Building Better Registers (cont'd)

The following diagram shows wait-free implementations that we cover in this course. (See March 6 lecture for second one, March 11 lecture for first one, Exercise 7 for fourth one. We'll do the third one today.) This shows that we can build multi-writer snapshot objects from swsr registers, so although multi-writer snapshot objects may seem much more powerful than swsr registers, they are actually equivalent in terms of their power to solve problems. (However, the complexity of solving problems with the two types of objects might be quite different.)

Single Writer Single Reader Register $O(n)$ time

↓

Single Writer Multi–Reader Register $O(n^2)$ time

↓

Single Writer Snapshots $O(n)$ time

↓

Multi–Writer Multi–Reader Register

↓

Multi–Writer Snapshots

Reference: [IL93], [AW98] Section 10.2.2.

Construction of *single-writer multi-reader* register from *single-writer single-reader* registers.

We showed last day how to construct a swmr register from swsr registers. In that implementation, processes who were READing the swmr register had to write to some swsr registers[1]. It seems a little odd that processes who only wish to READ the value of the swmr register should have to write to shared memory. We now show that this is in fact necessary.

Lemma In any *wait-free* implementation of a *single-writer multi-reader* register from *single-writer single-reader* registers, at least one READer of the *single-writer multi-reader* register must write to shared memory.

Proof Assume $\exists$ implementation of *single-writer multi-reader* registers from *single-writer single-reader* registers where READer processes $(Q_1, \cdots, Q_{n-1})$ never write to shared memory.

[1] I'm using READ and WRITE to refer to ops on the swmr register, and read and write to refer to ops on the swsr registers

Intuition, we must ensure the consistency among READs if they happen half way through a write operation, and we won't be able to do this if the readers cannot communicate with one another.

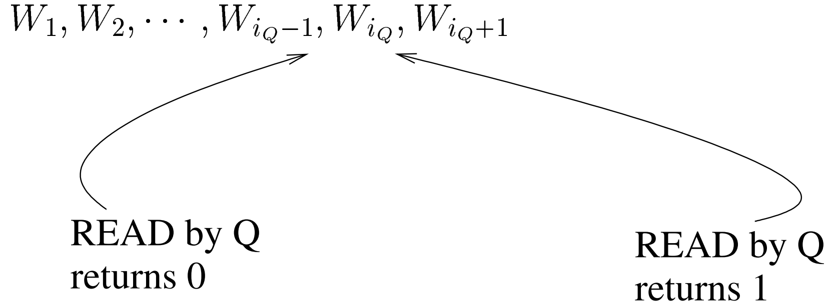
Consider the execution of WRITE(1) on the *single-writer multi-reader* register that is initially 0. The operation will do a bunch of operations on the *single-writer multi-reader* registers.

Let $w_1, w_2, w_3, \cdots, w_k$ be the sequence of write operations it does on the *single-writer single-reader* registers. Imagine inserting a READ by process R just after w_i. (Notice that this doesn't affect the WRITE operation, since the READ leaves no evidence of its existence in shared memory. The READ must return 0 or 1. If the READ is inserted just after w_k, it must return 1, because the WRITE has been completed.

Let i_R be the first i, such that inserting R's READ after w_i causes the READ to return 1. I.e. $i_R = min\{i : \text{READ by } R \text{ after } w_i \text{ returns } i\}$

Let i_Q be same quantity for process Q instead of R, without loss of generality, $i_Q \leq i_R$. (If this is not the case, just exchange Q and R in the following argument.)

Consider w_{i_Q}. Who can read the register that w_{i_Q} writes to?

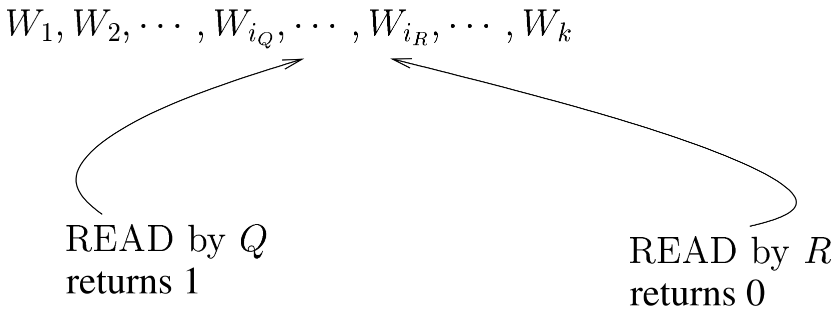


Q has to be able to tell whether w_{i_Q} has happened. Therefore, w_{i_Q} is a write operation to a register that (only) Q can read.

Can we have $i_Q = i_R$?

