Introduction to Parallel & Distributed Programming

Lec 09—Memory Consistency (Linearisability, Sequential Consistency)

Subodh Sharma | Jan 27, 2026

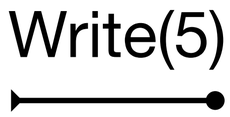
RECAP: What is a Memory Consistency Model?

- **Set of rules** for how memory operations **on shared variables** become **visible** to different threads in a system.
- The contract is between the H/w, the compiler and the programmer:
  - **The programmer**: “I have developed the code with some intuition so that I don’t get bugs”
  - **The Compiler**: “I will optimise your code, which means reordering some reads and writes differently from the way you programmed it”
  - **The Hardware:** “I will execute instructions as fast as possible, which may mean running them out of order”

RECAP: Linearisability

Strongest Memory Consistency Model

- An operation — has a start and has an end!
  - It starts with an invocation, and ends with a response

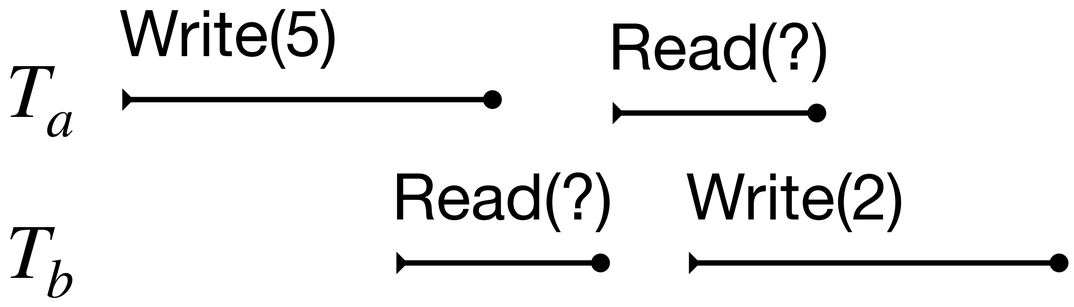


- **Rules of Linearisability:**
  - **Atomicity:** Every operations appears to happen **instantaneously** (between invocation & response)
  - **Order:** If operation A completes before operation B then A must appear to **happen before** operation B in the global history of events
  - **Correctness:** The final outcome must match the **sequential version**
  - **The order is not just logical but also respects real-time**

