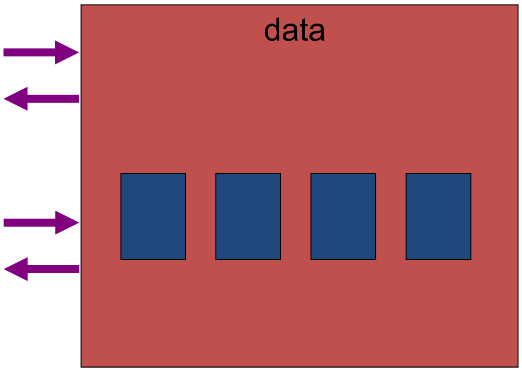
236755
Topic 4: Simulating Shared Objects

Winter 2019-20

Prof. Hagit Attiya

Abstract Data Types (ADT)

- Cover many concurrent applications
- Abstract representation of data & operations for accessing it
  - Signature
  - Specification

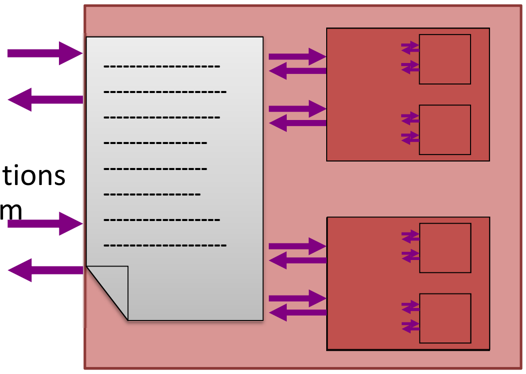


Implementing High-Level ADT

Using lower-level ADTs & procedures

High-level operations translate into primitives on base objects

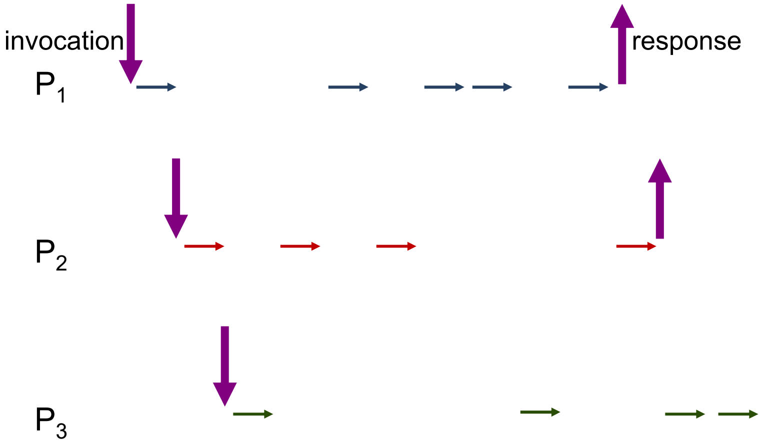
- read, write
- compare&swap
- read-modify-write



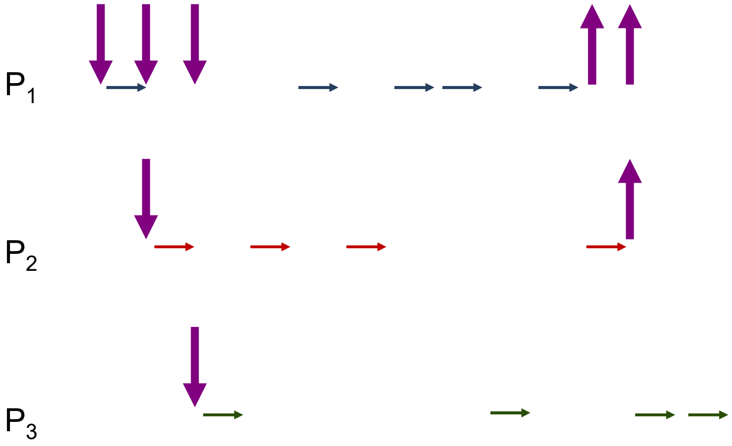
Low-level (primitive) operations are often implemented from more primitive operations

- A hierarchy of implementations

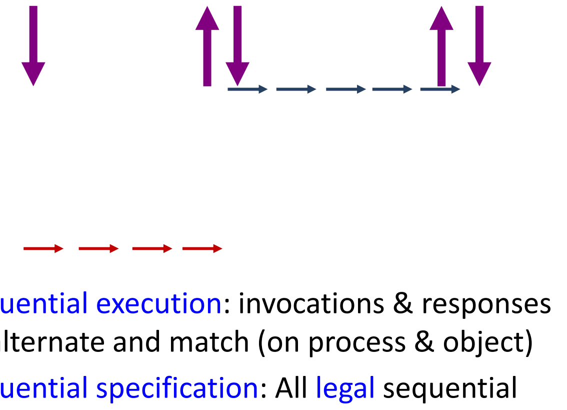
Interleaving Operations' Executions



Interleaving Operations’ Executions



Sequential executions



Sequential execution: invocations & responses alternate and match (on process & object)

Sequential specification: All legal sequential executions, satisfying ADT’s semantics

- E.g., stack: pop returns the last item pushed

Correctness: Linearizability

- For every concurrent execution, there is a sequential execution of the same operations that
  - Is legal (obeys the specification of the ADTs), and
  - Preserves the real-time order of non-overlapping operations
- Equivalently, each operation appears to takes effect instantaneously at some point between its invocation and its response (atomicity)
- When processes fail (there is a partitioning of processes into faulty and nonfaulty), this holds for **all completed operations and a subset of the pending operations**

Some operations never complete

Linearizability is Composable (Local)

- The whole system is linearizable ⇔ each object is linearizable
- Allows to implement and verify objects separately

General Objects

Registers support *read* and *write* operations

Later, we'll see wait-free simulations of one kind of register out of another kind (# values, readers, writers)

What about (wait-free) simulating a significantly different kind of data type out of registers?

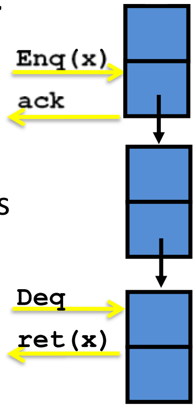
More generally, what about (wait-free) simulating an object of type *X* out of objects of type *Y* ?

Key Insight

- Focus on asynchronous, wait-free simulations
  - Typically, in shared memory

Ability to simulate object of type X using only objects of type Y and registers is related to the ability of those data types to solve consensus (or other problems)

Example: FIFO Queue



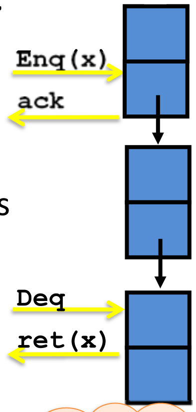
- invocation *enq(x)* and response *ack*
- invocation *deq* and response *ret(x)*

Each *deq* returns oldest enqueued value that is not yet been dequeued (⊥ if queue is empty)

Wait-free two-process algorithm for consensus with an initialized queue

```
prefer[i] = input
if deq(queue) == 0
      then return prefer[i]
else return prefer[1-i]
```

Example: FIFO Queue



- invocation *enq(x)* and response *ack*
- invocation *deq* and response *ret(x)*

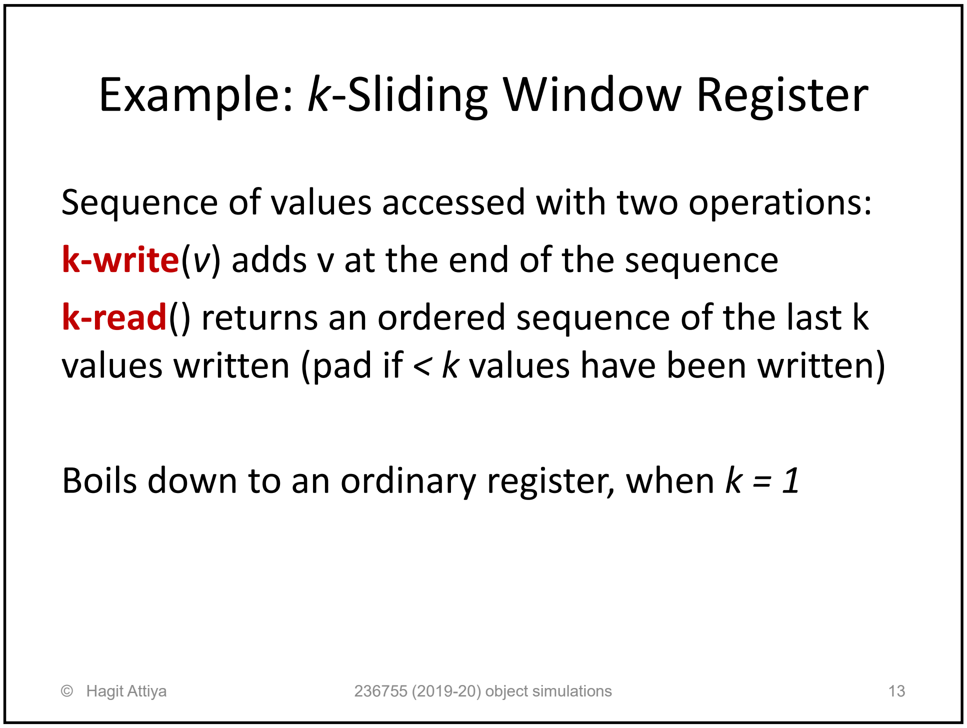
Each *deq* returns oldest enqueued value that is not yet been dequeued (⊥ if queue is empty)

Can we simulate a queue with registers?

