

# 6.852: Distributed Algorithms

## Fall, 2015

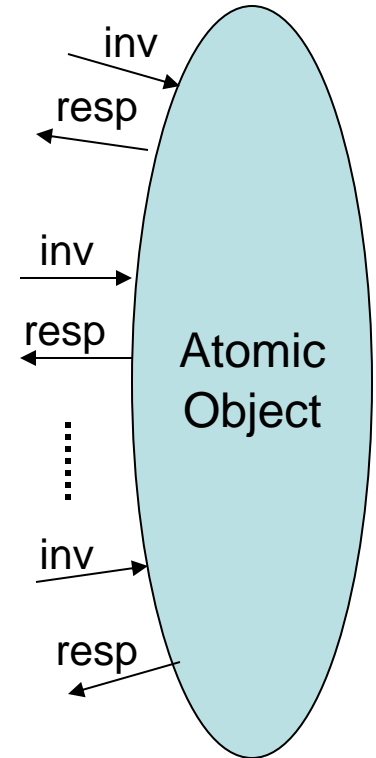
Class 19

# Today's plan

- Atomic objects:
  - Atomic snapshots of shared memory: Snapshot atomic objects.
  - Read/write atomic objects
- Reading: Sections 13.3-13.4
- Next:
  - Wait-free synchronization.
  - Reading:
    - [Herlihy, Wait-free synchronization] Dijkstra prize paper
    - [Attiya, Welch, Chapter 15]

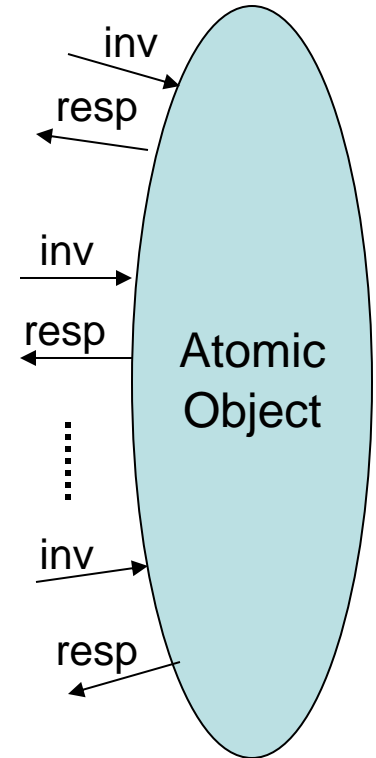
# Last week

- Defined **Atomic Objects**.
- An atomic object of a given type is similar to an ordinary shared variable of that type, but it allows concurrent accesses by different processes.
- Still looks “as if” operations occur one at a time, sequentially, in some order consistent with order of invocations and responses.
- Correctness conditions:
  - Well-formedness, atomicity, failure-free termination
  - Fault-tolerance conditions:
    - Wait-free termination
    - f-failure termination



# Atomic sequences

- Suppose  $\beta$  is any well-formed sequence of invocations and responses. Then  $\beta$  is **atomic** provided that one can
  - Insert serialization points for all complete operations.
  - Select a subset  $\Phi$  of incomplete operations.
  - For each operation in  $\Phi$ , insert a serialization point somewhere after the invocation, and make up a response.
  - In such a way that moving all matched invocations and their responses to the serialization points yields a trace of the variable type.



# Canonical atomic object automaton

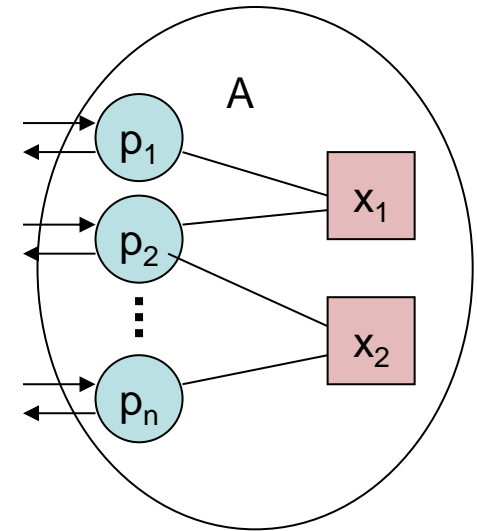
- The **canonical object automaton** keeps an internal copy of the variable, plus delay buffers for invocations and responses.
- 3 kinds of steps:
  - **Invoke**: Invocation arrives, gets put into **in-buffer**.
  - **Perform**: Invoked operation gets performed on the internal copy of the variable, response gets put into **resp-buffer**.
  - **Respond**: Response returned to user.
- **Perform** step corresponds to serialization point.

# Canonical atomic object automaton

- Equivalent to the original specification for a wait-free atomic object, in a precise sense.
- Can be used to prove correctness of algorithms that implement atomic objects, e.g., using simulation relations.
- **Theorem 1:** Every fair trace of the canonical automaton (with well-formed  $U$ ) satisfies the properties that define a wait-free atomic object.
- **Theorem 2:** Every trace allowed by a wait-free atomic object (with well-formed  $U$ ) is a fair trace of the canonical automaton.

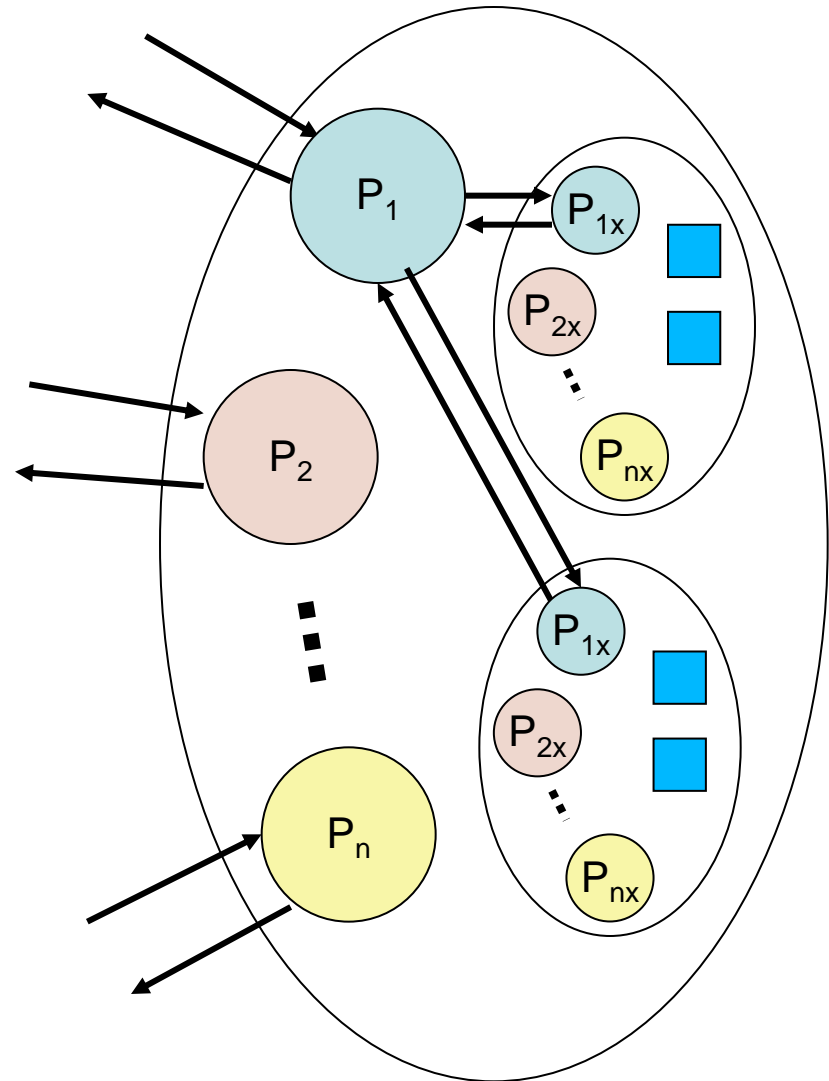
# Atomic objects vs. shared variables

- Can substitute atomic objects for shared variables in a shared-memory system, and the resulting system “behaves the same”.
- **Theorem:** For any execution  $\alpha$  of  $\text{Trans} \times U$ , there is an execution  $\alpha'$  of  $A \times U$  (the original shared-memory system) such that:
  - $\alpha \mid U = \alpha' \mid U$  (looks the same to the users), and
  - $\text{stop}_i$  events occur for the same  $i$  in  $\alpha$  and  $\alpha'$  (same processes fail).
- Needs a technical assumption.
- Construction also preserves liveness:
  - $\alpha$  fair implies  $\alpha'$  fair.
  - Provided that the atomic objects don't introduce new blocking.
    - E.g., wait-free.
    - E.g., at most  $f$  failures for  $A$  and each atomic object guarantees  $f$ -failure termination.



# Can use Trans to justify:

- Implementing fault-tolerant atomic objects using other fault-tolerant atomic objects.
- Building shared-memory systems, including shared-memory implementations of fault-tolerant atomic objects, hierarchically.

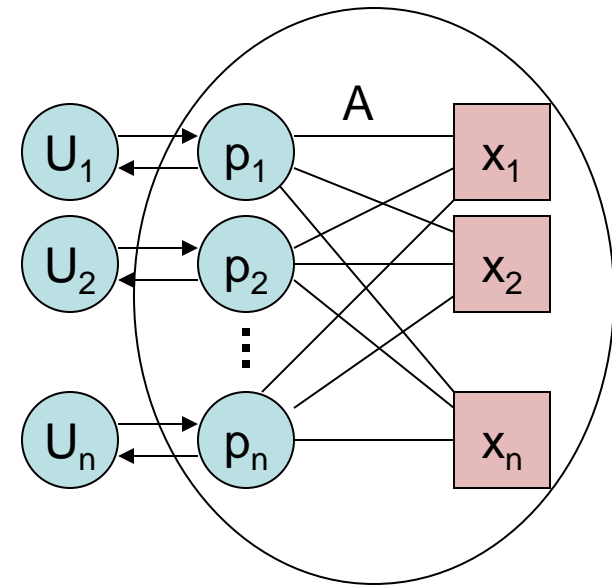




# Snapshot Atomic Objects

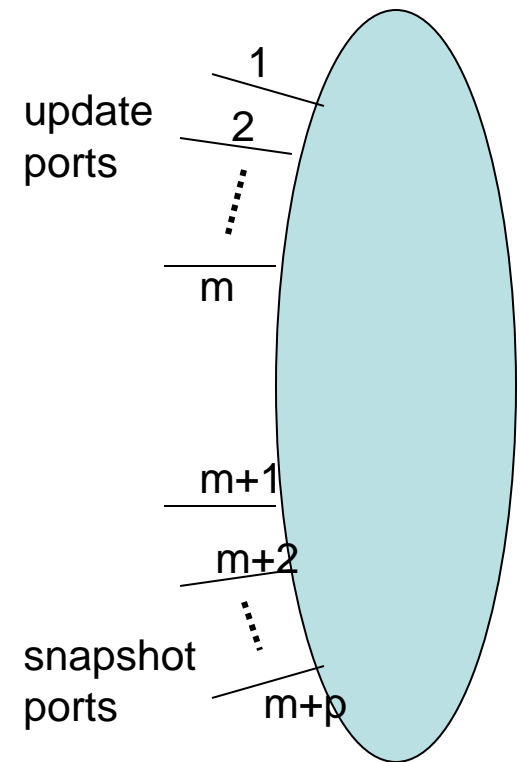
# Snapshot Atomic Objects

- Common shared-memory model:
  - Single-writer multi-reader read/write shared variables,
  - Each process writes to one variable, others read it.
- Limitation: Process can read only one variable at a time.
- **Atomic snapshot object** adds the capability for a process to read everyone's variables in one step.
- We will:
  - Define atomic snapshot objects.
  - Show that they do not add any power: they can be implemented, with wait-free termination, using only simple read/write shared variables!