RECAP: Linearisability

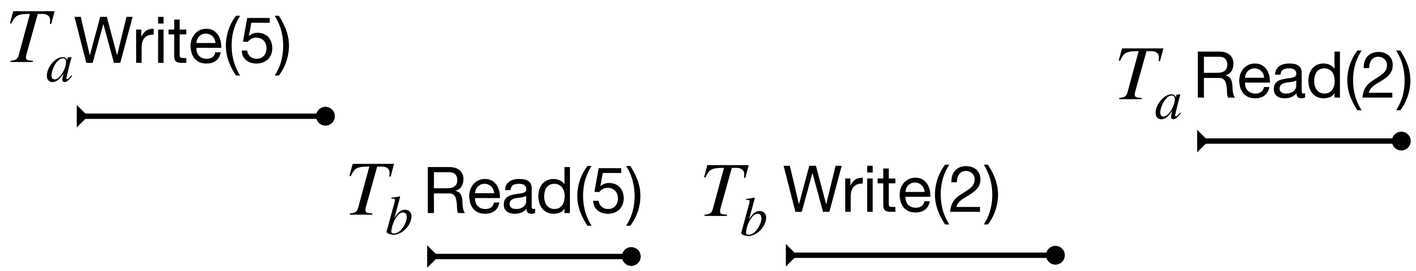
Visualising Linearisability

- Ordering is well-defined; **no overlap**
  - **Imagine a shared register**



- **Equivalent Sequential History**

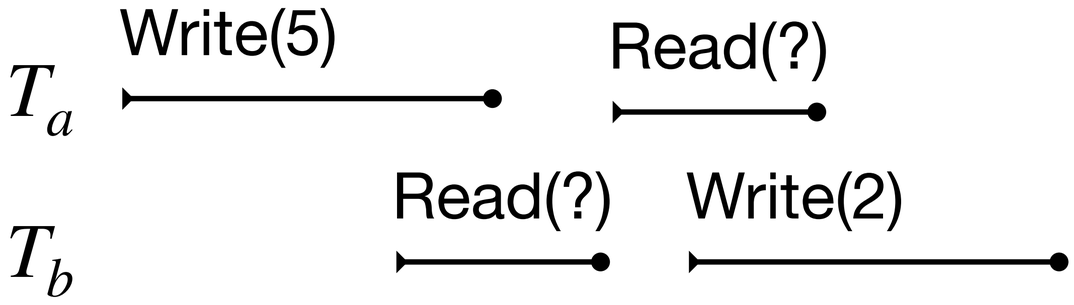
Valid, linearisable history!



RECAP: Linearisability

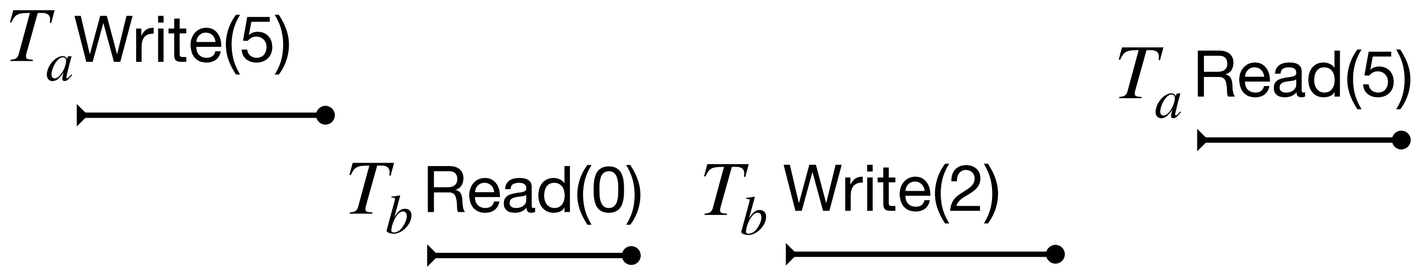
Visualising Linearisability

- Ordering is well-defined; **no overlap**
  - **Imagine a shared register**



- **Equivalent Sequential History**

Invalid, nonlinearisable history!



Linearisation Point

- It is the instant where the method takes effect
  - Example: For lock-based implementations each method's critical section can serve as its linearisation point
- Let us look at the formalisation of the problem
  - History H is a sequence of operation invocation and response events
  - H is **complete** if every invocation in H has a response in H
    - complete(H) could be a subsequence of H that matches the above definition
  - H is **sequential** if the first event of H is an invocation and each invocation (except possibly the last) is **immediately** followed by a matching response.

Linearisability

Realising OR Verifying Linearisability

- CPU cache update latencies may be non-uniform;
  - One would need expensive memory barriers to implement linearisability
- Network Latency: In distributed systems, this may impact visibility
  - Achieving linearisability via **consensus** such as **Paxos** or **Raft**
- **Verification: Hard problem**
  - Identify linearisation points
  - Show that all possible ordering of linearisation points lead to sequential (valid) histories

Linearisation — Formalisation

- History H is well-formed if each thread's sub-history is sequential
- Let $\rightarrow$ be a partial order and $<$ be a total order, then
  - $x \rightarrow y \implies x < y$
- Defn (Linearisability): H is linearisable if it has an extension H' and there is a legal sequential history S, s.t.
  - complete(H') is equivalent to S
  - $m_0 \rightarrow m_1$ in H then the same is true in SH

