1 Introduction

In this lecture, we prove a lower bound on the length of a READ in a linearizable Memory Consistency System. Then we briefly consider consistency systems that are weaker than Sequential Consistency. Finally, we introduce the Consensus Problem and the Consensus Hierarchy.

2 A time lower bound on linearizable READs

For a Sequentially Consistent implementation, there is no lower bound on either READ or WRITE. There exist implementations where the time required for either operation is 0. The only lower bound we’ve shown is that \( t_{\text{READ}} + t_{\text{WRITE}} \geq d \), where \( d \) is the time required for a message to be transmitted between two neighboring processors. Nevertheless, in the previous lecture, we have a lower bound for the time taken for a linearizable implementation of READ. The technique we used to show that there was, indeed a lower bound on a READ was called a Shifting Technique. We created an execution, then shifted the order of some of the operations, and also used the existence of message delivery uncertainty to change the timing of message delivery. As a result, we obtained two distinct executions that processors could not tell apart, which produced a contradiction. We will use a similar Shifting Technique now, to prove a lower bound on WRITEs.

**Theorem 1** For any linearizable Memory Consistency System that provides a Read/Write object \( X \) that can be read by two processors and written to by a third, \( t_{\text{READ}} \geq u/4 \). (Here, \( u \) is the uncertainty parameter of message delivery time.)

**Proof:** Let \( X \) be written by \( p_0 \), and read by \( p_1 \) and \( p_2 \). We assume for the sake of contradiction that \( t_{\text{READ}} < u/4 \).

**Proof Idea:** We start with an execution, and then shift that execution just enough that no process can tell the difference. We don’t need to know the exact messages that are sent and received, just the order in which processors commit to reads and writes.

The execution we build—and then shift—is more complex than the execution we built (and shifted) when we proved a time lower bound on the READ operation. In that argument, we could place the writes, and not care how long the reads took. This time around, we will have a single write, with overlapping reads. We didn’t need overlapping reads and writes in the previous argument, but now we do.
The values read \((v_0, v_1, \ldots)\) have to be monotonically increasing, from 0's to 1's. Suppose the change is at \(v_3\). Then the permutation \(\pi = r_0 \pi_1 r_2 \pi_3 \cdots r_k r_{k+1}\) satisfies linearizability. (Here, \(r_i\) is \(\text{READ}(v_i)\) and \(w\) is \(\text{WRITE}(X, 1)\).)

Formally, we assume there exists an execution \(\alpha\) in which all message delays are \(d - u/2\). Let \(k = \lceil t_{\text{WRITE}}/u \rceil\). So the last write returns before \(t = (4k + 1)(u/4)\).

For any \(i\), at time \(2i(u/4), 0 \leq i \leq 2k\), \(p_1\) performs a \(\text{READ}\) that returns before \((2i + 1)(u/4)\), since \(t_{\text{READ}} < u/4\). At time \((2i + 1)(u/4),\) for all \(i, 0 \leq i \leq 2k\), \(p_2\) performs a read, returning before \((2i + 2)(u/4)\), since \(t_{\text{READ}} < u/4\).

Without loss of generality, we assume that \(j\) is even, where \(v_j = 0\) and \(v_{j+1} = 1\). Any permutation of operations in \(\alpha\) that linearizes those operations must place the write \(w\) in between \(r_j\) and \(r_{j+1}\). We now produce an execution \(\beta\) which cannot be linearized by such a permutation, even though \(\beta\) is indistinguishable from \(\alpha\), which leads to contradiction.

We construct \(\beta\) by leaving the operations of \(p_0\) and \(p_1\) the same as in \(\alpha\), and by shifting the operations of \(p_2\) by \(u/2\) earlier. This can be written \(\beta = (\alpha, (0, 0, -u/2))\). We also modify the message delays for \(\beta\), as follows.