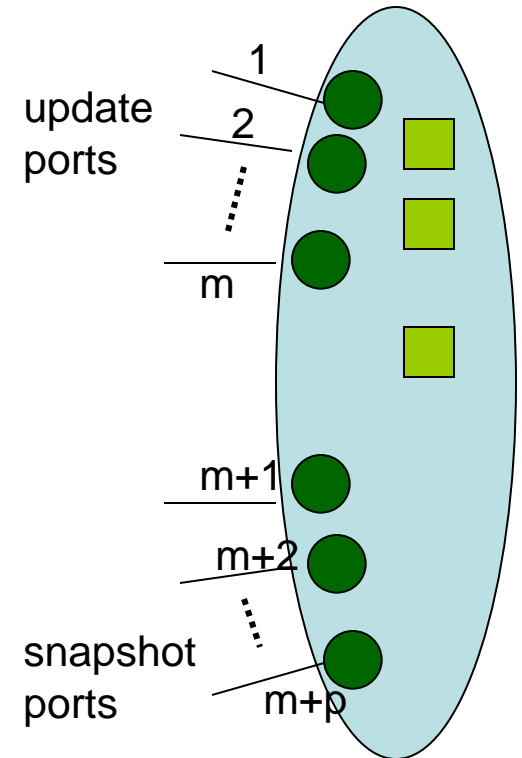
# Variable type for snapshot objects

- Assume a lower-level value domain  $W$  (for the values that the individual processes write), with initial value  $w_0$ .
- Value domain for the snapshot object: Vectors  $v$  of fixed length  $m$ , with values in  $W$ .
- Initial value:  $(w_0, w_0, w_0, \dots, w_0)$ .
- Invocations and responses:
  - **update**( $i, w$ ):
    - Writes value  $w$  into component  $i$ .
    - Responds “ack”.
  - **snap**:
    - Responds with the entire vector.
- External interface:  $m$  “update ports”,  $p$  “snapshot ports”.
- Each update port  $i$  is for updates of vector component  $i$ , **update**( $i, w$ ) <sub>$i$</sub> .



# Implementing snapshot atomic objects

- **Goal:** Implement an atomic snapshot object using a shared-memory system, one process per port, with only single-writer multi-reader shared variables.
- Unbounded-size-variable algorithm [Afek, Attiya, Dolev, Gafni, Merritt, Shavit]
- Also a bounded-size-variable version.
- **Shared variables:**
  - For each update port  $i$ , shared variable  $x(i)$ , written by update process  $i$ , read by everyone.
  - Each  $x(i)$  holds:
    - $val$ , an element of  $W$ .
    - $tag$ , a natural number.
    - Something else, we'll see shortly.
- Processes use all these separate read/write variables to implement a single snapshot atomic object.



# Idea 1

- **update(w,i)<sub>i</sub>:**
  - To write w to vector component i, update process i writes it in **x(i).val**.
  - Adds a tag that uniquely identifies the update (a sequence number, starting with 1).
- **snap:**
  - Read all the **x(i)s**, one at a time.
  - Read them all again.
  - If the two read passes yield the same **tags**, then return the vector of **x(i).val** values.
    - The vector actually appears in the memory at some point in real time.
    - That can be the serialization point for the snap.
  - If not, then keep trying, until two consecutive read passes yield the same **tags**.
  - This is correct, if it completes.
  - But the **snap** might never complete, because of continuing concurrent **updates**.

