Snapshot Object (single writer)

Reference: [AADGMS93] or [Lyn96], Sections 13.3.1, 13.3.2.

Idea

If we read all the components while no update occurs, then we have an accurate view of the entire snapshot object's state. So we could repeatedly read all the components until we see that no update happened during one set of reads.

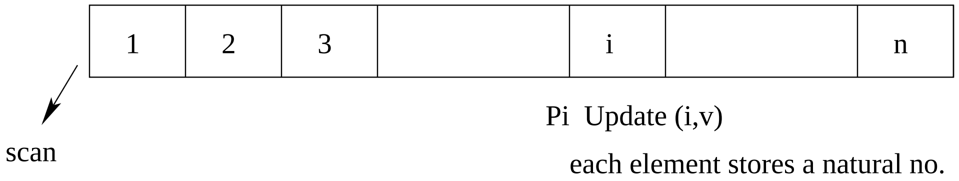

Figure 1: n registers

To check whether updates have occured, we attach a time-stamp to every component. Not only the value but also the time-stamp are stored in every components. Increment the time-stamp every time you do an update.

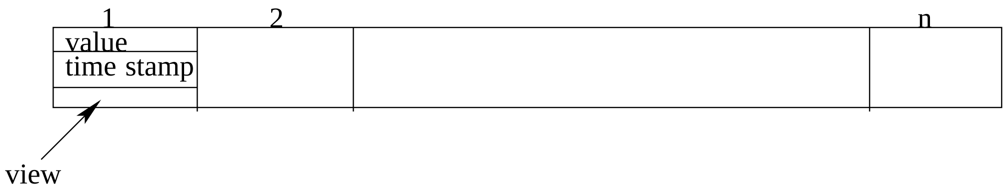

Figure 2: time-stampe

- To update (v,i): Write into register i (v,incremented time-stamp)
- To scan: Repeately read all the register $A[1]...A[n]$ until two sets of reads return exactly same results for all the registers.

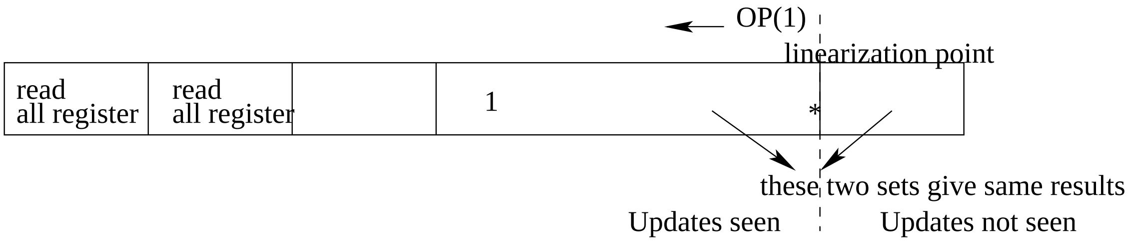

Figure 3:

We are going to linearize the updates at the moment the write occurs.

If we linearize the scan at the moment between the two sets of reads that return the same results, then scan result includes any update that was linearized before that point (since last read must see that update). Also, any update linearized after that point is not part of result retruned by scan (since the second last set of reads could not have seen it). So result of scan is actual value of snapshot at the moment the scan is linearized.

- If some operations on snapshot objects are being simulated, eventually one will complete (i.e. alg. is non-blocking). Someone makes progress.

However, it is not wait-free: a scan can run forever without returning a value.

To make this wait-free, get processes to help one another (just as we did in converting the universal construction from a non-blocking algorithm to a wait-free one). Get any "fast-running" updates to help a "slow-running" scan. Get updates to do scan themselves and tell results to scanners.

Update(v,i)

> execute the scan alg.;
> write into $A[i]$: (v, incremented time-stamp, result of its scan (called a view);
> return ack.;

Scan

> repeatedly read $A[1]...A[n]$;
> until you see 2 sets of reads that are the same (return result of last set of reads)
> OR if one component has been updated $\geq$ two times (i.e. we've seen 3 different
> time-stamps in the same register during scan), return scan in
> the 3rd version of that components register.

See Figure 5. Termination: after $2n+1$ sets of reads (attempts) (n is the number of processes), scan will terminate. So total time to simulate a snapshot operation is $O(n^2)$.

Show alg. is linearizable

Start with any execution. Define a linearization point for every operation. For updates, it is when the write happens. Go through the scan (snap) operation in the order that they finish. If scan saw same results in 2 sets of reads, linearize the scan right between those 2 sets (just as in non-blocking version). Otherwise, scan returns view produced by the scan$'$ embedded in some update operation. Scan$'$ has already been assigned a linearization point. Use that as the linearization point of the scan also.

