Lecture Notes for CSCI 351: Distributed Computing
Set 7-Register Strengthening

Professor Talmage

April 12, 2023

One of the many uses of simulation is to provide stronger objects in a system that actually only provides weaker ones. We previously discussed consensus objects and universality in wait-free systems, but now that we have a more formal view of simulations, we will zoom in to consider different types of $Read/Write$ registers and how we can get stronger versions from weaker ones.

1 Model

Our communication system is now a wait-free (asynchronous, up to $n-1$ crashes) shared memory system. This is relatively simple to model as abstraction layers in a simulation stack, as we already specify ADTs solely by their interface (and now you know why I have drilled this specification format into you for years!). The interface for the communication system is now the ADT interface of the data type we assume is provided by the system. We just need to implement the functions of the desired ADT, using calls to the functions provided by the communication system.

The big complication is that function invocations and responses are now separate events. We impose a user condition that a particular user process may not invoke an operation while it has an instance *pending*– it has invoked an operation but not yet received a matching response. An implementation must conversely satisfy a *liveness* condition: it must send (exactly one) response to every invocation. An operation instance is an invocation-response pair of events.

1.1 Consistency Conditions

With separate invocations and responses from different processes, we need a more robust way to discuss the legality of an ADT implementation's behavior. Recall that an ADT specifies legal sequences of operation instances. But in a distributed system with instances at different processes which may overlap in real time, there is not a single sequence for a given execution we can check for membership in the legal set to see if the implementation is behaving correctly. We fix this gap with *Consistency Conditions*.

A consistency condition tells us how to map the concurrent behavior of a particular data structure to the sequential world of ADT specifications. We can then take any execution of the structure, convert appropriately by the consistency condition, and check the resulting sequence(s) against the ADT. If every possible execution maps to a legal sequence, then the implementation is correct. If some execution does not map to a legal sequence, then the implementation is incorrect.

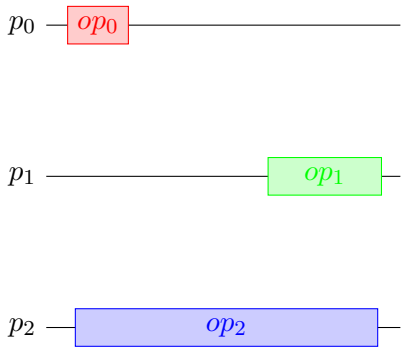
There are **many** consistency conditions in the literature. We will focus on two for the moment.

Definition 1. An execution E of an ADT implementation is *linearizable* if there exists a permutation P of all operation instances in E such that

1. Each object's behavior must be legal: For each shared object O, $P|_O$ is a legal sequence on O.
2. The permutation must respect real-time order: If the response of operation instance q occurs before the invocation of operation instance r, then q appears in p before r.

An algorithm A is linearizable if every admissible execution of A is linearizable.

Consider the following execution which contains three operation instances, indicated by rectangles (invocation is the left end, response is the right end):



Exercise: Draw (or list) all possible permutations of these three operation instances. Mark which ones respect real-time order of non-overlapping instances.

Exercise: Suppose that this is an execution of a $Read/Write$ register implementation, and that $op_2 = Write(0)$, $op_0 = Write(1)$, and op_1 is a $Read$. What value(s) should op_1 return?

Note that the correct value for op_1 to return depends on the linearization. Now, while this should help us understand how linearizability restricts the possible behaviors of an ADT implementation, we are actually using it backwards. In reality, the algorithm will determine what op_1 returns, and then we will be checking whether there is a *linearization* (permutation respecting real time order) that is legal.

Exercise: Is there an illegal value for op_1 to return in this execution?

In this particular case, there is a legal linearization for $op_1 = Read(-, 0)$ and for $op_1 = Read(-, 1)$, so the algorithm could return either and still be correct. We may need to look at other executions to try to find cases where an algorithm is incorrect. Of course, if op_1 returned a value besides 0 or 1, that would be incorrect.