# Idea 2 (Clever)

- Suppose the **snap** sees the same  $x(i)$  variable with four different **tag** values  $t_1, t_2, t_3, t_4$ .
- Then it knows that the interval of the **update** operation that wrote  $t_3$  is entirely contained in the interval of the **snap**.
- Why:
  - Since the **snap** sees  $t_1$ , i's **update** with **tag**  $t_2$  doesn't finish before the **snap** starts.
  - So i's **update** with **tag**  $t_3$  starts after the **snap** starts.
  - Since the **snap** sees  $t_4$ , i's **update** with **tag**  $t_4$  must start before the **snap** finishes.
  - So i's **update** with **tag**  $t_3$  finishes before the **snap** finishes.
- So, modify **update** process i:
  - Before it writes to  $x(i)$ , executes its own **embedded-snap** subroutine, which is just like a **snap**.
  - When it writes **(val, tag)** to  $x(i)$ , also writes the result of its **embedded-snap**.
- Now, a **snap** that sees four different tags  $t_1, t_2, t_3, t_4$ , in  $x(i)$  returns the recorded value of the **embedded-snap** associated with  $t_3$ .
- **Embedded-snap** behaves the same as a real **snap**.

# In more detail:

- $x(i)$  contains:
  - $val$  in  $W$ , initially  $w_0$
  - $tag$ , a natural number, initially 0
  - $view$ , a vector indexed by  $\{1, \dots, m\}$  of  $W$ , initially  $(w_0)^m$ .
- **snap**:
  - Repeatedly read all  $x(i)$ s (any order) until one of the following occurs:
    - 2 passes yield the same  $x(i).tag$  for every  $i$ .
      - Then return the common vector of  $x(i).val$  values.
    - For some  $i$ , four distinct  $x(i).tags$  are seen.
      - Then return  $x(i).view$  from the third  $x(i).tag$ .
- **update(i,w)**:
  - Perform **embedded-snap**, same as **snap** (but don't return).
  - Write to  $x(i)$ :
    - $val := w$
    - $tag :=$  next sequence number (local count)
    - $view :=$  vector produced by the embedded snap
  - Return ack.

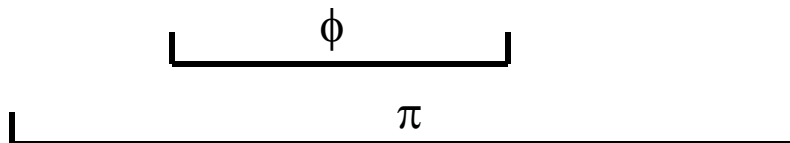
# Correctness

- **Theorem:** This algorithm implements a wait-free snapshot atomic object.
- **Proof:**
  - **Well-formedness:** Clear.
  - **Wait-free termination:** Easy---always returns by one case or the other.
  - **Atomicity:**
    - Show we can insert serialization points appropriately.
    - By **Lemma 13.10**, it's enough to consider executions in which all operations complete.
    - So, fix an execution  $\alpha$  of the algorithm + users, and assume that all operations complete in  $\alpha$ .
    - Insert serialization points:
      - For **update**: Just after the write step.
      - For **snap**: We need a more complicated rule:



# Serialization points for snaps

- Assign serialization points to all snaps/embedded-snaps.
- For every snap/embedded-snap that terminates by performing two read passes with the same tags (type 1):
  - Choose any point between end of the first pass and beginning of the second pass.
- Then, for all the snap/embedded-snaps that terminate by finding four distinct tags for some  $x(i)$  (type 2):
  - Insert serialization pts one by one, in order of operation completion.
  - For each such snap/embedded-snap  $\pi$  in turn:
    - The vector returned comes from some embedded-snap  $\phi$  (from some update) whose interval is completely contained within the interval of  $\pi$ :



- By the ordering,  $\phi$  has already been assigned a serialization point.
- Insert serialization point for  $\pi$  right after the one for  $\phi$ .