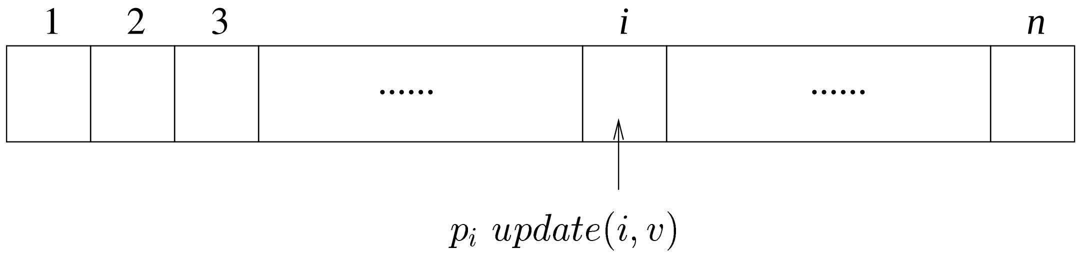
No. Since w_{i_Q} writes to a register readable by Q, and w_{i_R} writes to a register readable by R. These two registers are different, so $i_R \neq i_Q$. Since $i_Q \leq i_R$, we have $i_Q < i_R$.

Consider an execution where we insert, just after w_{i_Q} a READ by process Q and then a READ by process R. By definition of i_Q and i_R, Q's READ must return 1 and R's READ must return 0.



So WRITE linearized before Q's READ and hence before R's READ.
So R should return 1 $\rightarrow$ contradiction.

Construction of ***multi-writer multi-reader*** **register from** ***single-writer*** **snapshot.**



Implementing a counter (where the INC operation returns only ACK) from *single-writer* snapshot is trivial (cf. Feb 6 lecture).
Multi-writer multi-reader register can be done similarly. Every process will write its values for the mwmr register into its component of the snapshot object. To tead the mwmr register, a process will take a snapshot of the object and return the "most recently" written value. To determine which of the values is "most recent", we attach tags to the values that behave a little bit like the timestamps used in the implementation of snapshots.

Attach tag to each value we write into snapshot object.

1	2	3		n
val	*val*	*val*	......	*val*
tag	*tag*	*tag*	......	*tag*

Implementation of *Multi-writer multi-reader* register from a single-writer snapshot

WRITE(v):

SCAN snapshot
let t be 1 + max tag value seen in scan
UPDATE(i,(v,t))

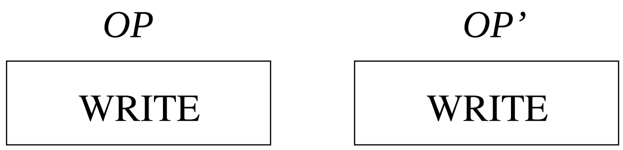
READ of *multi-writer multi-reader* register:
SCAN snapshot object
return the value that has the largest tag
(**if** two SCAN have same tag
then use rightmost value with that tag)

Clearly wait-free (there are no loops).

Check that this is linearizable.
Linearize the WRITEs in order by *tag* values (break ties according to process ids).
Linearize each READ just after the WRITE that wrote the value that the READ is returning (break ties between READs according to the READs' finish times).

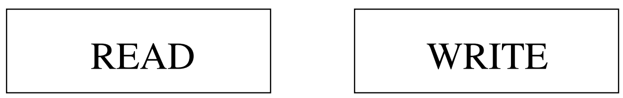
Clearly operations return same results as in linearization. We must just show that if OP completes before OP' begins, OP precedes OP' in the linearization.

Case I Both of the operations are WRITEs



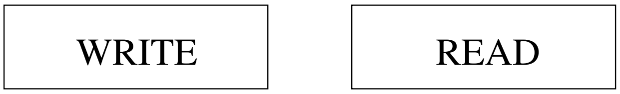
OP' will see tag $\geq$ as big as tag of OP' in component of process who did OP.
Therefore, tag of $OP' >$ tag of OP.
Thus, OP' comes later in linearization.

Case II OP is a READ and OP' is a WRITE



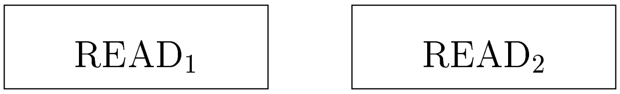
Again tag of the WRITE is bigger than tag of value returned by READ, so the WRITE will come later in the linearization.

Case III A WRITE comes before a READ



If READ's tag ¿ WRITE's tag, then the READ will definitely come later in the linearization. If they are equal, READ returns value written by rightmost process with that tag, and is therefore linearized after all WRITEs with that tag.

Case IV Both of the operations are READs



If tag of OP' is greater than tag of OP, then OP' is clearly linearized later.
If tags are equal:

1. If values returned by the two READs were written by same process, then OP precedes OP' in the linearization because of the way we broke ties between READs.
2. Else, value returned by OP' is to the right of value returned by OP. So OP precedes OP' in linearization since we broke ties between WRITEs with same tag according to process ID.

Randomization

Reference: [AW98], section 14.1.

Recall leader election in ring with anonymous processes (i.e. no process ids, all processes running the same code) is impossible, even in synchronous system with no failures. (See Jan 21 lecture)

Key idea in impossibility proof: You cannot break symmetry, so if one process declares itself leader, so will everyone else. Randomization can help to break symmetry.

- Idea: Everyone picks random id, and run standard algorithm to elect process with maximum id.
- Assuming we know the ring size, detect whether there is unique leader with that id. If the algorithm didn't work (because there was not a unique process with the max id), start over again.

To be continued next day.