No! Otherwise, can solve 2-processes consensus w/ registers:

  - simulate a FIFO queue using registers
  - run consensus algorithm with queues using simulated queue to solve 2-process conse

No 2-processes consensus w/ registers



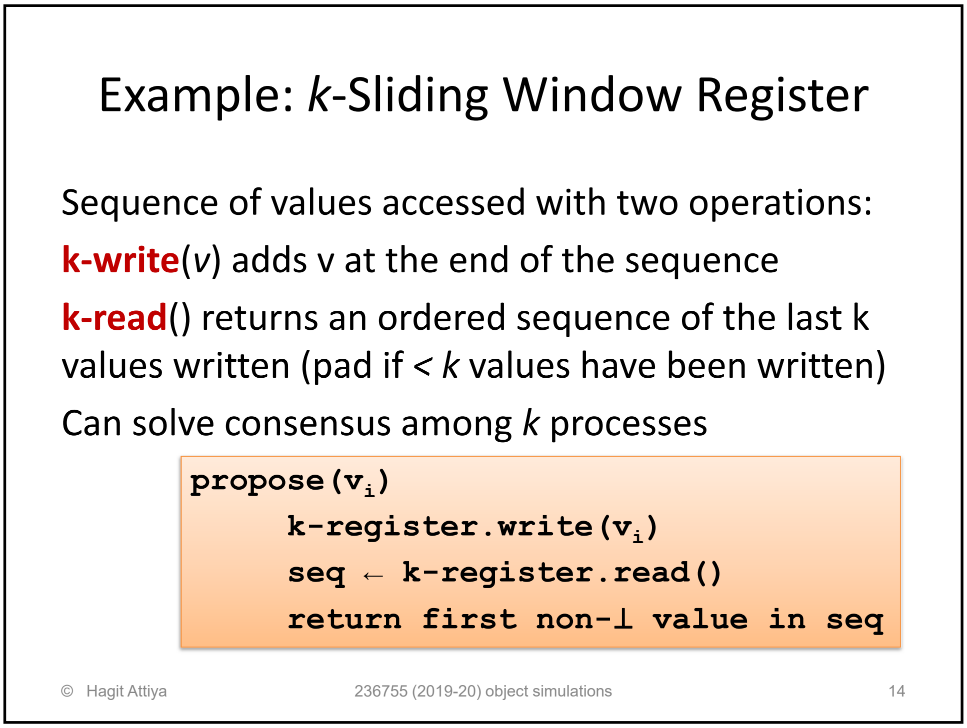
Example: *k*-Sliding Window Register

Sequence of values accessed with two operations:

k-write(*v*) adds v at the end of the sequence

k-read() returns an ordered sequence of the last k values written (pad if < *k* values have been written)

Boils down to an ordinary register, when $k = 1$



Example: *k*-Sliding Window Register

Sequence of values accessed with two operations:

k-write(*v*) adds v at the end of the sequence

k-read() returns an ordered sequence of the last k values written (pad if < *k* values have been written)

Can solve consensus among *k* processes

```
propose($v_i$)
    k-register.write($v_i$)
    seq ← k-register.read()
    return first non-⊥ value in seq
```

Example: *k*-Sliding Window Register

Sequence of values accessed with two operations:

k-write(*v*) adds v at the end of the sequence

k-read() returns an ordered sequence of the last k values written (pad if < *k* values have been written)

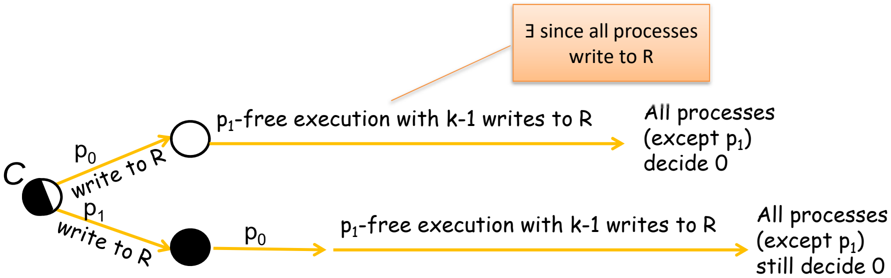
Can solve consensus among *k* processes

Cannot solve consensus among *k+1* processes

Standard bivalence-style proof (similar to queue)

Core Case of the Proof

Critical configuration, where the next steps by all *k+1* processes are writes to the same window register R



Consensus Numbers

Data type *X* has **consensus number** CN(X) = n if *n* is the largest number of processes for which consensus can be solved using only objects of type *X* and read/write registers

Determine if there is a wait-free simulation of Y from X based on their consensus number

data type	**consensus number**
read/write register, snapshots	1
FIFO queue, fetch&Inc	2
k-window register	*k*
compare &swap	∞

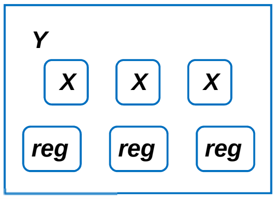
Consensus Numbers

Data type *X* has **consensus number** CN(X) = n if *n* is the largest number of processes for which consensus can be solved using only objects of type *X* and read/write registers

data type	**consensus number**
read/write register, snapshots	1
FIFO queue, fetch&Inc	2
k-window register	*k*

Theorem: If n = CN(*Y*) > CN(X) = m, then there is no wait-free simulation of an object of type *Y* using objects of type *X* and read/write registers for > m processes

Consensus Numbers: Proof



Assume there is a wait-free simulation of *Y* using *X* and registers in a system with *k*, *n* ≥ *k* > m processes

Consensus algorithm for *k* processes

- Since *CN(Y) = n,* there is a *k*-process consensus algorithm using Y and registers
- Execute this algorithm using simulated objects of type *Y* from objects of type *X* (and registers)

Theorem: If *n* = CN(*Y*) > CN(X) = *m*, then there is no wait-free simulation of an object of type *Y* using objects of type *X* and read/write registers for > *m* processes

---

Sample Corollaries

There is no wait-free simulation of any object with consensus number > 1 using read/write registers

There is no wait-free simulation of any object with consensus number > 2 using queues and read/write registers

There is no wait-free simulation of any object with consensus number > k using k-window registers

Universality of Consensus Numbers

Data type is **universal** if objects of that type and read / write registers can wait-free simulate any data type

Theorem: A data type with consensus number *n* is universal for a system with ≤ *n* processes

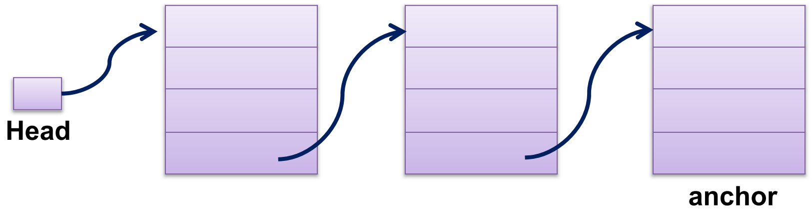
1. A **non-blocking** n-process algorithm to simulate any data type **using compare & swap**
2. Modify to use objects with **consensus number n**
3. Modify to be **wait-free**
4. Bound the shared memory used and handle non-determinism

Universal Simulation Using CAS

Represent object by a linked list with the sequence of operations applied to the simulated object

Apply an operation on the simulated object by inserting an appropriate node at the head of the linked list

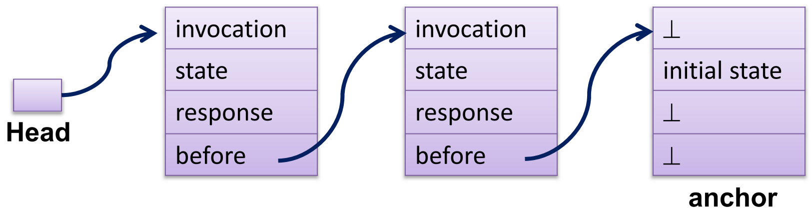
Use compare&swap on the Head pointer of the list



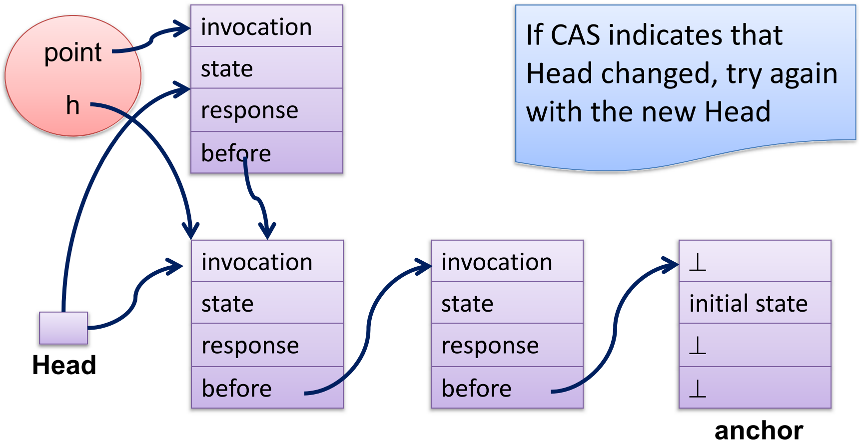
The Linked List

Each linked list node has

- operation invocation (= type and parameters)
- new state of the simulated object
- operation response
- pointer to previous node (= previous op)



Simulation w/ CAS: In Pictures



If CAS indicates that Head changed, try again with the new Head

Simulation w/ CAS: The Code

Initially Head points to anchor node

- represents initial state of simulated object

local variables h, point

When inv is invoked:

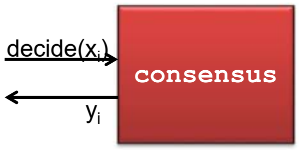
```
allocate a new linked list node in shared memory,
  pointed to by local var point
point.inv = inv
repeat
  h = Head
  point.state, point.response = apply(inv,h.state)
  point.before = h
until compare&swap(Head,h,point) == h
do the output indicated by point.response
```

depends on simulated data type

Head not changed, point it to new node

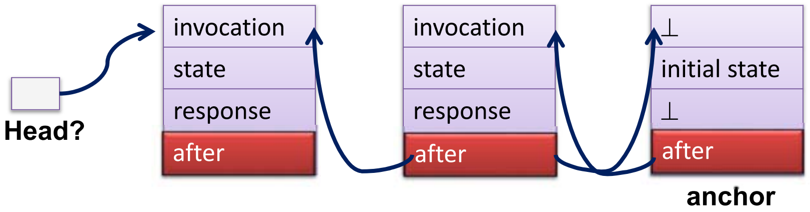
Strengthening the Algorithm 1: Using Arbitrary Consensus Objects

Replace compare&swap object with an arbitrary n-process consensus object



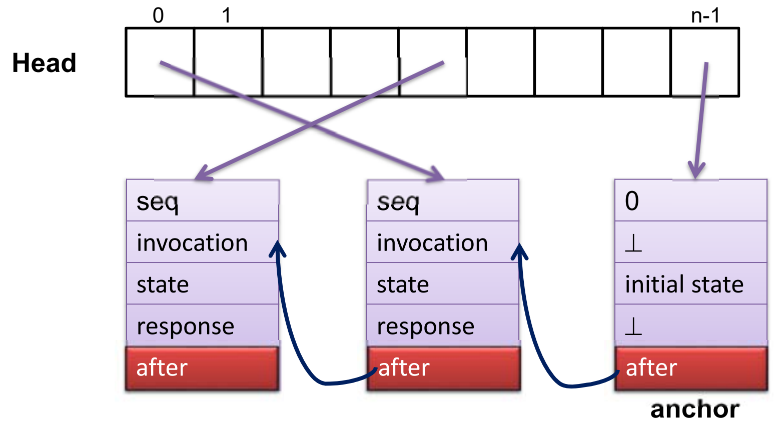
Use it to decide on the next operation

Since a consensus object is one-shot, use many copies, each allowing to thread an additional operation to the list



Strengthening the Algorithm 1: Finding the Head of the List

Per-process Head pointer, to the last node it has inserted
Sequence numbers allow to identify the latest node

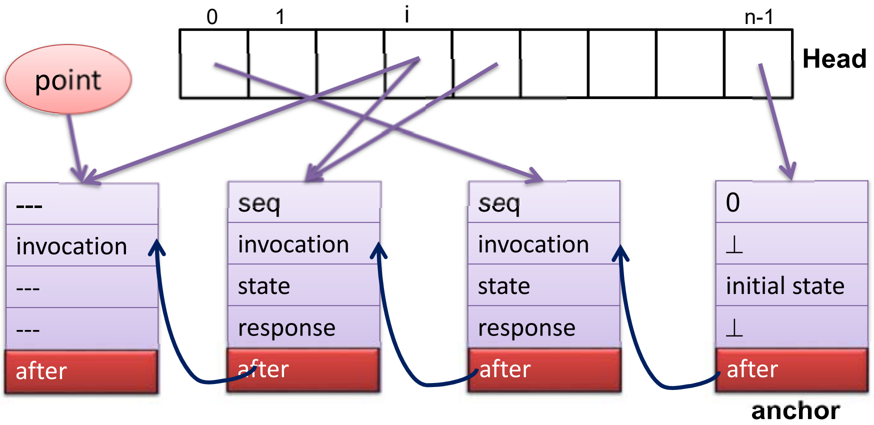


Algorithm with Consensus Objects

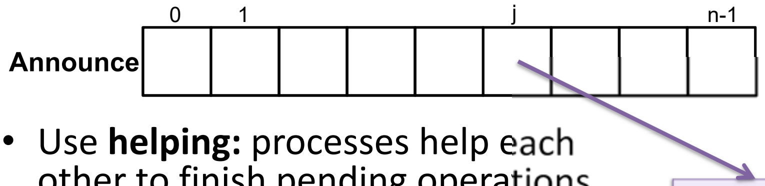
```
Initially all Head entries point to the anchor node

when inv occurs
   point = new opr, point.inv = inv
   for j=0 to n-1     find node with maximum sequence number
       if Head[j].seq > Head[i].seq then Head[i]=Head[j]
   repeat             try to thread your operation
       win = decide(Head[i].after,point)
       win.seq = Head[i].seq+1
       win.state, win.response =
              apply(win.inv, Head[i].state)
       Head[i]= win   point to the following node
   until win = point
   return point.response
```

Non-Blocking Universal Simulation: In Pictures

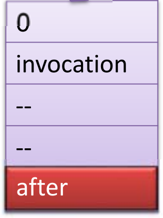
Successful case



Strengthening the Algorithm 2: Making it Wait-Free



- Use **helping:** processes help each other to finish pending operations (not just their own)
- When appending the rth operation, help process j = r mod n (if j has a pending operation)
- **Announce** array stores pending operations



Code for Wait-Free Simulation

```
Initially all Head and Announce entries point to anchor

When inv occurs
   Announce[i] = new opr, Announce[i].inv,seq = inv,0
   for j=0 to n-1
        if Head[j].seq > Head[i].seq then Head[i]=Head[j]
   while Announce[i].seq == 0 do
        priority = Head[i].seq+1 mod n            process with priority
        if Announce[priority].seq == 0 then       help is needed
                point = Announce[priority]        help the other process
        else point = Announce[i]                  perform own operation
        win = decide(Head[i].after, point)        like before
        win.state,reponse = apply(win.inv,Head[i].state)
        win.seq = Head[i].seq+1
        Head[i] = win
   return Announce[i].response
```

Strengthening the Algorithm 3: Bounding the List

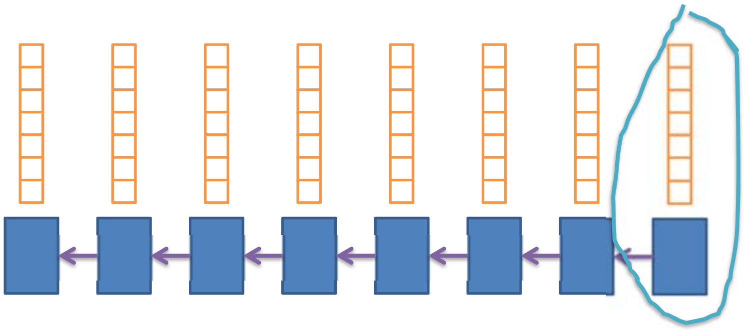
A process allocates nodes from a private pool

A node is recycled when it is not referenced anymore

When can we recycle node #r?

- No process trying to thread node ≥ (r+n+1) will access node r
- When the operations that thread nodes r…r+n terminate, node r can be recycled
- When a process p finishes threading node m it releases nodes m-1…m-n.
- After node r is released by the operations threading nodes r…r+n, it can be recycled

Pictorially



Strengthening the Algorithm 3: Randomized Consensus

- Suppose we relax the liveness condition for linearizable shared memory:
  - operations must terminate with high probability
- Now a randomized consensus algorithm can be used to simulate any data type out of any other data type, including read/write registers
- Need to have a non-deterministic simulation since different processes will have different outcomes

Simulating Read / Write Objects

Can we provide a **shared read / write variable** in an asynchronous message-passing system, when processes can fail?

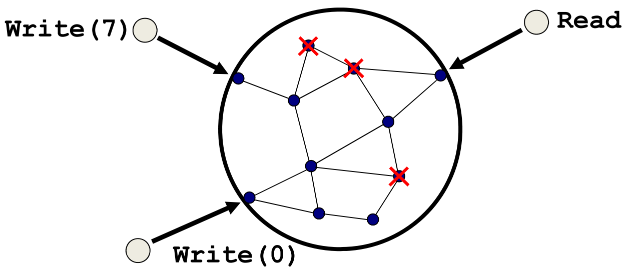
- Yes, if we have enough nonfaulty processes

Can we provide **stronger types** of read / write variables, when processes can fail?