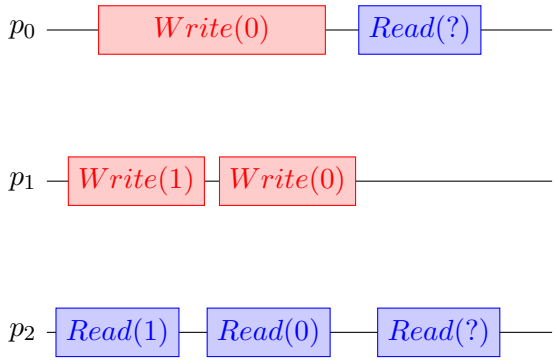
Proving that an algorithm is correct requires proving that for *every possible execution*, there is a legal linearization. This can be complicated.

Definition 2. An execution E of an ADT implementation is *sequentially consistent* if there exists a permutation P of all operation instances in E such that

1. P is legal.
2. For each $i \in [0, n-1]$, $P|_i$ is the order of invocations at process p_i. $P|_i$ is the restriction of P to instances at process p_i.

An algorithm A is sequentially consistent if every admissible execution of A is sequentially consistent.

Exercise: What must the last *Read* instances at each process return in the below execution to satisfy sequential consistency?



Exercise: Give an execution which is sequentially consistent but not linearizable.

We will be focused on linearizable implementations for the moment.

2 Registers

We will start with the most basic version of a register and build stronger and more advanced versions by adding consecutive abstraction layers. Our starting point is a single-bit, Single-Writer Single-Reader (SWSR), *Read*/*Write* register. The eventual goal will be to create a multi-valued, Multi-Writer Multi-Reader (MWMR), *Read*/*Write* register. You can think of this as an `int` object shared by all processes in the system. We will not go all the way in one step, instead adding one feature at a time until we have everything we want.

2.1 Binary to Multi-Valued

Idea 1: Binary representation. We can represent values from the larger domain of possible values in binary and use a collection of single-bit registers to store that value. *Write* would need to be able to update all the individual bits, *Read* would need to read all of them.

- **Exercise:** What happens when a *Write* and a *Read* overlap?
  - If the *Write* has updated some, but not all, the bits, *Read* could return a value which was never stored!
  - For example, if the old value was 1111 and we are writing 0000, a *Read* might return 1100.
- Need *Write* to appear to be a single step, so *Read* cannot happen in the middle of a *Write*.
- A locking or mutual exclusion mechanism would work, but that is hard to implement. General mutual exclusion would also be counter-productive, since we want multiple *Read* instances to be able to run concurrently. But we would also need fairness to ensure that an endless chain of *Read* instances does not prevent any *Write* from completing.
- If we only care that *Read* instances not overlapping any *Write* instance behave correctly (a *safe register*), this algorithm is viable. I have not yet found results indicating whether this approach can provide *regularity* (every *Read* returns the value of a previous or overlapping *Write*) or *atomicity/-linearizability*.

Idea 2: Unary representation. If we want to store v distinct values (say, 0 to $v-1$), we use v bits. If the register holds value x, bit x is set to 1.

- $Write(x)$ sets bit x to 1.
- $Read()$ scans across the array to find the first index containing a 1.

> **Exercise:** What is wrong with this implementation? Describe a specific execution that will not behave as expected.

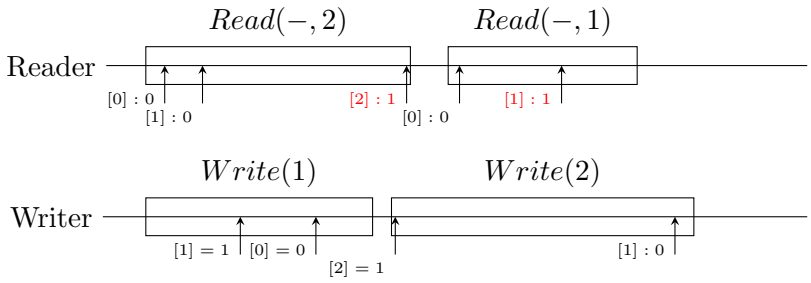
- $Write$ needs to clear the old bit. If it doesn't, you can never increase the stored value.

> **Exercise:** Should $Write$ clear the old bit before or after setting the new bit?

- Before: A $Read$ could never find a bit set to 1. Looping until a bit is set is not wait-free.
- After: Same problem with increasing $Write$s not being visible. Even worse, consecutive $Read$s by the same process can see out-of-order values.

> **Exercise:** Try to construct an execution where this happens.

Consider the following execution fragment. $v = 4$, and the register holds 3 at the beginning of the figure.



We can resolve this issue with a *one-sided clear*: observe that we do not need to clear higher indices, as $Read$ stops when it first sees a 1. $Read$s could still be returning stale values, though, as a $Write$ could have set a lower bit, which the $Read$ already passed. Thus, $Read$ can search until it finds a 1, then search back down to see if a smaller value has been written since, and return the smallest value it sees.

Complexity:

- $Write$ takes at most v steps, waits for no one.
- $Read$ takes at most $2v$ steps, waits for no one.

Correctness: We need to show that there is a linearization of the $Read$ and $Write$ instances in an arbitrary execution which is legal by the specification of a register (each $Read$ returns the argument of the latest preceding $Write$). We will actually do this by building a legal order P, then showing that it respects real time order.

- Observe that since there is a single writer, the order of $Write$s is well-defined, so place them all in P in invocation order.

Algorithm 1 Pseudocode for simulating a multi-valued register with binary registers.

```
    Binary registers data[0..v − 1]: Initially data[0] = 1, data[1..k] = 0
 1: function WRITE(x)
 2:     data[x] = 1
 3:     for i = x − 1 downto 0 do
 4:         data[i] = 0
 5:     end for
 6:     Trigger return for Write
 7: end function
 8: function READ
 9:     i = 0
10:     while data[i] == 0 do i++
11:     end while
12:     highOne = v = i
13:     for i = highOne downto 0 do
14:         if data[i] == 1: v = i
15:     end for
16:     Trigger return for Read with value v
17: end function
```

- Similarly for *Read*s, there is a well-defined order. For each *Read* instance r, in order, place r in P immediately before the $Write$ instance following that whose argument r returns, or at the end of P if none such exists. (This is awkward wording, but the order is important!)
- It should be evident that P is legal, since each *Read* was placed after the $Write$ whose value it returned, and possibly some other *Read* instances.
- We now have four cases to prove that P is a valid linearization. If operation instance op_1 precedes instance op_2 in real time, we must show that op_1 appears in P before op_2.

  Exercise: Prove this is true if both op_1 and op_2 are $Write$ instances and if op_1 is a *Read* instance and op_2 is a $Write$ instance.

- The other two cases are more involved, so refer to the textbook.

2.2 Single-Reader to Multi-Reader

Exercise: Try to think about how to implement a multi-reader register from single-reader registers. What is the primary challenge?

We have to maintain separate banks of SWSR registers for each reader, and we have to update them "simultaneously", so that one reader cannot get a stale value after the other reader has seen a new value the writer has not finished writing to the first reader's bank. Note that, we don't actually need to reason about "banks" of SWSR registers, since we can build on our previous abstraction and use multi-valued SWSR registers. These may (or may not) consist of multiple smaller registers, but that is hidden in the abstraction, so we can consider just one SWSR register for each reader.

Idea 1 One SWSR register for each reader.

- $Write$ writes to all SWSR registers.
- *Read* reads the single SWSR register for that reader.

- This is wait-free, as each operation depends on no one else.
- This algorithm will not satisfy linearizability.
- We can actually prove that there is no way to satisfy linearizability without a mechanism for communication between the readers.

Theorem 1. *In any wait-free simulation of a SWMR register from SWSR registers, at least one reading process must perform a low-level (SWSR) Write.*

Proof. Assume not. Consider readers r_1, r_2.

- r_1, r_2 read from disjoint sets of SWSR registers, since each register has a single reader.
- A *Write* instance must perform SWSR writes on some subset of these registers in some order (recall that only the SWMR writer can write, by our assumption).
- Find the points where r_1 and r_2 first "detect" the *Write* instance. That is, the point where if we paused the *Write* and let r_1 and r_2 continue, they would return the new value. Call these points in the code for *Write* U_1 and U_2.
- $U_1 \neq U_2$, since each step only writes a single register, so the step that makes r_1 detect the *Write* is invisible to r_2 and vice versa. WLOG, U_1 is before U_2.
- Build an execution where the writer runs to U_1, then r_1 runs, seeing the new value, then r_2 runs, seeing the old value. This is not linearizable, since non-overlapping *Read* instances must be linearized out of order.

This contradicts the assumption that readers never write. □

Idea 2: Readers must use more SWSR registers to communicate among themselves to order *Read* instances.

Exercise: How might they do this? What information do we need them to communicate?

- The writer tags each *Write* instance with a sequence number.
- Readers announce the timestamp of the last *Write* instance they read.

See pseudocode in Algorithm 2

Exercise:

- Show that this algorithm is wait-free. What is its complexity (count low-level operation instances)?
- Build a permutation of all instances in an arbitrary execution that is legal.
- Argue that your permutation respects the real-time order of non-overlapping instances.

As before, place *Write* instances in permutation π in invocation order. Note that this is timestamp (sequence number) order. Next, consider each *Read* instance in real-time return order. Add each to π immediately before the *Write* instances following that whose argument the *Read* instances returned.

Algorithm 2 Pseudocode implementing a SWMR register from SWSR registers.

p_w is the writing process. $p_1, \ldots, p_n$ are reading processes.
Shared variables:
$val[1..n]$: Each $val[i]$ is a SWSR register written by p_w, read by p_i.
$report[1..n, 1..n]$: Each $report[i, j]$ is a SWSR register written by p_i, read by p_j used to announce p_i's most recent *Read*.
Local variables for readers: $v[0..n], s[0..n]$: values and sequence numbers. $v[0], s[0]$ hold info from writer, $v[j], s[j]$ hold info from p_j.

1: **function** READ ▷ code for invocation at p_i
2: $v[0], s[0] = val[i]$
3: **for** $i = 1$ to n **do**
4: $v[j], s[j] = report[j, i]$
5: **end for**
6: $m = argmax_j\{s[j]\}$ ▷ index with largest sequence number
7: **for** $j = 1$ to n **do**
8: $report[i, j] = (v[m], s[m])$
9: **end for**
10: Trigger *Read* response with value $v[m]$
11: **end function**
12: **function** WRITE(v) ▷ at single writer p_w
13: $seq++$
14: **for** $i = 1$ to n **do**
15: $val[i] = (v, seq)$
16: **end for**
17: Trigger $Write$ response
18: **end function**

Lemma 1. *Let op_1 and op_2 be two high-level Read or $Write$ instances s.t. op_1 returns before, in real time, op_2's invocation. Then op_1 is before op_2 in the permutation of all instances.*

Proof. $Write$ options are in real-time order, by construction. Consider the possible cases for op_1 and op_2:

- op_1 is a *Read*, op_2 is a $Write$: Assume in contradiction that we linearized op_2 before op_1. then op_1 reads from op_2 or a later $Write$, which implies that op_1 reads from a $Write$ instance which was invoked after op_1 returned, an impossibility.
- op_1 is a $Write$, op_2 is a *Read*: op_2 must low-level read the argument of op_1 or a later $Write$, by the linearizability of the low-level registers. Since we always return the largest-timestamped written value, op_2 will return the argument of op_1 or a later $Write$, so it will linearize after op_1.
- op_1 and op_2 are both *Read*: By linearizability of the low-level registers, op_2 will receive op_1's report and return a value with a timestamp at least as large. Thus, op_2 will linearize after op_1, since we place it in π later, before the same or a later $Write$ instance.

□

2.3 Single-Writer to Multi-Writer

The big difference here is that we will need to order $Write$ instances, since they do not all come from the same process. The readers-must-write result should not cause us any trouble, as the lower-level SWMR implementation handles that.

Exercise: How might we order $Write$ instances? Can you extend that to an idea for the whole simulation?

Idea: Use vector clocks to timestamp $Write$ instances and get a total order.

- Note that we will (yet again) modify our vector clocks. Causal order is partial, we need total. Use lexicographic order on timestamps. That is, compare place-by-place and order according to the first difference.
- When we did vector clocks before, we were in message passing. In shared memory, to communicate our clock vector, we write it in a register. SWMR registers are perfect for this, since no one else should write a given process' local clock.
- Denote writers as $p_0, \ldots, p_{m-1}$. All processes are $p_0, \ldots, p_{n-1}$.
- Each writer $(p_0, \ldots, p_m)$ will have a Val register, with p_i writing $Val[i]$. All processes can read all Val registers. Holds register value and timestamp of $Write$.

Algorithm 3 Pseudocode for simulating multi-writer registers. Code for p_i, readers do not need the $Write$ function.

```
1: function READ
2:     for j = 0 to m − 1 do
3:         (v[j], ts[j]) = Val[j].Read()
4:     end for
5:     newest = argmax_j(ts[j])
6:     Trigger Read response with value v[newest]
7: end function
8: function WRITE(x)
9:     ts = getNewTS()
10:    Val[i].Write(x, ts)
11:    Generate Write response
12: end function
13: function GETNEWTS                                   ▷ Helper function
14:    for j = 0 to m − 1 do
15:        newTS[j] = Val[j][1][j]    ▷ Read what p_j wrote, get the timestamp from the pair, read p_j's component.
16:    end for
17:    newTS[i] = newTS[i] + 1
18:    return newTS
19: end function
```

Exercise: Why does `getNewTS` not need to read other processes' opinions of each others' clocks? (Recall that in message passing, we would update our view of, e.g., p_2's component based on a message from p_1.)

Complexity

Exercise: Argue that this code is wait-free and analyze the complexity of each function.

- $O(m)$ low-level operation instances for each function, never waiting for another process.

Correctness Consider an arbitrary execution of the algorithm. We prove several lemmas:

Lemma 2. *Each process publishes timestamps in increasing order.*

Proof Idea: `getNewTS` updates the writing process' component to more than the last call's, and no other process ever decreases its own component, so by linearizability of the low-level registers, each component of the new timestamp is at least as large as that of a previous timestamp, and the calling process' component is strictly larger.

- It is critical that our lower-level SWMR registers do not have new-old inversions.

Construction 1. Define the permutation π of all operation instances in an execution by ordering $Write$ instances lexicographically by timestamp. Then consider all *Read* instances in increasing real-time order of response. Add each to π immediately before the $Write$ instance following that whose timestamp the *Read* instance returned. [1]

Lemma 3. *π respects the real-time order of non-overlapping instances.*

Exercise: Prove this lemma. Split into groups to consider the three cases:

- *Read* then $Write$
- $Write$ then *Read*
- *Read* then *Read*

Why is $Write$ then $Write$ not worthy of separate consideration?

Lemma 4. *π is a legal sequence by the specification of a register.*

Follows by construction.

Theorem 2. *This algorithm is a correct, wait-free implementation of a MWMR register from SWMR registers.*

Note that this is a very typical proof structure for a distributed ADT implementation, whether in shared memory or in message passing. We start with an arbitrary execution, and need to prove that each invocation has a matching response (part of wait-free here), then construct a total order of all instances, prove that it is a valid linearization, and prove that it is legal by the ADT specification. We typically design our algorithm and order together so that either the proof that the permutation respects real-time order of non-overlapping instances or the proof of legality (as we did here) is trivial.

[1] *Read* instances do not return timestamps, but they return the argument of a particular $Write$, so we can abuse terminology to refer to the timestamp of that $Write$ instance.