CSE 451: Operating Systems

Spring 2022

Module 8

Memory Consistency

John Zahorjan

Preliminaries

- Q: What is a program?
  - A1: Instructions for what to do
  - A2: Instructions for what to compute

- Example:
  - A = 10;
    B = A + 8;

    Do the compiled assembly/machine instructions include an add instruction?

Preliminaries (cont.)

- What are machine instructions?
  - A1: What to do?
  - A2: What effect to get?

- Example:

```
lw      x3, 0(x2)
addi     x3, x3, 1
sw      x3, 0(x2)
lw      x3, 4(x2)
addi    x3, x3, 1
sw      x3, 4(x2)
```

Preliminaries (cont.)

-

Instruction Given	Dependence
lw x3, 0(x2)	
addi x3, x3, 1	read-after-write (RAW)
sw x3, 0(x2)	read-after-write (RAW)
lw x3, 4(x2)	write-after-read (WAR)
addi x3, x3, 1	read-after-write (RAW)
sw x3, 4(x2)	read-after-write (RAW)

-

Instruction Executed	Dependence
lw x3, 0(x2)	
addi x3, x3, 1	read-after-write (RAW)
sw x3, 0(x2)	read-after-write (RAW)
lw x40, 4(x2)	
addi x40, x40, 1	read-after-write (RAW)
sw x40, 4(x2)	read-after-write (RAW)

Preliminaries (final)

- Q: What can the hardware know that the compiler can’t?
- A: Dynamic behavior of program

- What is in the cache right now?
- Was the conditional branch taken or not?
  - Is this conditional branch usually taken or not taken?
- Has a load from memory completed yet?
- Is there an idle ALU right now?
- What the CPU hardware is (as contrasted with the CPU architecture)

Background: Cache Coherence

5.2.1 Cache Coherence and Sequential Consistency

Several definitions for cache coherence (also referred to as cache consistency) exist in the literature. The strongest definitions treat the term virtually as a synonym for sequential consistency. Other definitions impose extremely relaxed ordering guarantees. Specifically, one set of conditions commonly associated with a cache coherence protocol are: (1) a write is eventually made visible to all processors, and (2) writes to the same location appear to be seen in the same order by all processors (also referred to as serialization of writes to the same location) [13]. The above conditions are clearly not sufficient for satisfying sequential consistency since the latter requires writes to all locations (not just the same location) to be seen in the same order by all processors, and also explicitly requires that operations of a single processor appear to execute in program order.

We do not use the term cache coherence to define any consistency model. Instead, we view a cache coherence protocol simply as the mechanism that propagates a newly written value to the cached copies of the modified location. The propagation of the value is typically achieved by either invalidating (or eliminating) the copy or updating the copy to the newly written value. With this view of a cache coherence protocol, a memory consistency model can be interpreted as the policy that places an early and late bound on when a new value can be propagated to any given processor

From https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-7.pdf

This Module's Topic: Memory Consistency

- "Memory consistency" is not the same as "cache coherence"
  - Cache coherence is about the visibility of operations to the same address

- Memory consistency is the set of rules that govern the visibility of memory operations issued by multiple cores
  - Memory consistency is about the visibility of operations to all addresses

- What is the meaning of this code (as a single threaded program)?
  - [initially A==B==0]

    ```
    A = 1
    print B
    B = 1
    print A
    ```

Program Order

- What is the meaning of this code (as a single threaded program)?
  - [initially A==B==0]
    ```
    A = 1
    print B
    B = 1
    print A
    ```
- Prints 01
- The order in which the operations are expected to be executed is called program order

Multiple Threads / Memory Consistency

- Suppose I have the same code, but implemented in two threads
- What are the possible outcomes of running this code?