# Correctness

- All serialization points are in the required intervals:
  - updates:
    - Obvious.
  - Type 1 snaps/embedded-snaps (terminate with two identical read phases):
    - Obvious.
  - Type 2 snaps/embedded-snaps (terminate with four distinct values):
    - Argue inclusion by induction on the number of response events.
    - Use the containment property.
- Result of shrinking operations to their serialization points is a serial trace:
  - Because each snap returns the “correct” vector at its serialization point (result of all writes up to that point).
  - Easy for Type 1 snaps.
  - For Type 2 snaps, use induction on number of response events.

# Complexity

- **Shared memory size:**
  - $m$  variables for length  $m$  vector.
  - Each variable is of unbounded size (because  $x(i).tag$  is unbounded).
- **Time for snapshot:**
  - At most  $(3m+1)m$  shared memory accesses
  - $O(m^2 l)$  time
- **Time for update:**
  - Also  $O(m^2 l)$ , because of the embedded-snap.

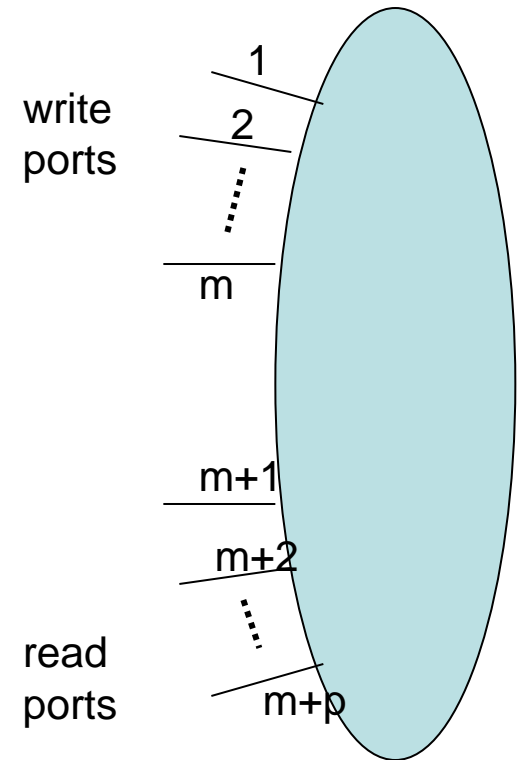
# Algorithm using bounded variables

- Also by [Afek, Attiya, Dolev, Gafni, Merritt, Shavit], based on ideas by Peterson.
- Uses bounded tags.
- Involves a slightly tricky handshake protocol.
- See [book, Section 13.3.3].
- Other snapshot algorithms have been developed, improving further on complexity, more complicated.
- **Moral:** Wait-free snapshot atomic objects can be implemented from simple wait-free read/write registers.
- So snapshots don't add extra computing power, compared to reads/writes.