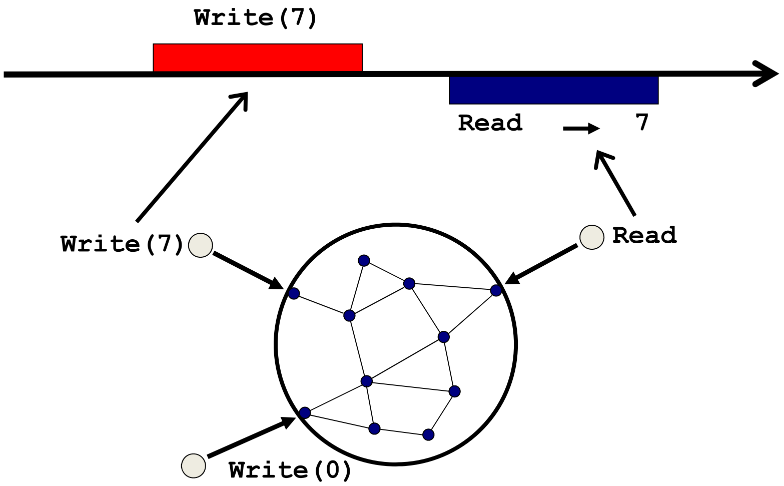
- Yes, as long as we don't read-**and-**write

Simulating Shared Memory

- Provide a single-writer single-reader register (this is the high-level) in a message-passing system
  - Accessed by **`read`** and **`write`** operations
- Underlying system is asynchronous message passing (this is the low-level), where less than half the processes can crash

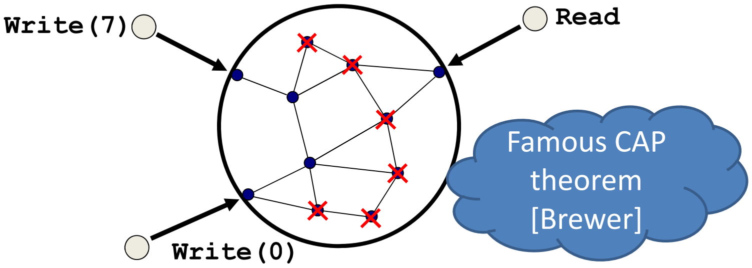


Linearizability



Simulating Shared Memory w/ Failures

- Requires a majority of nonfaulty processes
- Otherwise, the system can be partitioned
  - A read “misses” the latest write



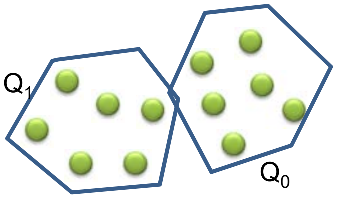
Famous CAP theorem [Brewer]

Must have n > 2f

Theorem: A simulation of a 1-reader, 1-writer read/write linearizable register in an asynchronous message passing tolerates at most f < n/2 crash failures

Proof: Suppose in contradiction there is an algorithm tolerating *f = n/2* crash failures

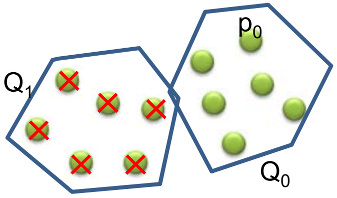
Partition processes into two sets, Q_0 and Q_1, each of size f



Must have n > 2f: Writing

Consider an execution in which

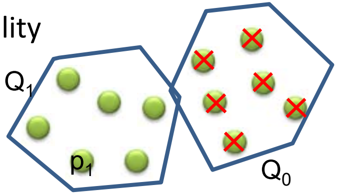
- initial value of simulated register is 0
- all processes in Q_1 crash initially
- process p_0 in Q_0 invokes write(1) at time 0 and no other operations are invoked
- the write completes at some time t_0 without any process in Q_0 receiving a message from any process in Q_1



Must have n > 2f: Reading

Consider another execution in which

- initial value of simulated register is 0
- all processes in Q_0 crash initially
- process p_1 in Q_1 invokes a read at time t_0+1 and no other operations are invoked
- the read completes at some time t_1 without any process in Q_1 receiving a message from any process in Q_0
- the read returns 0, due to linearizability



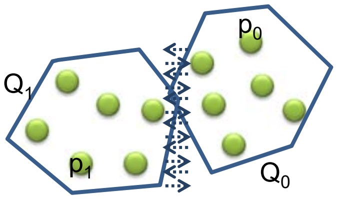
Must have n > 2f: Arithmetic

Now paste the views of processes in Q_0 from the first execution with the views of processes in Q_1 from the second execution

- messages between Q_0 and Q_1 are delayed to arrive after time t_1

This execution is not linearizable, since read(0) follows write(1)

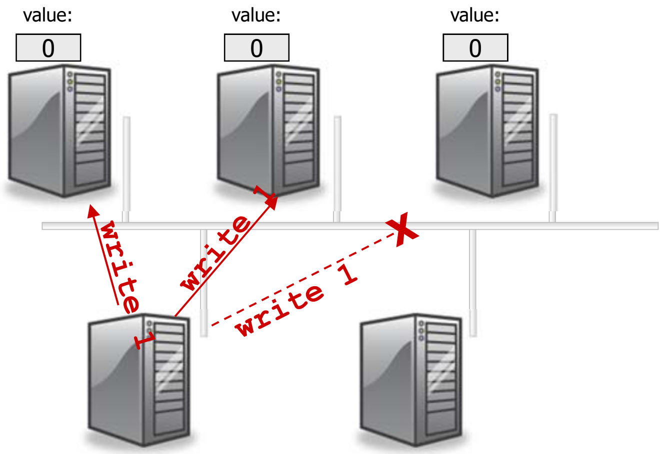
➔ Must assume a majority of nonfaulty processes



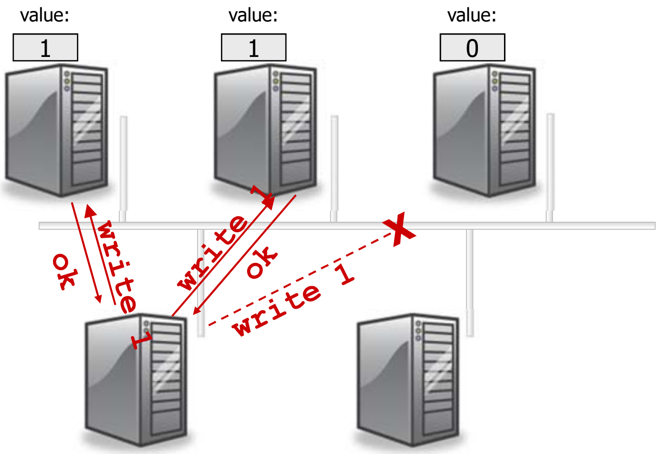
The Algorithm in a Nutshell: Write

- The simulated register is replicated at each process
- Each data item has a unique **sequence number**
  - sequence of values
- `write(d, val, seq#)`
  - generate next sequence number
  - send a message with the value and the sequence number to all processes
  - each recipient updates its replica and sends ack
  - writer waits for n-f > n/2 acks

The Algorithm in Action: Write



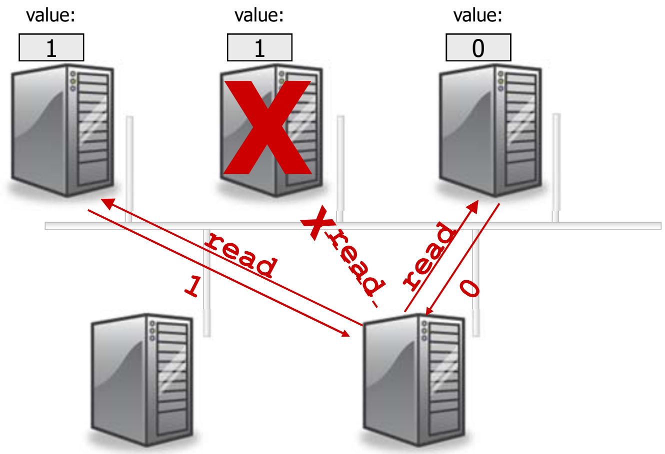
The Algorithm in Action: Write



The Algorithm in a Nutshell: Read

- Each data item has a unique **sequence number**
- `read(d)` returns (val, seq#)
  - send a request to all processes
  - each recipient sends back current value of its replica
  - wait for > *n/2* replies
  - return value associated with largest sequence number
  - do a **write-back** to ensure atomicity of reads

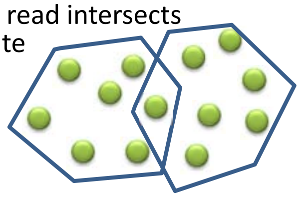
The Algorithm in Action: Read



Key Idea for Correctness

- Each read should return the value of "the most recent" write
- Each read or write communicates with > n/2 processes

➔ The set of processes communicating with a read intersects the set of processes communicating with a write



- Since system is asynchronous, a message on behalf of an operation might be overtaken by a message on behalf of a later operation
  - reader and writer keep track of "status" of each link
  - don't send a message on a link before receiving ack on previous message (**ping-pong**)

Proving Linearizability

Let ts(W) = sequence number of W

Let ts(R) = sequence number of write that R reads from

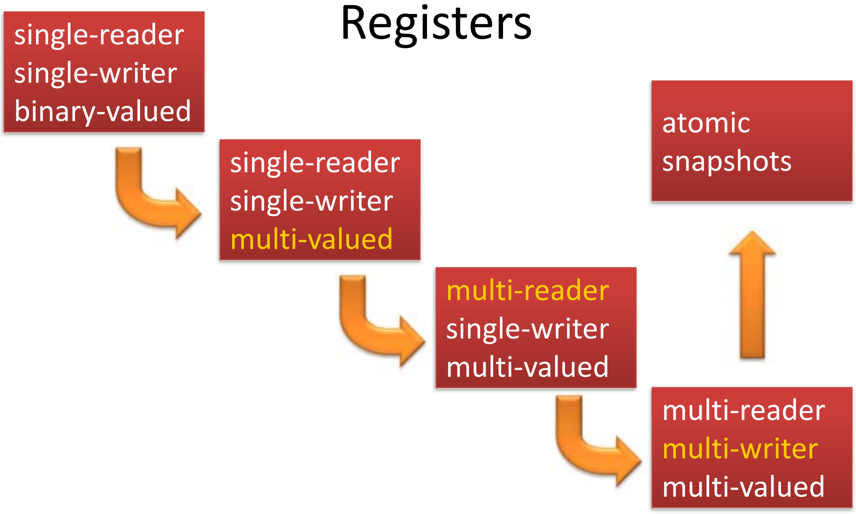
$O_1 \rightarrow O_2$ denotes O_1 completes before O_2 starts