T1

```
A = 1
print B
```

T2

```
B = 1
print A
```

Sequential Consistency

- Sequential consistency is an ordering of all memory operations that respects the program order of each thread

```
A = 1
print B
```

```
B = 1
print A
```

Possible sequentially consistent executions:

A = 1	A = 1	A = 1	B = 1	B = 1	B = 1
print B	B = 1	B = 1	print A	A = 1	A = 1
B = 1	print B	print A	A = 1	print A	print B
print A	print A	print B	print B	print B	print A
01	11	11	01	11	11

Sequential Consistency

- Sequential consistency is the easiest ordering guarantee for programmers to reason about
    - It’s the natural extension of the model that applies to single threaded programs
- Major programmer issue: race conditions
    - Our example code produces two different results, depending on timing
- Major systems issue: code performance
    - Insisting on program order for actual execution rules out applying some useful (single threaded) code optimizations
- Major systems issue: hardware performance
    - Insisting on program order for actual execution rules out applying some useful hardware optimizations

Memory Consistency

- The general issue is what the rules are for re-ordering reads and writes, relative to program order, even when they're to distinct memory addresses

Earlier Instruction	Later Instruction
read	read
read	write
write	read
write	write

- For instance, is it legal to execute these two instructions out of program order?

```
write X
read Y
```

Out of (Program) Order Execution: Compiler

- Both the hardware and the compiler sometimes “would like” to execute instructions out of order
- For example, moving a later read before a write:

```
for (int i=0; i<N; i++) {
    A[i] = A[i] + K;
}
```

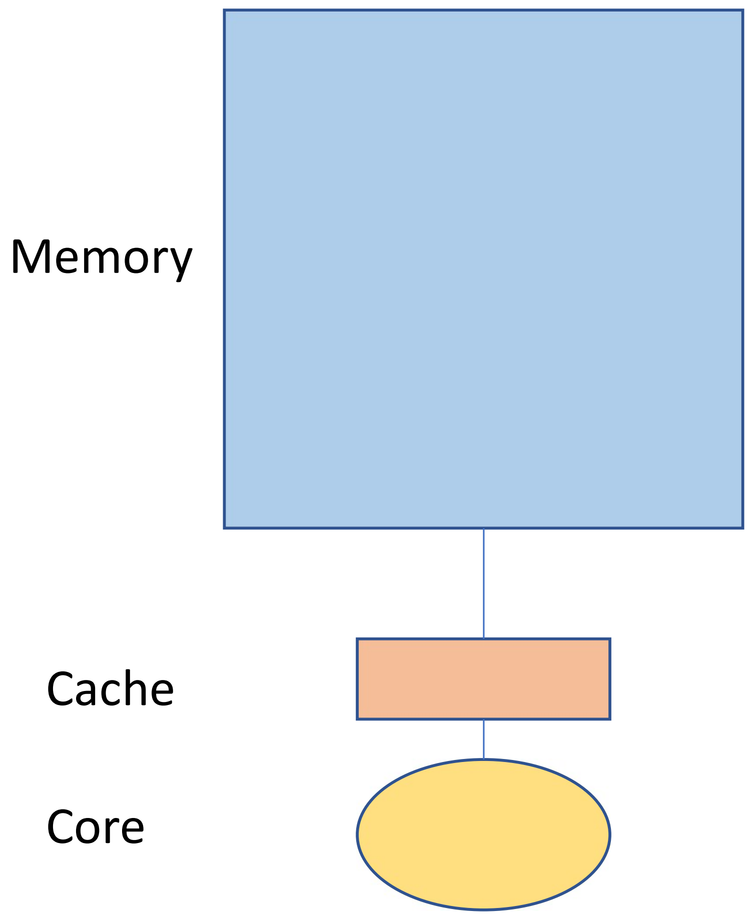
$$\Downarrow$$

```
<temp register> = K;
for (int i=0; i<N; i++) {
    A[i] = A[i] + <temp register>;
}
```

The point is that the N-1 reads of K were moved before writes they follow in program order.

It isn’t about some other thread possibly modifying K while the loop is in execution.

Out of (Program) Order Execution: Hardware



- write X
  read Y

- Suppose the write to X misses in the cache
- Cache has to fetch cache line containing X, which is slow
- Suppose the read of Y would be a cache hit
- Why should the read of Y wait for the write of X to complete?

What Can Happen if Reads Can Move Past Writes?

```
A = 1
print B
```

```
B = 1
print A
```

Possible executions:

```
A = 1
print B
B = 1
print A
```

```
A = 1
B = 1
print B
print A
```

```
A = 1
B = 1
print A
print B
```

```
B = 1
print A
A = 1
print B
```

```
B = 1
A = 1
print A
print B
```

```
B = 1
A = 1
print B
print A
```

```
print B
print A
B = 1
A = 1
```

00

```
print B
print A
A = 1
B = 1
```

00

```
print A
print B
A = 1
B = 1
```

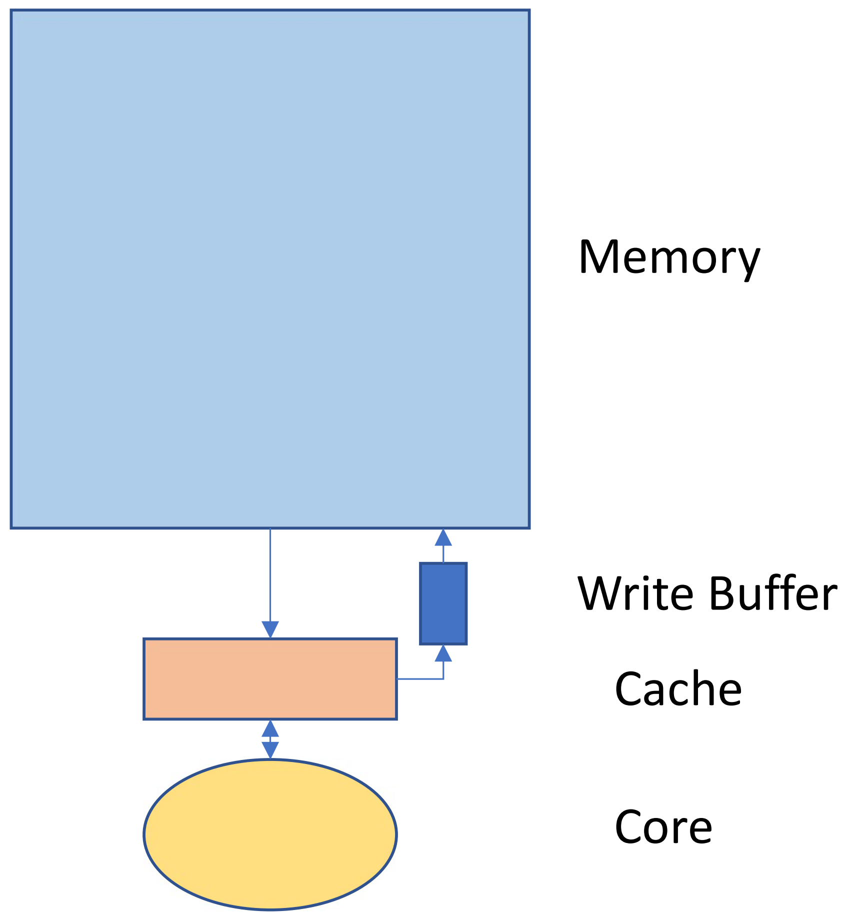
00

```
print A
print B
B = 1
A = 1
```

00

etc.

More Hardware Issues: Write Buffers



Write buffer absorbs (limited) bursts of writes.

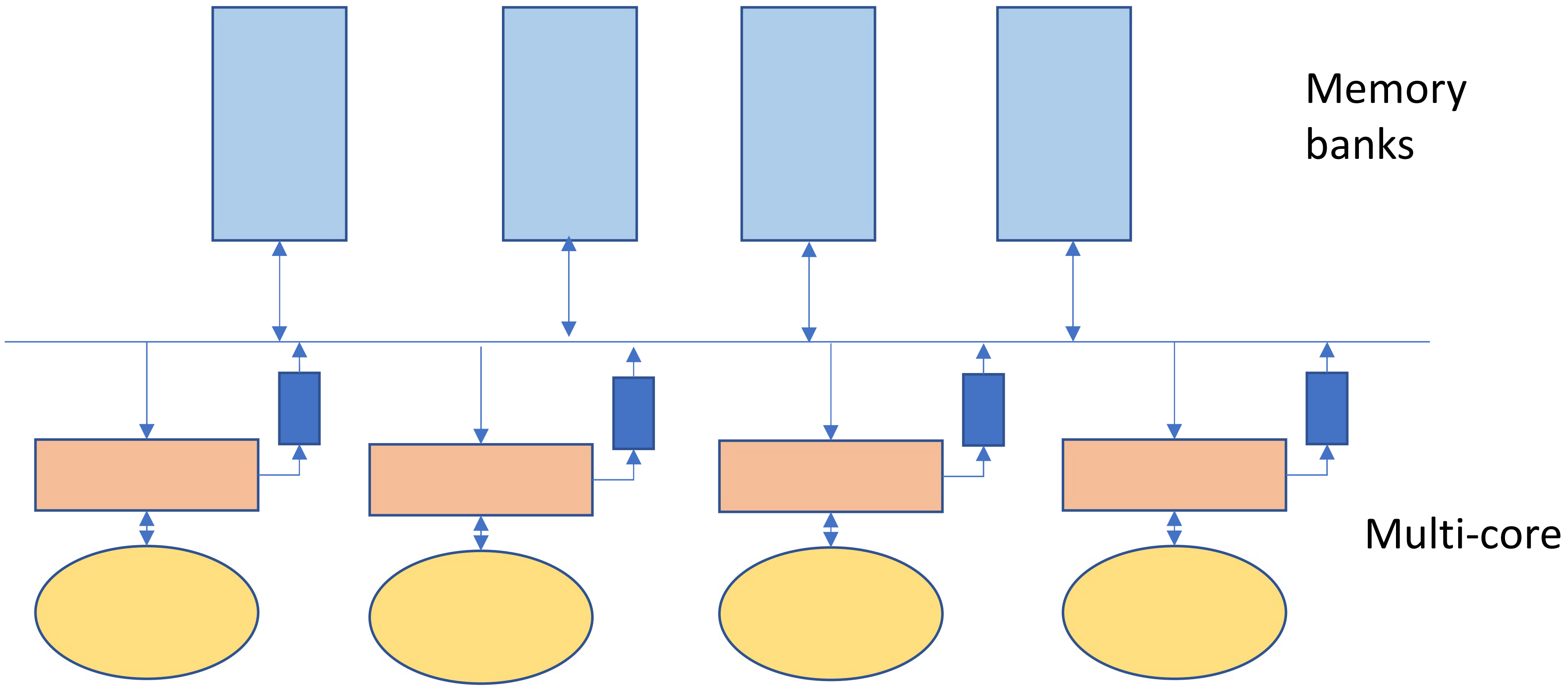
When reading a memory location, most recent value written may be in:

- cache
- write buffer
- memory

So, single core always sees "sequential consistency" for its own writes

- the value it reads is the last one it wrote
- as seen by its own reads, writes happen in order

But what about this?

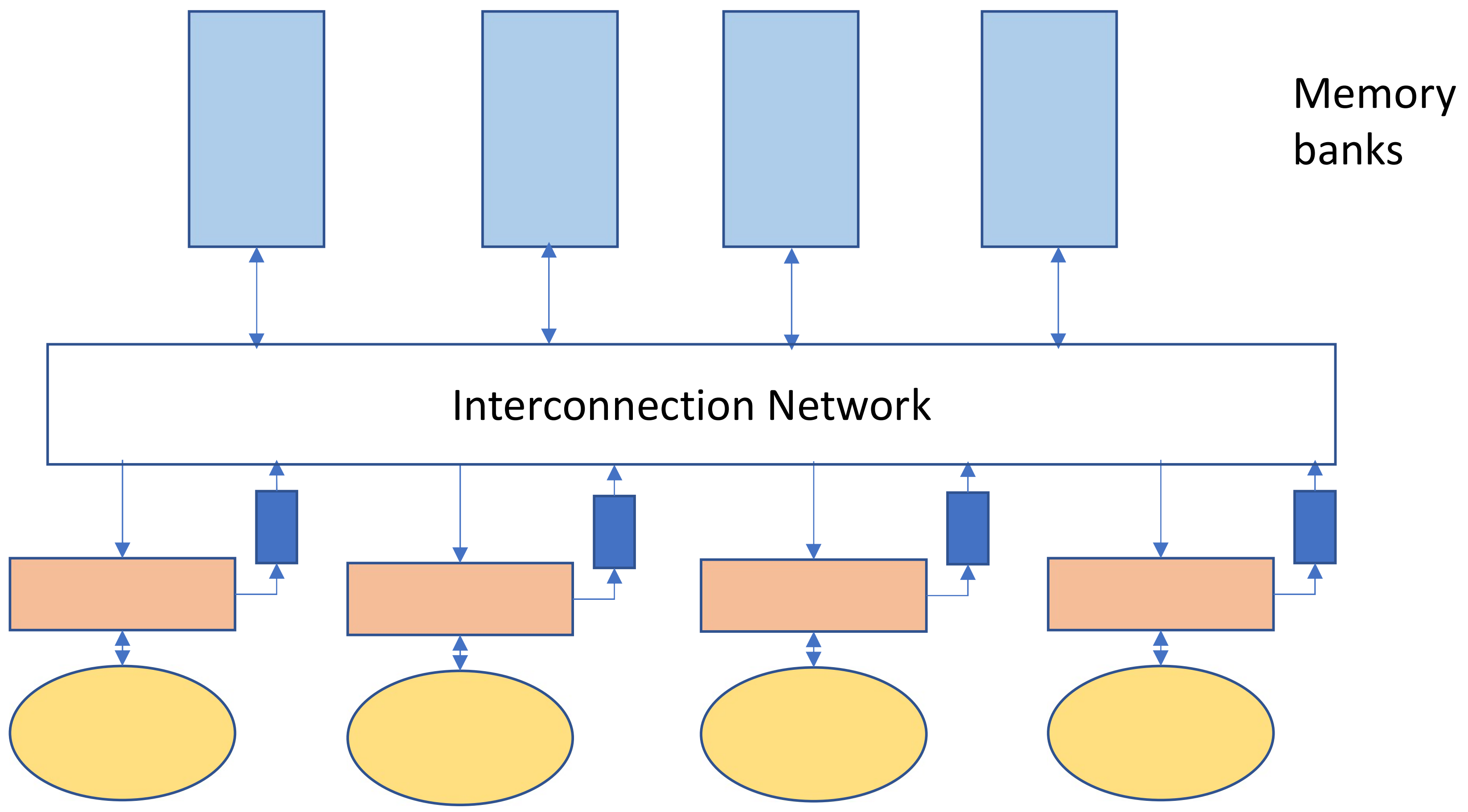


Suppose locations A and B start out with value 0.
Core 0 writes 1 to A and then 1 to B, and no one else writes.

Can core 0 read memory and find B==1 and A==0?
Can core 2 read memory and find B==1 and A==0?

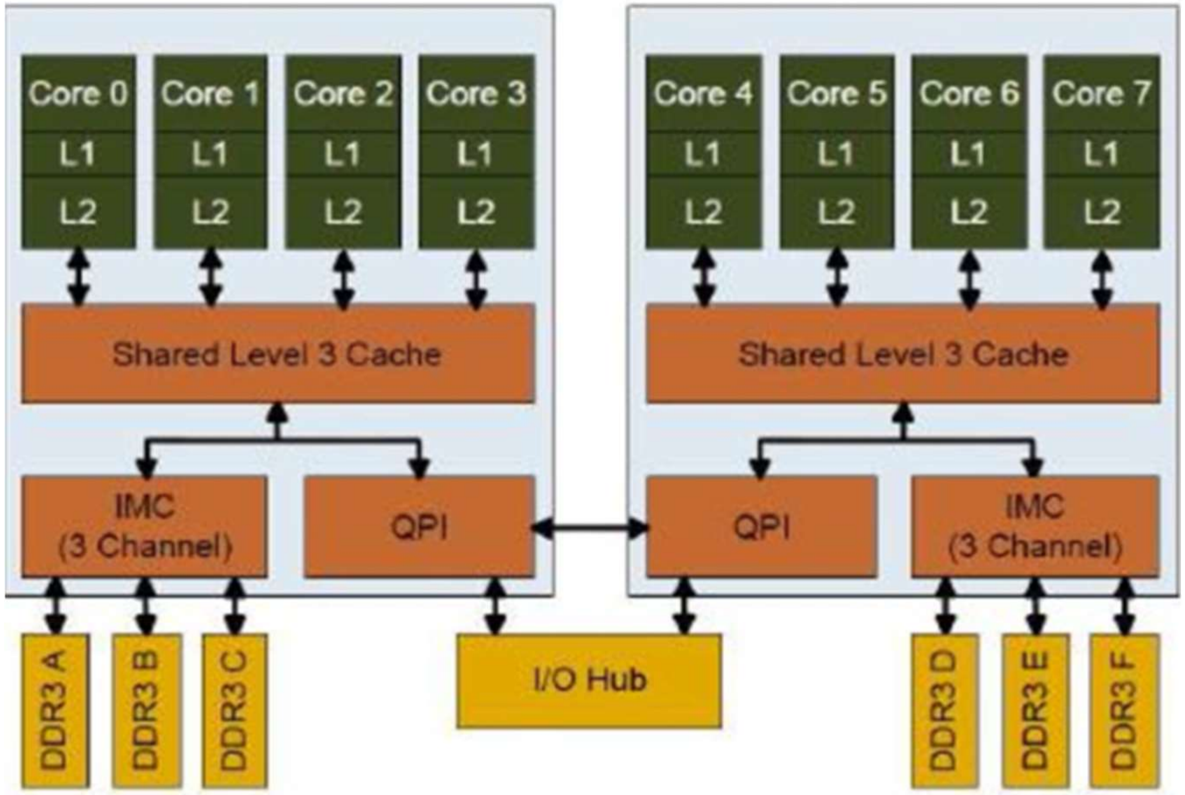
Does each buffer have to write back in program order?

It gets even worse?



There’s another ordering vs. performance issue:
Some write-back buffer may want to write X then Y in order, but the memory bank for X is busy while the memory bank for Y is idle.

And Worse: Do all cores see writes done in the same order?



Are writes “atomic”?

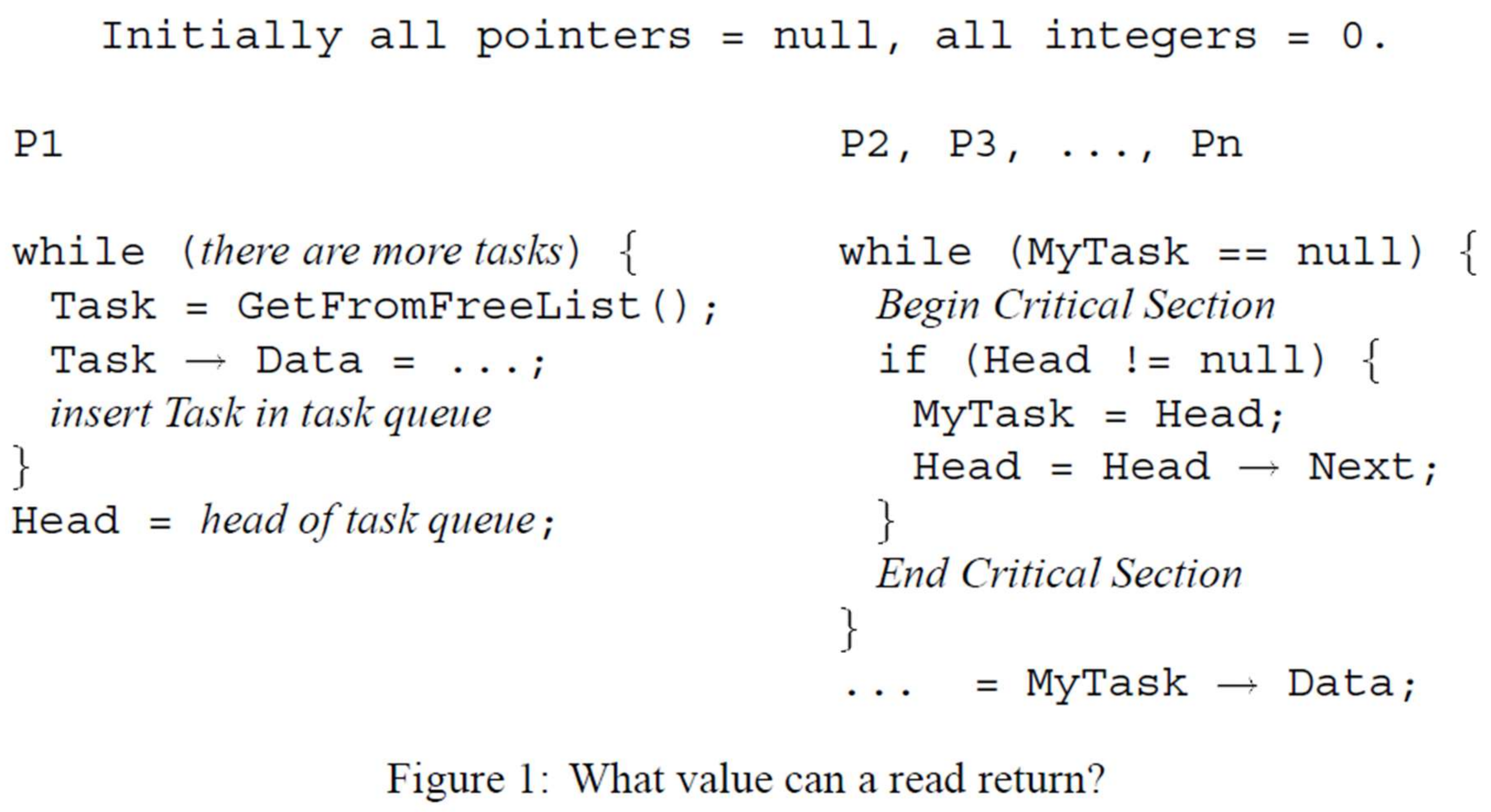
Re-ordering Summary

- The hardware would like to be free to re-order all operations, respecting only data flow constraints
- The compiler would like to be free to re-order, but only in ways that respects single-threaded data flow constraints
  - X = Y + 3; Z = X/2;   // cannot be reordered because data flows from 1st to 2nd

- Result:
  - The hardware makes some kind of memory consistency guarantees
  - The language (may) provide some kind of memory consistency guarantees
  - The compiler is responsible for translating code written to language spec to operate as expected on whatever the hardware provides

- It’s still very complicated

Can This Fail?



```
    Initially all pointers = null, all integers = 0.

P1                                        P2, P3, ..., Pn

while (there are more tasks) {            while (MyTask == null) {
  Task = GetFromFreeList();                 Begin Critical Section
  Task → Data = ...;                        if (Head != null) {
  insert Task in task queue                   MyTask = Head;
}                                             Head = Head → Next;
Head = head of task queue;                  }
                                            End Critical Section
                                          }
                                          ...  = MyTask → Data;
```

Figure 1: What value can a read return?