By induction, easy to prove that each scan (including the ones embedded in update ops) returns the state of the snapshot as its linearization point:

Claim The scan OP with the earliest completion times return the true state of the snapshot object at their linearization points (use induction on i).

one big register n registers

attempt 1:
1st version

0	0	236	0	0
0	***0***	1	0	0
...	...	...		

attempt 2:
2nd version

562	631	236	0	0
2	***1***	1	0	0
...	...	...		

attempt 3:
3rd verion

562	631	762	0	0
2	***1***	3	0	0
...	...	...		

attempt 4:
4th version

562	56	762	0	0
2	***2***	3	0	0
...	...	...		

Figure 4: 4 changes/local register

Show linearization point of every OP is within the interval of time between the OP's invocation and its completion

This is trivial, except for the case where a scan returns the result of a scan′ embedded in some update operation. We must show that the linearization point of the scan′ is within the time interval of the scan.

Let up_2 and up_3 be the updates that write the 2nd and 3rd distinct timestamps seen by the scan in one register. The scan′ operation is embedded in up_3.

The scan′ completes before up_3 performs its write, which is before scan reads the third timestamp. So the linearization point of the scan′ operation (and hence of the scan) is before the end of the scan. (See Fig 6.)

up_3, which contains the embedded scan′ starts after up_2 has completed. But up_2 wrote to a register after the scan read the first distinct timestamp. Therefore, scan′ (and hence scan) is linearized after the beginning of scan. (See Fig 7.)

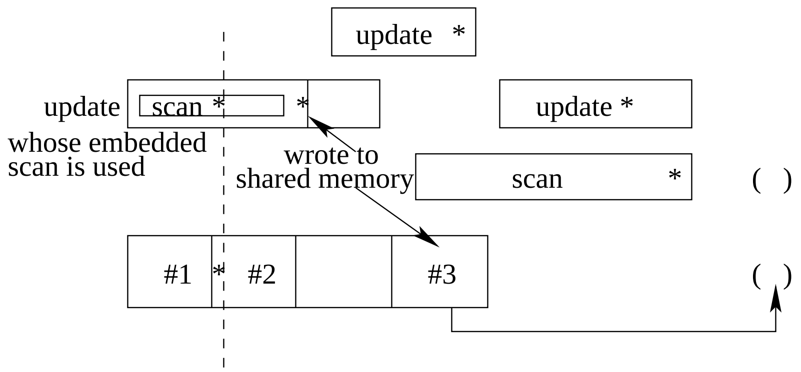

Figure 5:

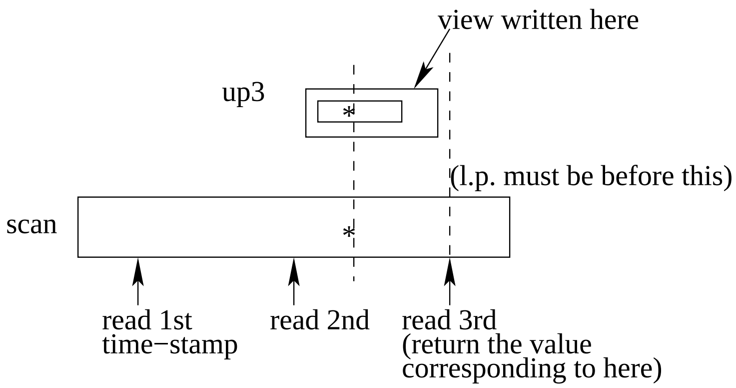

Figure 6:

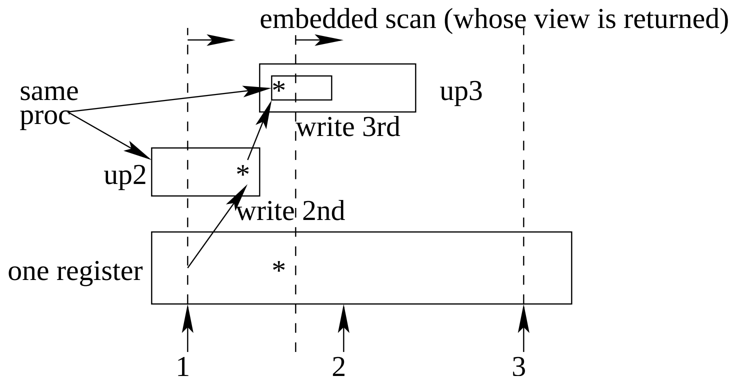

Figure 7: