Time, clocks, and consistency and the JMM

Jeff Chase
Duke University
JMM

• We’re discussing consistency in the context of the Java Memory Model [Manson, Pugh, Adve, PoPL 05].

• The question: What is the right memory abstraction for multithreaded programs?
  – Admits an efficient machine implementation.
  – Admits compiler optimizations.
  – (Maximizes allowable concurrency.)
  – Runs correct programs correctly.
  – Conforms to Principle of No Unreasonable Surprises for incorrect programs.
  – Or: “No wild shared memory.”
  – (Easy for programmers to reason about.)
JMM: Three Lessons

1. Understand what it means for a program to be correct.
   - Synchronize! Use locks to avoid races.

2. Understand the memory model and its underpinnings for correct programs.
   - Happens-before, clocks, and all that,
   - Expose synchronization actions to the memory system.
   - Synchronization order induces a happens-before order.

3. Understand the need for a rigorous definition of the memory model for unsafe programs. Since your programs are correct, and we aren’t writing an optimizing compiler, we can “wave our hands” at the details.
Concurrency and time

- Multicore systems and distributed systems have no global linear time.
  - Nodes (or cores) “see” different subsets of events.
  - **Events**: messages, shared data updates, inputs, outputs, synchronization actions.
  - Some events are **concurrent**, and nodes that do see them may see them in different orders.

- If we want global linear time, we must make it.
  - Define words like “before”, “after”, “later” **carefully**.
  - Respect “natural” ordering constraints.
Concurrency and time

A and B are cores, or threads, or networked nodes, processes, or clients. Each executes in a logical sequence: time goes to the right.

Occasionally, one of them generates an event that is visible to the other (e.g., a message or a write to memory).

**Consistency** concerns the order in which participants observe such events. Some possible orders “make sense” and some don’t. A **consistency model** defines what orders are allowable. Multicore memory models and JMM are **examples** of this concept.
Concurrency and time

What do these words mean?  
*after? last? subsequent? eventually?*

Time, Clocks, and the Ordering of Events in Distributed systems, by Leslie Lamport, CACM 21(7), July 1978
Same world, different timelines

Which of these happened first?

- e1a is concurrent with e1b
- e3a is concurrent with e3b and e4

This is a partial order of events.
Lamport *happened-before* (→)

1. If $e_1, e_2$ are in the same process/node, and $e_1$ comes before $e_2$, then $e_1 \rightarrow e_2$.

   - Also called program order

Time, Clocks, and the Ordering of Events in Distributed Systems, by Leslie Lamport, CACM 21(7), July 1978
Over 8500 citations!
Lamport *happened-before* (→)

2. If \( e_1 \) is a message send, and \( e_2 \) is the corresponding receive, then \( e_1 \rightarrow e_2 \). - The receive is “caused” by the send event, which happens before the receive.

_Time, Clocks, and the Ordering of Events in Distributed systems_, by Leslie Lamport, CACM 21(7), July 1978
3. $\rightarrow$ is transitive

*happened-before* is the transitive closure of the relation defined by #1 and #2

potential causality

Time, Clocks, and the Ordering of Events in Distributed systems, by Leslie Lamport, CACM 21(7), July 1978
Two events are **concurrent** if neither happens-before the other.
Significance of happens-before

• **Happens-before** defines a *partial order* of events.
  – Based on some notion of a causal event, e.g., message send
  – These events capture causal dependencies in the system.

• In general, execution orders/schedules must be “consistent with” the happens-before order.

• Key point of JMM and multicore memory models: *synchronization accesses* are causal events!
  – JMM preserves happens-before with respect to lock/unlock.
  – Multi-threaded programs that use locking correctly see a consistent view of memory.
Thinking about data consistency

• Let us choose a total (sequential) order of data accesses at the storage service.
  – Sequential schedules are easy to reason about, e.g., we know how reads and writes should behave.
  – \( R(x) \) returns the “last” \( W(x)=v \) in the schedule

• A data consistency model defines required properties of the total order we choose.
  – E.g., we require the total order to be consistent with the “natural” partial order (\( \rightarrow \)).
  – Application might perceive an inconsistency if the ordering violates \( \rightarrow \), otherwise not detectable.

• Some orders are legal in a given consistency model, and some orders are not.
Clocks: a quick overview

- **Logical clocks** *(Lamport clocks)* number events according to *happens-before* (*→*).
  - If e₁ → e₂, L(e₁) < L(e₂)
  - No relation defined if e₁ and e₂ are concurrent.

- **Vector clocks** label events with a vector V, where V(e)[i] is the logical clock of the latest event e₁ in node i such that e₁ → e.
  - V(e₂) dominates V(e₁) iff e₁ → e₂.
  - Two events are **concurrent** iff neither vector clock dominates the other.
  - You’ll see this again…
Same world, unified timelines?

This is a total order of events. Also called a sequential schedule. It allows us to say “before” and “after”, etc. But it is arbitrary.
Same world, unified timelines?

Here is another total order of the same events. Like the last one, it is consistent with the partial order: it does not change any existing orderings; it only assigns orderings to events that are concurrent in the partial order.
Example: sequential consistency

For all of you architects out there…

**Sequential consistency model** [Lamport79]:
- Memory/SS chooses a global total order for each cell.
- Operations from a given P are in program order.
- (Enables use of lock variables for mutual exclusion.)
How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Program

LESLEY LAMPORT

Abstract—Many large sequential 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. Additional conditions are given which do guarantee that a computer correctly executes multiprocess programs.

1979: An early understanding of multicore memory consistency. Also applies to networked storage systems.
Sequential consistency is too strong!

- Sequential consistency requires the machine to do a lot of extra work that might be unnecessary.
- The machine must make memory updates by one core visible to others, even if the program doesn’t care.
- The machine must do some of the work even if no other core ever references the updated location!
- Can a multiprocessor with a weaker ordering than sequential consistency still execute programs correctly?
- Answer: yes. Modern multicore systems allow orderings that are weaker, but still respect the happens-before order induced by synchronization (lock/unlock).
Memory ordering

- Shared memory is complex on multicore systems.
- Does a load from a memory location (address) return the latest value written to that memory location by a store?
- What does “latest” mean in a parallel system?

It is common to presume that load and store ops execute sequentially on a shared memory, and a store is immediately and simultaneously visible to load at all other threads. But not on real machines.
**Memory ordering**

- A **load** might fetch from the local cache and not from memory.
- A **store** may buffer a value in a local cache before draining the value to memory, where other cores can access it.
- Therefore, a **load** from one core does not necessarily return the “latest” value written by a **store** from another core.

A trick called Dekker’s algorithm supports mutual exclusion on multi-core without using atomic instructions. It assumes that **load** and **store** ops on a given location execute sequentially. **But they don’t.**
“Sequential” Memory ordering

A machine is **sequentially consistent** iff:

- Memory operations (loads and stores) appear to execute in some sequential order on the memory, and
- Ops from the same core appear to execute in program order.

No sequentially consistent execution can produce the result below, yet it can occur on modern machines.

To produce this result: 4<2 (4 **happens-before** 2) and 3<1. No such schedule can exist unless it also reorders the accesses from T1 or T2. Then the reordered accesses are out of program order.
The first thing to understand about memory behavior on multi-core systems

- Cores must see a “consistent” view of shared memory for programs to work properly. A machine can be “consistent” even if it is not “sequential”. But what does it mean?
- Synchronization accesses tell the machine that ordering matters: a happens-before relationship exists. Machines always respect that.
  - Modern machines work for race-free programs.
  - Otherwise, all bets are off. Synchronize!

The most you should assume is that any memory store before a lock release is visible to a load on a core that has subsequently acquired the same lock.
Thread A

1. y = 1
2. lock M
3. x = 1
4. unlock M

Everything before the unlock on M...