From WRL-95-7

Example: Total Store Ordering (TSO)

- Strongest of the consistency models (below sequential consistency)
- Implemented by x86

- Respects program order for
  - read before read
  - read before write
  - write before write
- May not respect program order for
  - write before read

Writing Correct Multithreaded Programs

- The language (may) provide some kind of memory consistency guarantees
- The compiler is responsible for translating code written to language spec to operate as expected on whatever the hardware provides

- It's still very complicated

std::memory_order

Defined in header `<atomic>`

```
typedef enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
} memory_order;
```

(since C++11)
(until C++20)

```
enum class memory_order : /*unspecified*/ {
    relaxed, consume, acquire, release, acq_rel, seq_cst
};
inline constexpr memory_order memory_order_relaxed = memory_order::relaxed;
inline constexpr memory_order memory_order_consume = memory_order::consume;
inline constexpr memory_order memory_order_acquire = memory_order::acquire;
inline constexpr memory_order memory_order_release = memory_order::release;
inline constexpr memory_order memory_order_acq_rel = memory_order::acq_rel;
inline constexpr memory_order memory_order_seq_cst = memory_order::seq_cst;
```

(since C++20)

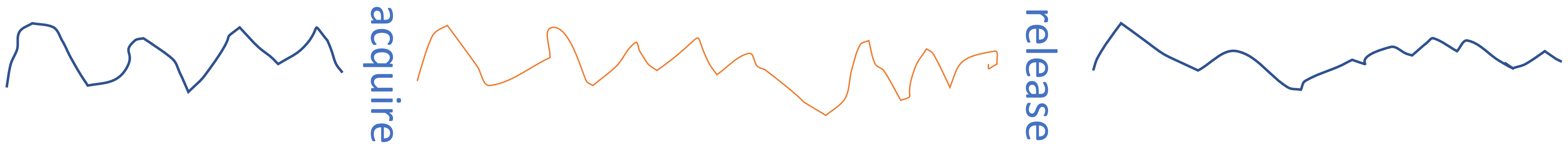
`std::memory_order` specifies how memory accesses, including regular, non-atomic memory accesses, are to be ordered around an atomic operation. Absent any constraints on a multi-core system, when multiple threads simultaneously read and write to several variables, one thread can observe the values change in an order different from the order another thread wrote them. Indeed, the apparent order of changes can even differ among multiple reader threads. Some similar effects can occur even on uniprocessor systems due to compiler transformations allowed by the memory model.

The default behavior of all atomic operations in the library provides for *sequentially consistent ordering* (see discussion below). That default can hurt performance, but the library's atomic operations can be given an additional `std::memory_order` argument to specify the exact constraints, beyond atomicity, that the compiler and processor must enforce for that operation.

How Can You Possibly Cope?

- The compiler and/or hardware provides mechanisms to restrict reordering
  - Memory fence

- Memory fence acquire
  - a read such that all reads and writes later in program order are executed after in memory order
- Memory fence release
  - a write such that all reads and writes that come before in program order are also before in memory order



lock()/unlock() implementations do acquire/release

Programming Convention

- Guard all uses of shared memory with acquire/release pairs
- lock()/unlock() implementations do acquire/release

T1

```
lock()
A = 1
print B
unlock()
```

T2

```
lock()
B = 1
print A
unlock()
```

Possible Programming Convention

- Program in the style of monitors
  - Encapsulate shared data structures in classes
  - Synchronize all operations on the shared data structure, even just reads
  - Acquire a lock at the beginning of each method, release it at the end

- Advantage:
  - You can reason about your program as though execution guarantees sequential consistency

- Disadvantages
  - Locking issues: granularity, deadlock
  - Performance issues
    - locking overhead
    - memory barrier costs