Key lemmas:

• If $W_1 \rightarrow W_2$, then ts(W_1) < ts(W_2)	one writer generates ts
• If $W \rightarrow R$, then ts(W) ≤ ts(R)	majorities intersect
• If $R \rightarrow W$, then ts(R) < ts(W)	can't read from the future
• If $R_1 \rightarrow R_2$, then ts(R_1) ≤ ts(R_2)	majorities intersect

---

Simulating R/W Registers from R/W Registers



single-reader
single-writer
binary-valued

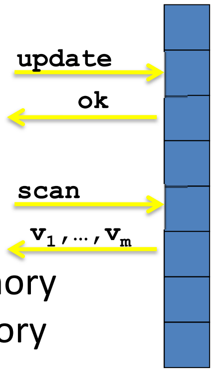
single-reader
single-writer
multi-valued

multi-reader
single-writer
multi-valued

multi-reader
multi-writer
multi-valued

atomic
snapshots

Atomic Snapshots



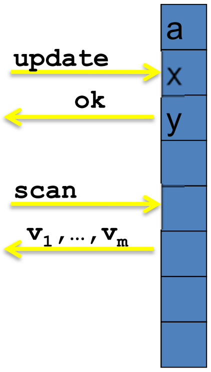
- m components
- Update a single component
- Scan all the components "at once" (atomically)

Instantaneous view of the whole memory

very useful for designing shared-memory algorithms

Has a wait-free implementation from read/write variables

Atomic Snapshots: More Formally



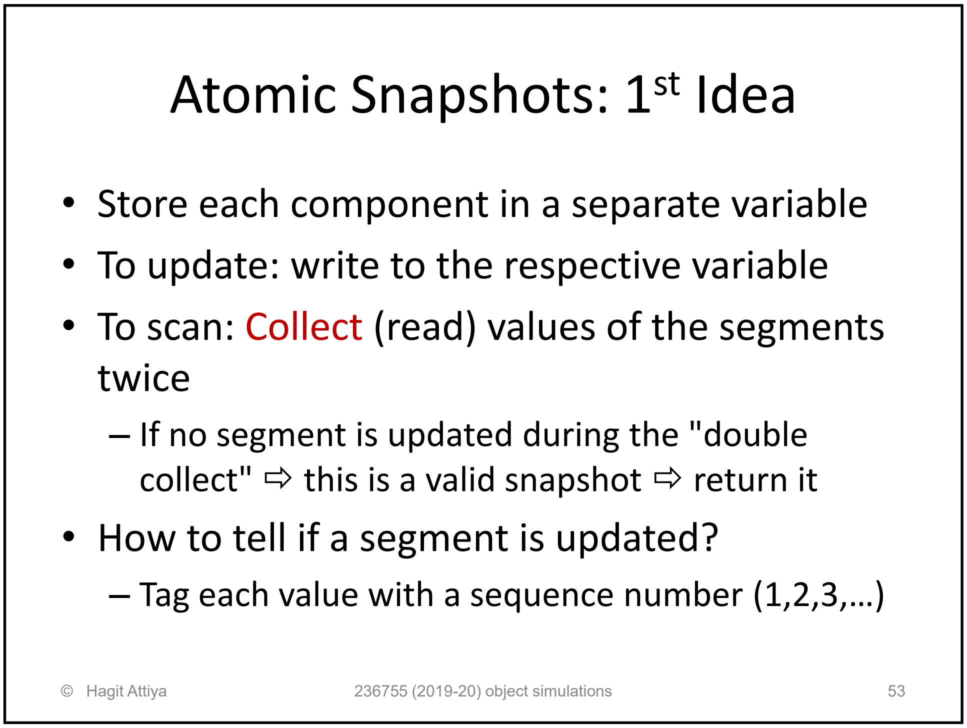
Operations are

- invocation $scan_i$, returns V, where V is an array of n values
- invocation update(i,d) where d is a data value, returns ok

Legal sequences: if scan returns V, then V[k] is the parameter of latest preceding update(k,_)

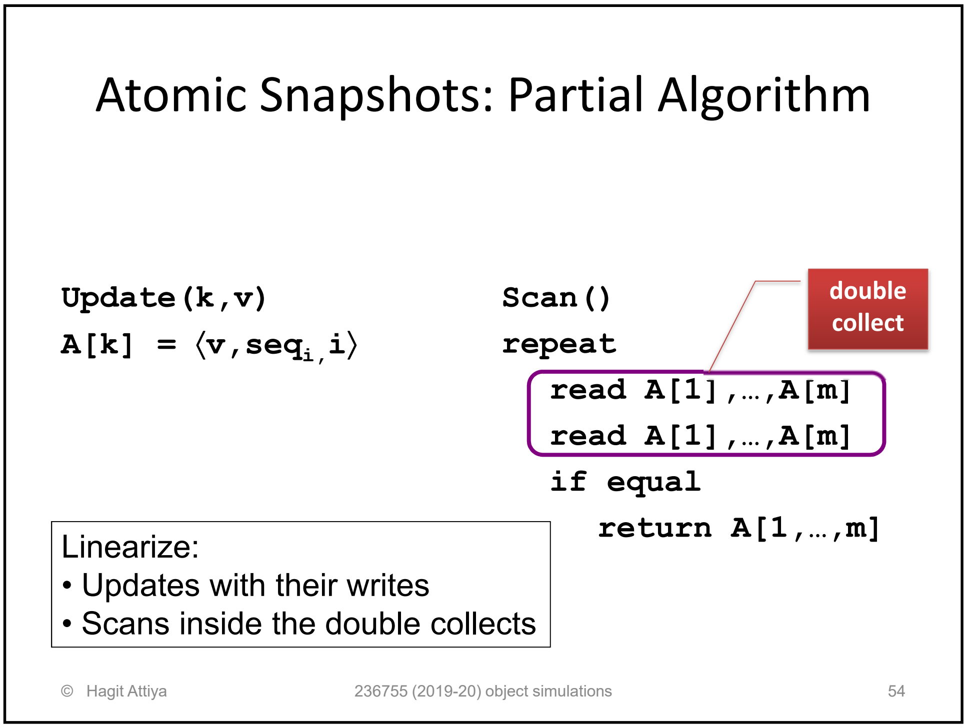
For example:

update(*1*,*x*) update(*2*,*y*) scan([*a*,*x*,*y*]) update(*0*,*z*) scan([*z*,*x*,*y*])



Atomic Snapshots: 1^{st} Idea

- Store each component in a separate variable
- To update: write to the respective variable
- To scan: Collect (read) values of the segments twice
  - If no segment is updated during the "double collect" ⇨ this is a valid snapshot ⇨ return it
- How to tell if a segment is updated?
  - Tag each value with a sequence number (1,2,3,…)



Atomic Snapshots: Partial Algorithm

```
Update(k,v)
A[k] = ⟨v,seq_i,i⟩
```

```
Scan()
repeat
  read A[1],…,A[m]
  read A[1],…,A[m]
  if equal
    return A[1,…,m]
```

double collect

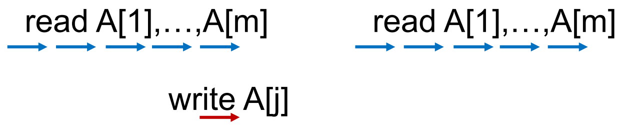
Linearize:
- Updates with their writes
- Scans inside the double collects

Atomic Snapshot: Linearizability

Double collect (read a set of values twice)

If equal, there is no write between the collects

- Assuming each write has a new value (seq#)



Creates a safe zone, where the scan is linearized

Wait-free Atomic Snapshot

Embed a scan within the update & write its view to the segment

Scanner returns view obtained in last collect

```
Update(v,k)
V = scan
A[k] = ⟨v,seq_i,i,V⟩
```

```
Scan()
repeat
    read A[1],…,A[m]
    read A[1],…,A[m]
    if equal
        return A[1,…,m]
    else record diff
    if twice p_j
        return V_j
```

direct scan

borrowed scan

Linearize:
- Updates with their writes
- Direct scans as before
- Borrowed scans with source

Atomic Snapshot: Borrowed Scans

Interference by process p_j

And another one…

⇨ p_j does a scan inbeteween

Linearizing with the borrowed scan is OK.

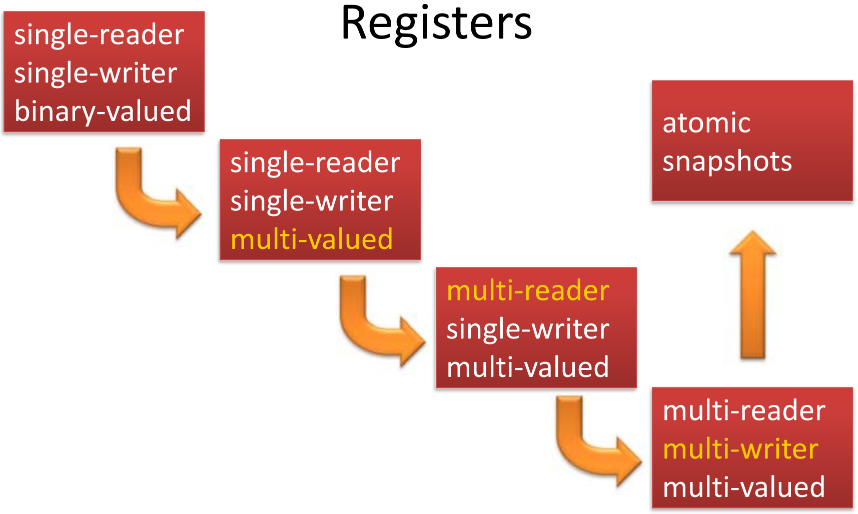
Complexity of Atomic Snapshots

Uses $O(m)$ read/write variables (some are large)

Scan needs $O(n^2)$ reads and writes, why?

