OpenMP Memory Ordering

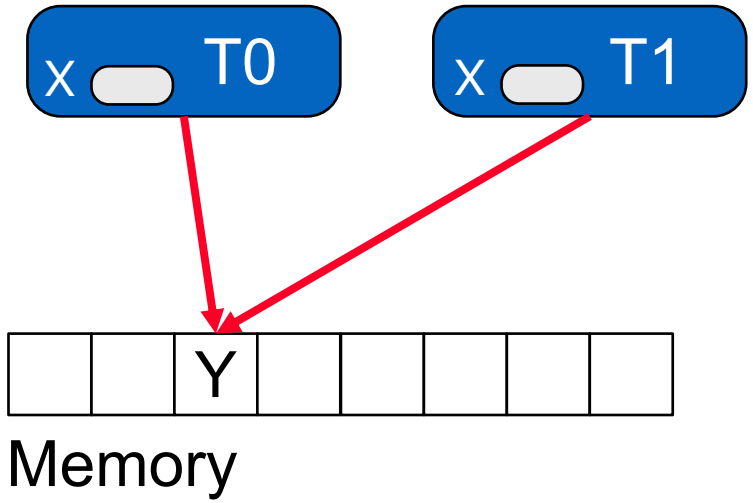
- Since OMP is yet another threading library, its memory model differs from C++11
- Key differences:
  - Variables can have different “scope”
  - There is no concept of a “synchronization variable” like Java’s `@volatile` and C++11’s `std::atomic<T>`
  - Locks must be accompanied by a `Flush` (ex. to come)

- Officially, OMP implements **weak consistency** except for `flush` operations
  - Almost any reordering is therefore possible

OMP Variable Scope

- Recall from Lecture 2.1, each thread has a set of shared and private variables
  - Shared – refers to the same **address** as the original
  - Private – runtime provides a **new** var, same size & type
- When writing code, need to consider data races only on shared variables

```
#pragma omp parallel
private(x) shared(y)
{
    x = ...
    *y++; // data race
}
```



OMP Flushes

- Definition of a flush:
  - "Memory operations for variables in the flush set that precede the flush in program execution order must be firmly lodged in memory and available to all threads before the flush completes, and memory operations for variables in the flush set, that follow a flush in program order cannot start until the flush completes."
  - In plain English, a flush is a memory fence

- In order to communicate between threads, do all of the following:
  - First thread writes the **shared** variable, flushes the variable
  - Second thread flushes, and **then** reads

Example of Communication

- One thread updating a shared variable `y`
  - Assume `thr_id` is already set
- Note how we flush after write, and before read

```
int y;
#pragma omp parallel shared(y)
{
  if( thr_id == 0 ) {
    y = ...;
    #pragma omp flush(y)
  }
  #pragma omp flush(y)
  ... = y;
}
```

Summary

- The hardware's memory model percolates all of the way up the stack, to how we all program
  - Corollary: Write data race free programs!
- Even in relaxed hardware, need special support to behave SC in some cases!

Rust: A Memory-Safe Language from Mozilla

- Rust was created to solve several challenges in systems programming
  - Memory safety without garbage collection
  - Concurrency without data races
  - Zero-cost abstractions

- Rust's approach: ensure programs do not have undefined behaviors
  - Check statically as much as possible
  - Programmers cannot do whatever they want, including sharing data among threads without synchronization
  - Otherwise, the compiler complains!

Multithreading in Rust vs. C++

- Rust does not allow threads write to the same variable
  - Programmers must rely on explicit synchronization primitives

```
// declare a vector
std::vector<int> data = {1, 2, 3};

// create and run a thread



std::thread handle([&data]() {
    data.push_back(4);
});



data.push_back(5);
```

```
// declare a vector
let mut data = vec![1, 2, 3];

// create and run a thread
// error[E0373]: closure may outlive the
current function, but it borrows `data`,
which is owned by the current function
let handle = thread::spawn(|| {
    data.push(4);
});

// error[E0499]: cannot borrow `data` as
mutable more than once at a time
data.push(5);
```

Safe Way of Sharing Data in Rust

- Correctly using safe primitives, or compilation errors!

```
// Arc: Atomic Reference Count
let data = Arc::new(Mutex::new(vec![1, 2, 3]));
// clone the Arc to create another reference to the same data
let data_clone = Arc::clone(&data);

let handle = thread::spawn(move || {
    // grab lock and get the reference
    let mut v = data_clone.lock().unwrap();
    v.push(4);
});
{
    // grab lock and get the reference
    let mut v = data.lock().unwrap();
    v.push(5);
}
```

Go: A Modern Programming Language from Google

- Go was created to address challenges in software development
  - Scalability: efficiently handle large codebase and systems
  - Concurrency: simplify writing programs that utilize multicore processors
  - Simplicity: reduce complexity compared to C++ and Java

- Go's approach
  - Do not communicate by sharing memory; use message passing
  - So no need to care about memory consistency anymore!

Components in A Go Program

- Goroutine: light-weight thread
  - More on this in Lecture 8 and 9!
- Channel: typed pipes between goroutines

```
func send(c chan int) {
    c <- 1
}
func recv(c chan int) {
    <- c
}

// create a channel
c = make(chan int);

// start goroutines
go send(c);
go recv(c);
```

Components in A Go Program

- Goroutine: light-weight thread
  - More on this in Lecture 8 and 9!
- Channel: typed pipes between goroutines

```
func send(c chan int) {
    c <- 1
}
func recv(c chan int) {
    <- c
}

// create a channel
c = make(chan int);

// start goroutines
go send(c);
go recv(c);
```

Components in A Go Program

- Goroutine: light-weight thread
  - More on this in Lecture 8 and 9!
- Channel: typed pipes between goroutines

```
func send(c chan int) {
    c <- 1
}
func recv(c chan int) {
    <- c
}

// create a channel
c = make(chan int);

// start goroutines
go send(c);
go recv(c);
```