Thread B

...visible to everything after the lock on M

1. lock M
2. i = x
3. unlock M
4. j = y
What’s a race?

• Suppose we execute program \( P \).
• The events are synchronization accesses (lock/unlock) and loads/stores on shared memory locations, e.g., \( x \).
• The machine and scheduler choose a schedule \( S \).
• \( S \) imposes a total order on accesses for each lock, which induces a \texttt{happens-before} order on all events.
• Suppose there is some \( x \) with a \texttt{concurrent} load and store to \( x \). (The load and store are \texttt{conflicting}.)
• Then \( P \) has a \texttt{race}. A race is a bug. \( P \) is unsafe.
• \textbf{Summary}: a race occurs when two or more conflicting accesses are concurrent.
Synchronization order

An execution schedule defines a total order of synchronization events (at least on any given lock/monitor): the synchronization order.

Different schedules of a given program may have different synchronization orders.

Just three rules govern synchronization order:

1. Events within a thread are ordered.
3. The order is transitive:
   if \((A < B)\) and \((B < C)\) then \(A < C\).

Purple’s unlock/release action synchronizes-with the subsequent lock/acquire.
An execution schedule defines a partial order of program events. The ordering relation (\(<\)) is called happens-before.

Two events are concurrent if neither happens-before the other in the schedule.

Just three rules govern happens-before order:

1. Events within a thread are ordered.
2. Mutex handoff orders events across threads: the release \(\#N\) happens-before acquire \(\#N+1\).
3. Happens-before is transitive: if \((A < B)\) and \((B < C)\) then \(A < C\).

Machines may reorder concurrent events, but they always respect happens-before ordering.
“Happens-before is the transitive closure of program order and synchronization order.”

“A program is said to be correctly synchronized or data-race-free iff all sequentially consistent executions of the program are free of data races.” [According to happens-before.]
JMM model

The “simple” JMM happens-before model:

- A read cannot see a write that *happens after* it.
- If a read sees a write (to an item) that happens before the read, then the write must be the last write (to that item) that happens before the read.

Augment for sane behavior for unsafe programs (loose):

- Don’t allow an *early* write that “depends on a read returning a value from a data race”.
- An *uncommitted* read must return the value of a write that happens-before the read.
The point of all that

- We use special atomic instructions to implement locks.
- E.g., a TSL or CMPXCHG on a lock variable `lockvar` is a synchronization access.
- Synchronization accesses also have special behavior with respect to the memory system.
  - Suppose core C1 executes a synchronization access to `lockvar` at time $t_1$, and then core C2 executes a synchronization access to `lockvar` at time $t_2$.
  - Then $t_1 < t_2$: every memory store that happens-before $t_1$ must be visible to any load on the same location after $t_2$.
- If memory always had this expensive sequential behavior, i.e., every access is a synchronization access, then we would not need atomic instructions: we could use “Dekker’s algorithm”.
- We do not discuss Dekker’s algorithm because it is not applicable to modern machines. (Look it up on wikipedia if interested.)
7.1. LOCKED ATOMIC OPERATIONS
The 32-bit IA-32 processors support locked atomic operations on locations in system memory. These operations are typically used to manage shared data structures (such as semaphores, segment descriptors, system segments, or page tables) in which two or more processors may try simultaneously to modify the same field or flag.

Note that the mechanisms for handling locked atomic operations have evolved as the complexity of IA-32 processors has evolved.

Synchronization mechanisms in multiple-processor systems may depend upon a strong memory-ordering model. Here, a program can use a locking instruction such as the XCHG instruction or the LOCK prefix to insure that a read-modify-write operation on memory is carried out atomically. Locking operations typically operate like I/O operations in that they wait for all previous instructions to complete and for all buffered writes to drain to memory.

This is just an example of a principle on a particular machine (IA32): these details aren’t important.
All processors share a single global memory
Symmetric Multiprocessing – SMP,  
Multi-core processor PC

Processors don’t have shared memory but communicate via network
Multicomputer Multiprocessing – MPP,
Commodity compute clusters