Update needs $O(n^2)$ reads and writes

Simulating R/W Registers from R/W Registers



single-reader
single-writer
binary-valued

single-reader
single-writer
multi-valued

multi-reader
single-writer
multi-valued

multi-reader
multi-writer
multi-valued

atomic
snapshots

Multi-Writer from Single-Writer: Key Ideas

- Each writer announces each value it wants to write to all the readers, by writing the value to its own (single-writer multi-reader) register
- Each reader reads all the values written by the writers and returns the latest one
- How to determine latest value?
  - use timestamps (as in Bakery algorithm)
  - since multiple processes generate timestamps, need to coordinate timestamp generation

Multi-Writer from Single-Writer

✓ Wait-free by construction

Create linearization:

– Place writes in timestamp order

– Insert each read before the write following the write it returns

Add logical time to values

```
Write(v,X)
read $TS_1$,...,read $TS_n$
$TS_i$ = max $TS_j$ +1
write ⟨v,$Ts_i$,i⟩ to $R_i$
```

```
Read(X)
read $R_1$,...,read $R_n$
return $v_j$ with
  maximal <$TS_j$,j>
```

Multi-Writer from Single-Writer

✓ Wait-free by construction

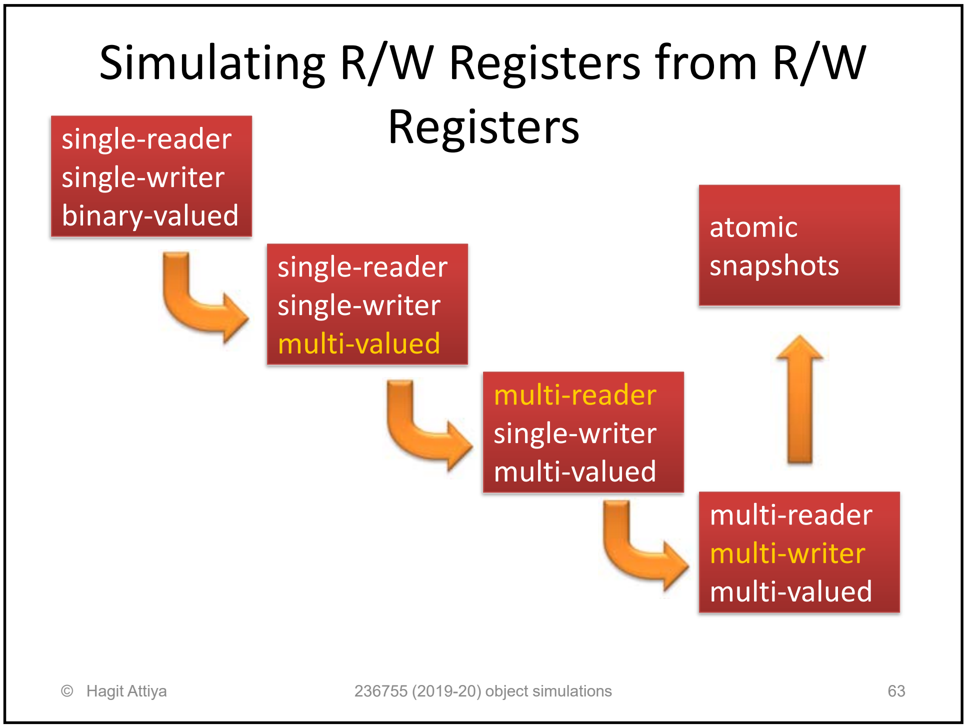
Create linearization:

– Place writes in timestamp order

– Insert each read before the write following the write it returns

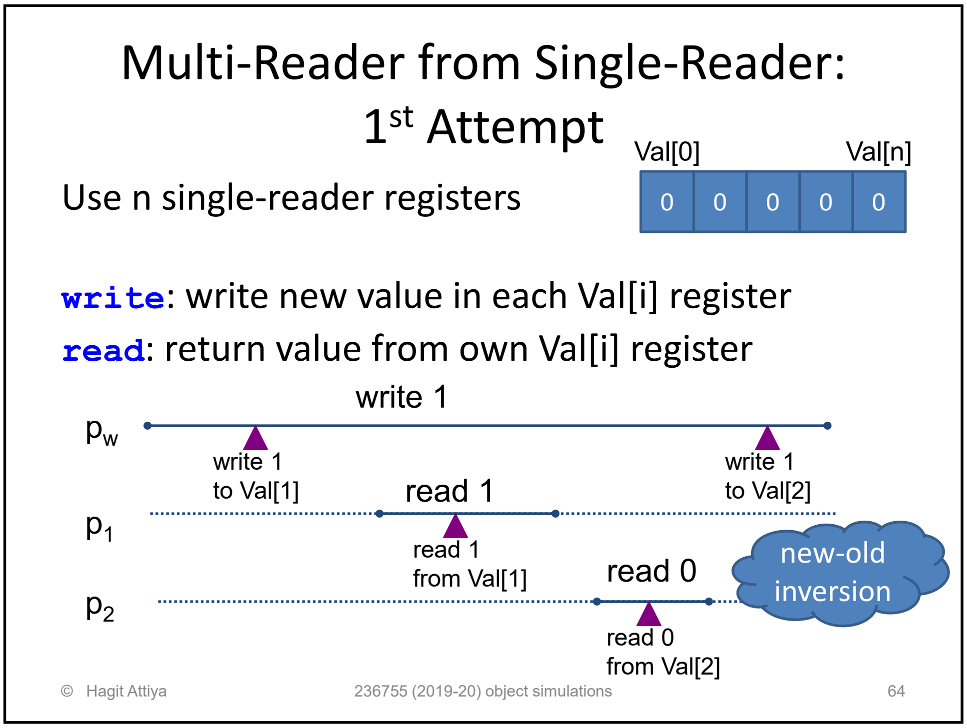
✓ Legality is immediate

✓ Real-time order is preserved since a read returns a value (with timestamp) larger than all preceding operations



Simulating R/W Registers from R/W Registers

© Hagit Attiya 236755 (2019-20) object simulations 63



Multi-Reader from Single-Reader: 1st Attempt

Use n single-reader registers

write: write new value in each Val[i] register

read: return value from own Val[i] register

© Hagit Attiya 236755 (2019-20) object simulations 64

Readers Must Write

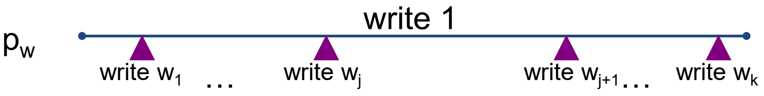
Theorem: In a wait-free simulation of a multi-reader single-writer register from single-reader single-writer registers, at least one reader writes

Proof: Suppose, in contradiction, there is an algorithm in which readers never write

- p_w is the writer, p_1 and p_2 are the readers
- initial value of simulated register is 0
- S_1 are the single-reader registers read by p_1
- S_2 are the single-reader registers read by p_2

Readers Must Write

- Consider execution in which p_w writes 1 to the simulated register, by a sequence of writes, $w_1,\ldots,w_k$, to the single-reader registers
  - Each of them is either in S_1 or in S_2 (but not both)



Readers Must Write

v_j^i denotes the value returned if p_i reads after w_j

For each reader p_i the value of the simulated register "switches" from 0 (old) to 1 (new), at some point

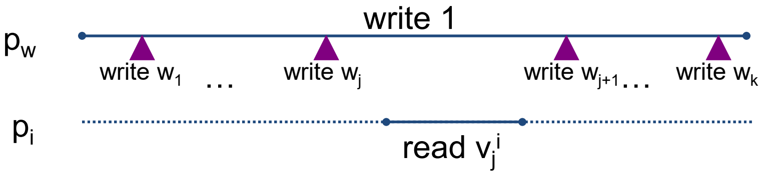
- $v_1^1 = \ldots = v_{a-1}^1 = 0,\ v_a^1 = \ldots = v_k^1 = 1$

⇨ w_a is a write to a register in S_1

- $v_1^2 = v_2^2 = \ldots = v_{b-1}^2 = 0,\ v_b^2 = \ldots = v_k^2 = 1$

⇨ w_b is a write to a register in S_2

a ≠ b

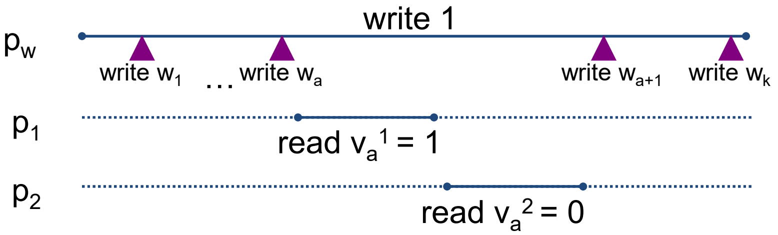


Readers Must Write

Assume a < b

Since readers do not write, they return the same values as when running alone

⇨ new-old inversion, not linearizable



Corrected Multi-Reader Algorithm

0	0	0	0	0

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

In the simulated read,
announce the value to be returned

Check values returned by previous reads

Sequence numbers allow to compare returned values

Writer's Algorithm

0	0	0	0	0

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

- get the next sequence#
  - an integer, incremented by 1 each time
- write (value, sequence#) to *Val[1],…,Val[n]* (one copy for each reader)

Reader p_i's Algorithm

0	0	0	0	0

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

- read (value, sequence#) from Val[i]
- read (value, sequence#) from Report[j,i]
- pick pair with largest sequence#
- write that pair to row i of Report
- return value component of that pair

Correctness of Multi-Reader Algorithm

- Obviously wait-free
  - Write: n primitive writes
  - Read: $n+1$ primitive reads and n primitive writes
- To prove linearizability, show a permutation of the high-level operations that is clearly legal and then prove it preserves real-time order of non-overlapping operations

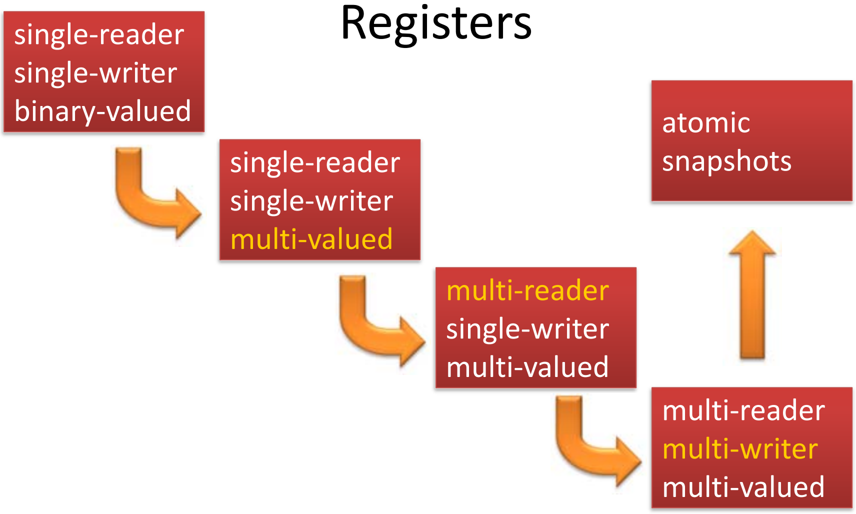
Constructing the Permutation

- Put all writes in the order they occur in the execution
  - Single writer ⇨ writes do not overlap
- Consider the reads in the order of their responses in the execution
  - read *R* **reads from** write *W* if *W* generates the sequence# associated with the value *R* returns
  - place *R* immediately before the write that follows *W*
- By construction, the permutation is legal

Preserving Real-Time Order

- **write-write:** by construction
- **read-write:** *R* precedes *W* in the execution.
  Then *R* cannot read from *W* or any later write.
  ⇨ *R* is placed before *W* in the permutation
- **write-read:** *W* precedes *R* in the execution.
  Then *R* reads *W*'s sequence# or a larger one from *Val[ ]* and reads from *W* or a later write.
  ⇨ *R* is placed after *W* in the permutation
- **read-read:** R_i by p_i precedes R_j by p_j in the execution.
  Then p_j reads R_i's sequence# or a larger one from *Report[i,j]*.
  ⇨ R_j reads from the write that R_i read from or a later one
  ⇨ R_j is placed after R_i in the permutation

Simulating R/W Registers from R/W Registers



single-reader
single-writer
binary-valued

single-reader
single-writer
multi-valued

multi-reader
single-writer
multi-valued

multi-reader
multi-writer
multi-valued

atomic
snapshots

Multi-Valued From Binary

The simulated register takes values {0,…,K-1}

Binary approach: a different binary register stores each bit of the multi-valued register being simulated

- Read algorithm reads all registers and returns the resulting value
- Write algorithm writes the new bits in some order

Errors when the reader overlaps a slow write and sees some new bits and some old bits

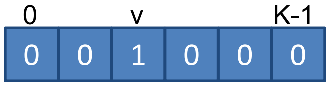
A Unary Approach

Use an array of *K* binary registers, *B[0..K-1]*

- value *v* is represented with *B[v] = 1* and other entries 0

- **Read** algorithm: read *B[0], B[1],…,* until finding the first 1; return the index
- **Write** algorithm: set new entry of B and zero the old entry of *B*

OK if reads and writes don't overlap.

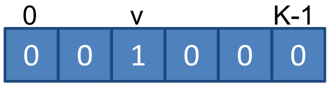


0		v			K-1
0	0	1	0	0	0

When Reads and Writes Overlap…

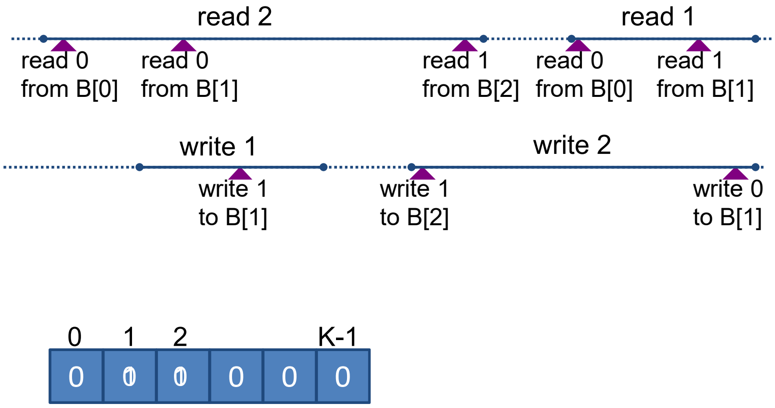
Problem: reader may never find a 1 in *B*

Solution: write algorithm only clears (sets to 0) entries that are smaller than the entry that is set (to 1)



0		v			K-1
0	0	1	0	0	0

New-old inversion



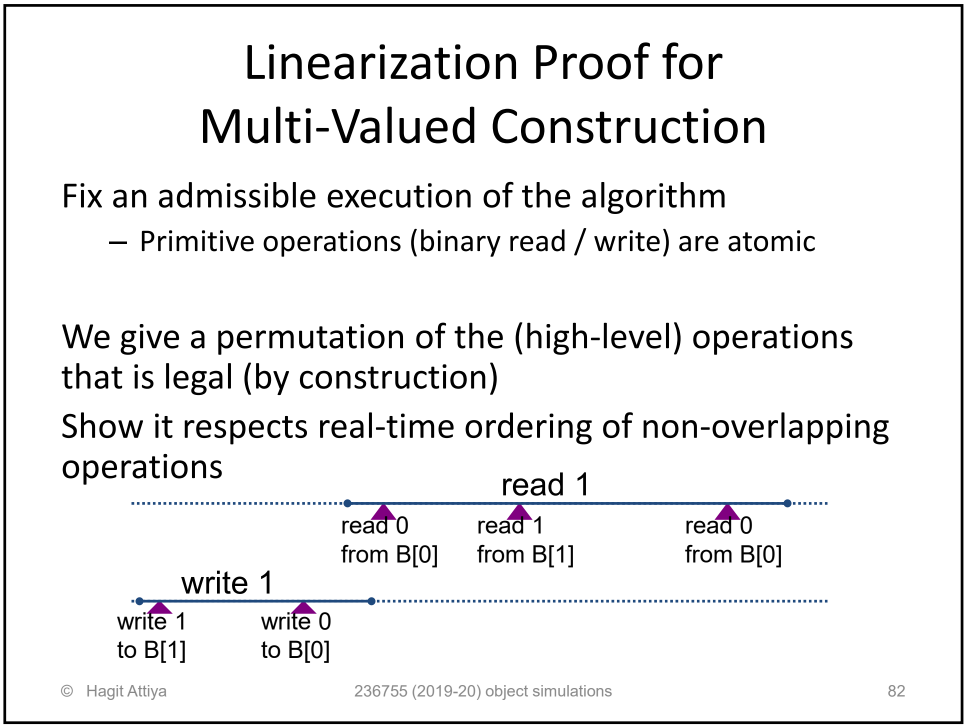
Corrected Algorithm

Read: scans up to first 1, *then read down* to check those entries are still 0; return smallest index set during downward read

Write(r): set r to 1 and then set to 0 entries smaller than r

Clearly, **wait-free**:

- writer does at most *K* (primitive) writes
- reader does at most *2K-1* (primitive) reads



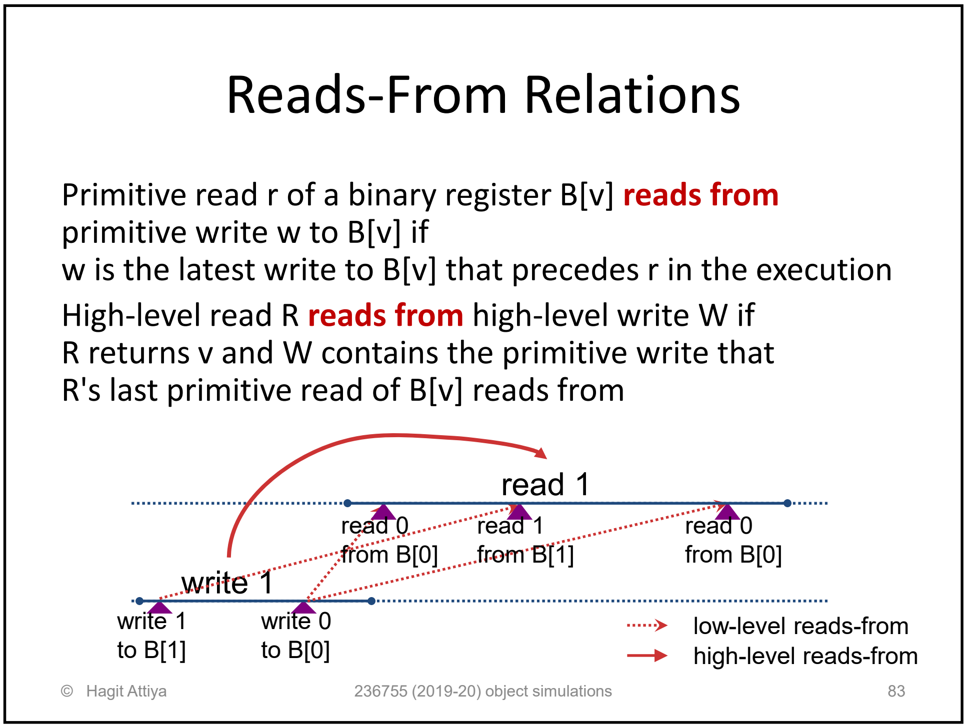
Linearization Proof for Multi-Valued Construction

Fix an admissible execution of the algorithm

- Primitive operations (binary read / write) are atomic

We give a permutation of the (high-level) operations that is legal (by construction)

Show it respects real-time ordering of non-overlapping operations



Reads-From Relations

Primitive read r of a binary register B[v] **reads from** primitive write w to B[v] if w is the latest write to B[v] that precedes r in the execution

High-level read R **reads from** high-level write W if R returns v and W contains the primitive write that R's last primitive read of B[v] reads from

The Permutation

Primitive read r of a binary register B[v] **reads from** primitive write w to B[v] if w is the latest write to B[v] that precedes r in the execution

High-level read R **reads from** high-level write W if R returns v and W contains the primitive write that R's last primitive read of B[v] reads from

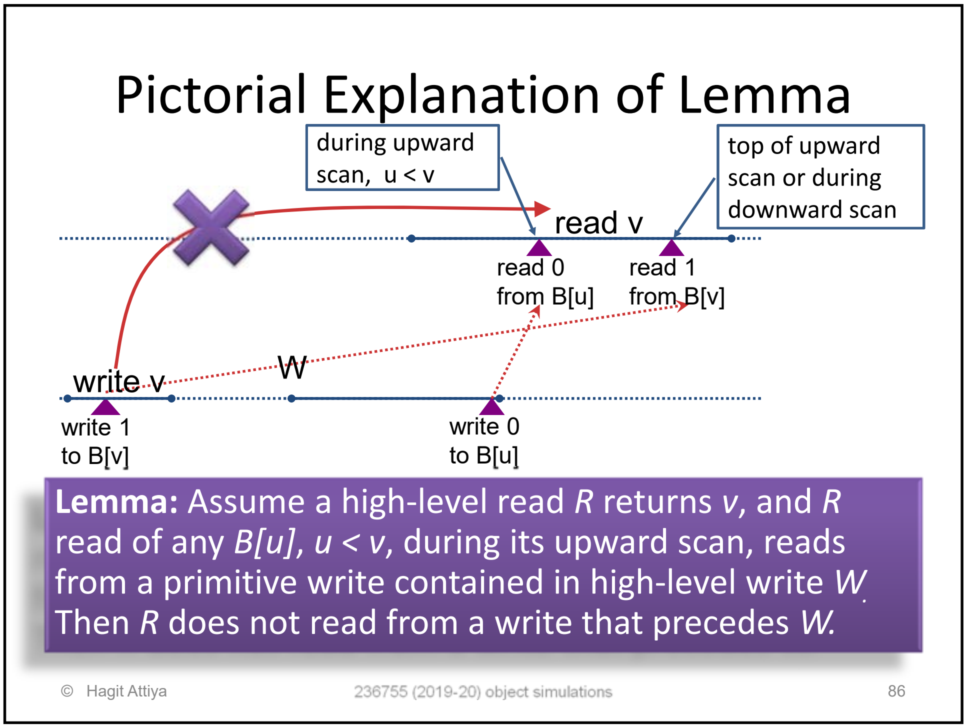
- Place (high-level) writes in the order they occur
  - no concurrent writes
- Consider each (high-level) read R in the order they occur
  - no concurrent reads
- If *R* **reads from** write *W*, place *R* immediately before the write that follows *W* in the permutation

Legal by construction

Permutation Preserves Real-Time Order

- **write-write**: OK, by construction
- **read-write**: OK, since cannot **read from** a later write
- Two cases remain:
  - **write-read**
  - **read-read**

Lemma: Assume a high-level read *R* returns *v*, and *R* read of any *B[u]*, *u* < *v*, during its upward scan, reads from a primitive write contained in high-level write *W*. Then *R* does not read from a write that precedes *W*.



Pictorial Explanation of Lemma

during upward scan, u < v

top of upward scan or during downward scan

read v

read 0 from B[u]

read 1 from B[v]

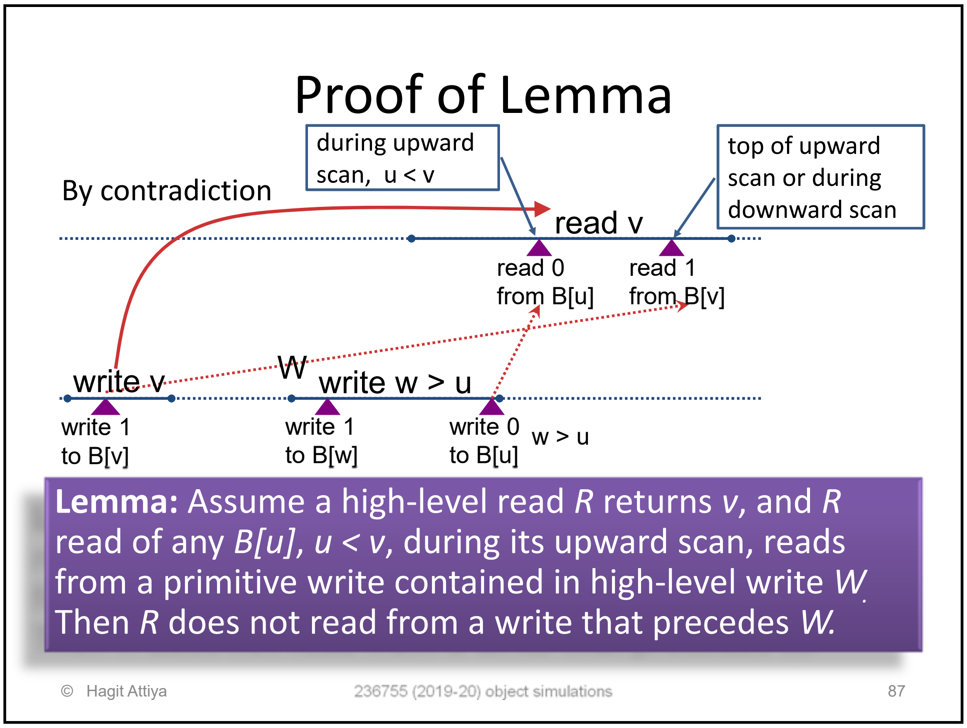
write v

W

write 1 to B[v]

write 0 to B[u]

Lemma: Assume a high-level read *R* returns *v*, and *R* read of any *B[u]*, *u < v*, during its upward scan, reads from a primitive write contained in high-level write *W*. Then *R* does not read from a write that precedes *W.*



Proof of Lemma

By contradiction

during upward scan, u < v

top of upward scan or during downward scan

read v

read 0 from B[u]

read 1 from B[v]

write v

W

write w > u

write 1 to B[v]

write 1 to B[w]

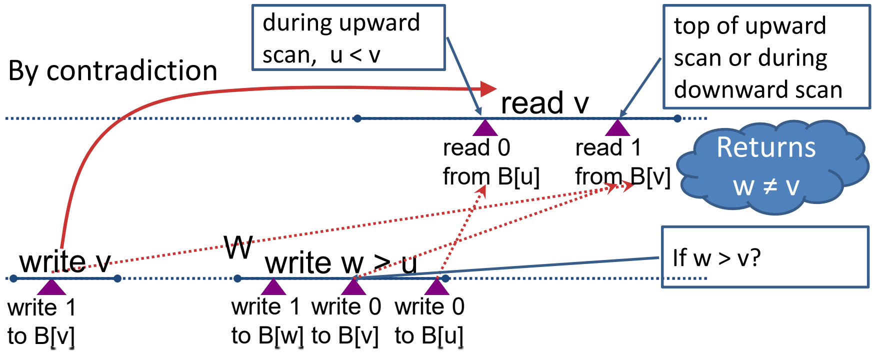
write 0 to B[u]

w > u

Lemma: Assume a high-level read *R* returns *v*, and *R* read of any *B[u]*, *u < v*, during its upward scan, reads from a primitive write contained in high-level write *W*. Then *R* does not read from a write that precedes *W.*

Proof of Lemma

By contradiction



during upward scan, u < v

top of upward scan or during downward scan

read v

read 0 from B[u]

read 1 from B[v]

Returns w ≠ v

W

write v

write w > u

If w > v?

write 1 to B[v]

write 1 to B[w]

write 0 to B[v]

write 0 to B[u]

Lemma: Assume a high-level read *R* returns *v*, and *R* read of any *B[u]*, *u < v*, during its upward scan, reads from a primitive write contained in high-level write *W*. Then *R* does not read from a write that precedes *W.*