Linearisation — Formalisation

- **Thm1:** H is linearisable **if and only if** for each object x, $H|x$ is linearisable
  - Proof by induction on the number of operations/method-calls in H' (extension of H with possible response calls)
  - Compositionality is important — **why**?
    - It allows for modular design of concurrent systems
      - Linearisable objects can be verified and implemented in isolation

Progress Conditions

Linearisability is non-blocking

- A method/operation is **wait-free** if it guarantees that every call finishes its execution in **finite number of steps**
  - **Bounded wait-free:** There is an upper-bound in the number of steps
- Being wait-free is an example of a non-blocking progress condition
  - Arbitrary and unexpected delay by one thread doesn't prevent others from progressing
- Other progress conditions — **lock-free, deadlock-free & starvation-free (revist when we discuss synchronisations)**

Sequential Consistency

Weaker than Linearisability

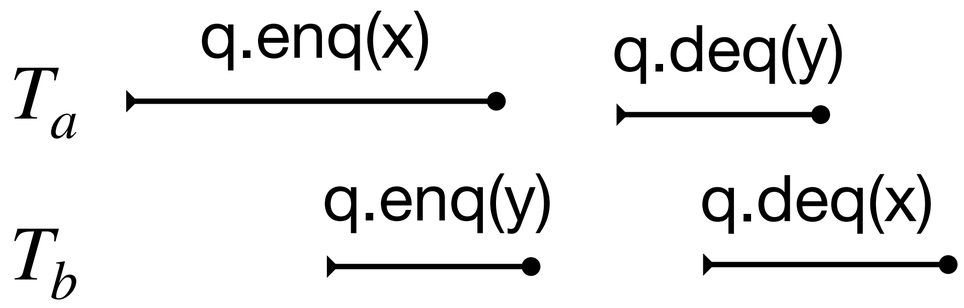
- C1: operations should happen in one-at-a-time sequential order
- C2: operations should appear to take effect in program order
- C1 AND C2 define Sequential Consistency

> "A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]

Sequential Consistency

Weaker than Linearisability

- NOTE: There is no requirement to depend on real global time

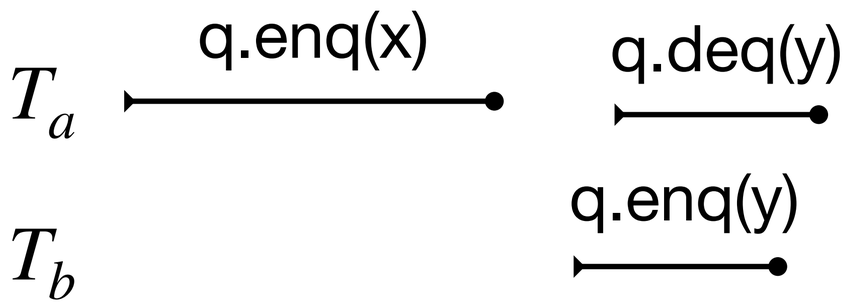


- Two possible SC orders to justify the above execution:
  - $T_a : q . enq(x) \text{ -> } T_b : q . enq(y) \text{ -> } T_b : q . deq(x) \text{ -> } \ T_a : q . deq(y)$

Sequential Consistency: Is Linearisable?