- \(p_0\) to \(p_1\), \(p_1\) to \(p_0\) same as in \(\alpha\)
- \(p_0\) to \(p_2\), \(p_1\) to \(p_2\) \(d - u\)
- \(p_2\) to \(p_0\), \(p_2\) to \(p_1\) \(d\)

These message delays are still admissible, as they are within the range \([d - u, d]\). Note that (1) \(p_2\) receives and sends all messages \(u/2\) time earlier in global time in \(\beta\), and (2) \(p_0\) and \(p_1\) receive and send at the same time as in \(\alpha\). So the processors cannot distinguish \(\alpha\) from \(\beta\), and \(\beta\) should be linearizable using the same permutation of operations that demonstrated linearizability of \(\alpha\). But now:

\[ r_1 < r_0 < r_3 < r_2 < \cdots < r_{j+1} < r_j \]

This is a contradiction, as \(r_{j+1} = 1\) and \(r_j = 0\).

3 Relaxed consistency models

Sequential Consistency is, of course, weaker than Linearizability, but it is still “too strong,” in the sense that it reduces the use of performance optimizations. In a nutshell, the problem is that Sequential Consistency requires the existence of a global sequence of operations, and many performance optimizations depend on the ability to execute operations in whichever order is best at the moment, as long as the order of those operations does not matter. (For example, two reads of the same memory location can happen in either order, as neither read changes anything about the system.) Requiring a global sequence reduces the benefit of multiprocessors, because it would be nice to be able to hand off work to each processor, and allow that work to be processed concurrently, with only limited communication between processors. We would like to be able to parallelize operations. There are several relaxed consistency models that allow for this. We briefly take note of two.
Weak Ordering  This model defines two types of operations: data ops and synchronization ops. The synchronization ops create barriers between operations on one side and the other. Within the barriers, the data ops can happen in any order. Program order between two operations is preserved if at least one is a synchronization op.

Release Consistency  This model is part of the Stanford DASH multiprocessor. It includes the synchronization ops acquire and release.

4 The Consensus Problem and the Consensus Hierarchy

4.1 The Consensus Problem

The Consensus Problem is a synchronization primitive that more or less formalizes the idea that “every processor agrees on the same value.” It is defined using three conditions. Each processor $p_i$ starts with an input $x_i$ and ends with an output $y_i$.

Termination  In every admissible execution, $y_i$ is eventually assigned a value for every non-faulty processor $p_i$.

Agreement  In every execution, if $y_i$ and $y_j$ have been assigned, then $y_i = y_j$ for all nonfaulty $p_i, p_j$.

Validity  In every execution, if $y_i = v$ for some nonfaulty $p_i$, then $v$ must be an input value of some processor.

Termination and Agreement ensure that all correct processors eventually halt with an agreed-upon value. Validity disallows use of uninteresting algorithms like, “Always choose value 0 no matter what.”

4.2 The Consensus Hierarchy

We can use the Consensus Problem as a primitive to compare the relative power of synchronization operations. First, two definitions.

A concurrent object is wait-free if every invocation finishes in a finite number of steps generating a response. A concurrent object is lock-free if infinitely often some invocation finishes in a finite number of steps. Wait-freedom is a form of complete fault tolerance. Any invocation will generate a response even if all other processors crash. So a wait-free algorithm is an $(n - 1)$-resilient algorithm.

We would like to be able to formalize the notion that read/write objects are intrinsically weaker than read/modify/write objects. More generally, we’d like some way to compare two different shared objects $X$ and $Y$. One way to do this is to ask whether there is a wait-free simulation of objects using only objects of type $X$ and read/write objects. However, this question requires a
different answer for every distinct pair of objects. Instead, we’d like a more general classification, so we ask a different question.

*Can we construct a wait-free algorithm for n-processor consensus using only share objects of type X and read/write objects?*

If the answer that question is YES for $n = k$ and NO for $n = k + 1$ then we say that $X$ has consensus number $k$. This notion of consensus number induces a Consensus Hierarchy, which we will look at in more detail next lecture.