And NOT REAL TIME

- NOTE: There is no requirement to depend on real global time

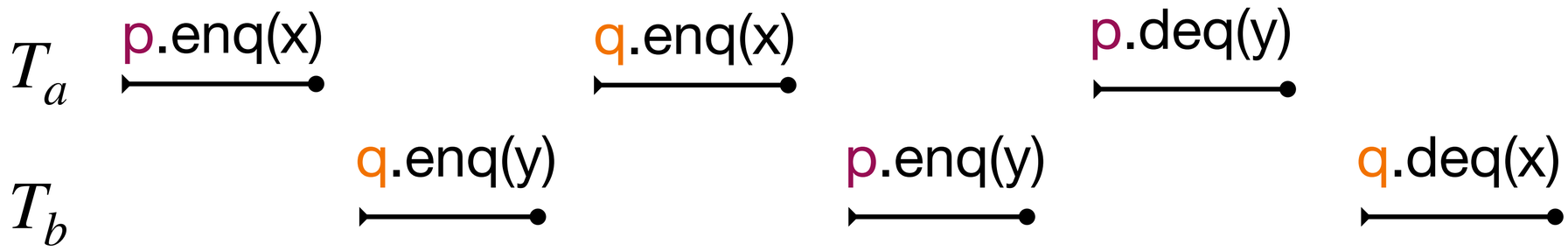


- SC execution to justify the above result:
  - $T_b : q . enq(y) \text{ -> } T_a : q . enq(x) \text{ -> } T_a : q . deq(y)$

This movement not possible under linearisability

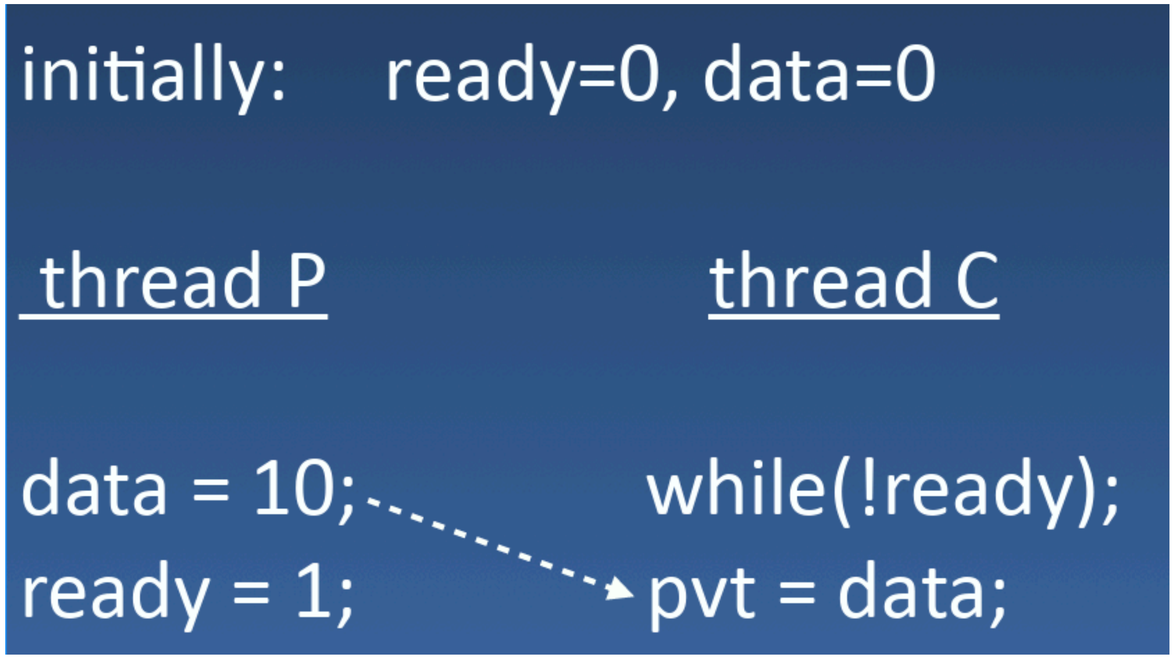
Sequential Consistent: Is Composable?

Weaker than Linearisability



- Not hard to see that Queue objects p and q are individually SC
  - **But the composition is NOT! — WHY?**

Applying SC in Real Programs



- **Threads always see the most recent written values constrained by the thread-order**
  - If read==1 then data has to be 10

Consistency

- Consistency is about global state of the memory (not per-variable)
  - Per-variable consistency is coherency!
- Inconsistencies can arise
  - Due to compiler re-ordering instructions
  - Due to N/w or H/w reordering them
- To address inconsistencies
  - Use **synchronisations**