# Read/Write Atomic Objects

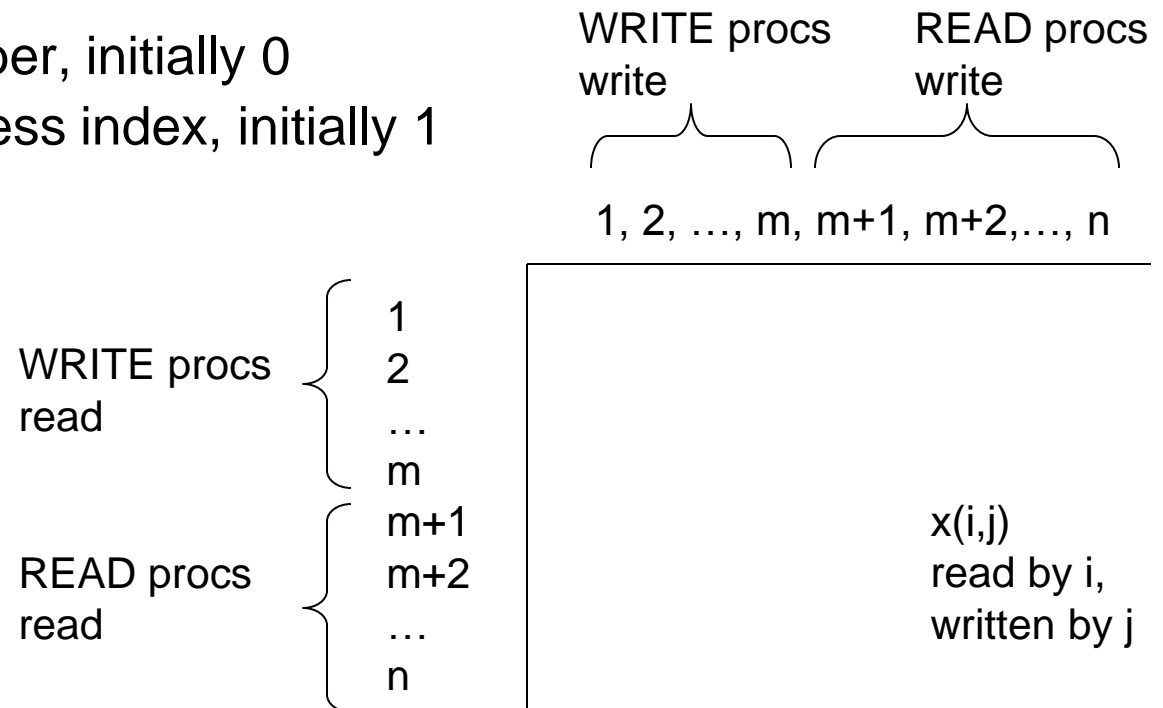
# Read/write atomic objects

- Consider implementing an atomic  $m$ -writer,  $p$ -reader register, using lower-level primitives.
- **Q:** What lower-level primitives?
- Try 1-writer, 1-reader registers.
- Several published algorithms, some quite complicated.
- Consider a simple one, with unbounded tags [Vitanyi, Awerbuch].
- **Caution:** Bounded-tag algorithm in that same paper is incorrect.



# Vitanyi-Awerbuch algorithm

- m-writer, p-reader read/write atomic objects from 1-writer, 1-reader read/write registers.
- Use  $n^2$  shared variables,  $n = m + p$ :
- Caps for high-level operations
- $x(i,j)$  has:
  - $val$  in  $V$ , initially  $v_0$
  - $tag$ , a natural number, initially 0
  - $index$ , a write process index, initially 1



# Vitanyi-Awerbuch algorithm

- **WRITE( $v$ )<sub>i</sub>:**
  - Process  $i$  reads all variables in its row.
  - Let  $k$  = largest **tag** it sees.
  - Writes to each variable in its column:
    - **val** :=  $v$ , **tag** :=  $k+1$ , **index** :=  $i$
  - Responds “ack”.
- **READ<sub>i</sub>:**
  - Process  $i$  reads all variables in its row.
  - Let  $(v,k,j)$  be a triple with maximum (tag,index) (lexicographic order).
  - “Propagates” this information by writing to each variable in its column:
    - **val** :=  $v$ , **tag** :=  $k$ , **index** :=  $j$
  - Finally, responds  $v$ .



# Correctness

- **Theorem:** Vitanyi-Awerbuch implements a wait-free m-writer p-reader read/write atomic object.
- **Well-formed, wait-free:** Easy.
- **Atomicity:**
  - Proceed as in snapshot proof, describing exactly where to put the serialization points?
  - But it's not so obvious where to put them:
    - E.g., each WRITE and READ does many write steps.
    - Contrast: Each update in snapshot algorithm does just one write step.
    - Placement of serialization points seems to depend on “races” between processes reading their rows and other processes writing their columns.
  - Use a different proof method:
    - Define a **partial ordering of the high-level operations**, based on (tag,index), and prove that the partial order satisfies certain conditions:

# A useful lemma (Lemma 13.16)

- Let  $\beta$  be a (finite or infinite) sequence of invocations and responses for a read/write atomic object, that is well-formed for each  $i$ , and that contains no incomplete operations.
- Let  $\Pi$  be the set of operations in  $\beta$ .
- Suppose there is an irreflexive partial ordering  $<$  of  $\Pi$  satisfying:
  1. For any operation  $\pi$  in  $\Pi$ , there are only finitely many operations  $\phi$  such that  $\phi < \pi$ .
  2. If the response for  $\pi$  precedes the invocation for  $\phi$  in  $\beta$ , then we don't have  $\phi < \pi$ .
  3. If  $\pi$  is a **WRITE** in  $\Pi$  and  $\phi$  is any operation in  $\Pi$  then either  $\pi < \phi$  or  $\phi < \pi$ .
  4. Value returned by each **READ** is the one written by the last preceding **WRITE**, according to  $<$ . (Or  $v_0$ , if there is no such **WRITE**.)
- Then  $\beta$  satisfies the atomicity property.

# Proof of lemma

- Insert serialization points using the rule:
  - Insert serialization point for  $\pi$  just after the latest of the invocations for  $\pi$  and for all operations  $\phi$  with  $\phi < \pi$ .
  - Condition 1 implies this latest invocation is well-defined.
  - Order contiguous serialization points consistently with  $<$ .
- **Claim 1:** The order of the serialization points is consistent with the  $<$  ordering on  $\Pi$ ; that is, if  $\phi < \pi$  then the serialization point for  $\phi$  precedes the serialization point for  $\pi$ .
- **Claim 2:** The serialization point for each  $\pi$  is in the interval of  $\pi$ .
  - Obviously after the invocation of  $\pi$ .
  - Could it be after the response of  $\pi$ ?
  - No: If it were, then the invocation for some  $\phi < \pi$  would come after the response of  $\pi$ , violating Condition 2.
- **Claim 3:** Each **READ** returns the value of the **WRITE** whose serialization point comes right before the **READ**'s serialization point.
  - Condition 3 says all **WRITES** are ordered w.r.t. everything.
  - Condition 4 says that the **READ** returns the result written by the last preceding **WRITE** in  $<$ .
  - Since order of ser. pts. is consistent with  $<$ , that's the right value to return.

# The Lemma

- Let  $\beta$  be a (finite or infinite) sequence of invocations and responses for a read/write atomic object, that is well-formed for each  $i$ , and that contains no incomplete operations.
- Let  $\Pi$  be the set of operations in  $\beta$ .
- Suppose there is an irreflexive partial order  $<$  of  $\Pi$  satisfying:
  1. For any operation  $\pi$  in  $\Pi$ , there are only finitely many operations  $\phi$  such that  $\phi < \pi$ .
  2. If the response for  $\pi$  precedes the invocation for  $\phi$  in  $\beta$ , then we don't have  $\phi < \pi$ .
  3. If  $\pi$  is a **WRITE** in  $\Pi$  and  $\phi$  is any operation in  $\Pi$  then either  $\pi < \phi$  or  $\phi < \pi$ .
  4. Value returned by each **READ** is the one written by the last preceding **WRITE**, according to  $<$ . (Or  $v_0$ , if there is no such **WRITE**.)
- Then  $\beta$  satisfies the atomicity property.

# Using the lemma to show atomicity for [Vitanyi, Awerbuch]

- Consider any execution  $\alpha$  of V-A, assume no incomplete operations.
- Construct a partial order based on (tag,index) pairs:
  - $\pi < \phi$  iff
    - $\pi$  writes (or propagates) a smaller tag pair than  $\phi$ , or
    - $\pi$  and  $\phi$  write (or propagate) the same tag pair,  $\pi$  is a **WRITE** and  $\phi$  is a **READ**.
  - That is,  $\pi < \phi$  iff
    - $\text{tagpair}(\pi) < \text{tagpair}(\phi)$ , or
    - $\text{tagpair}(\pi) = \text{tagpair}(\phi)$ ,  $\pi$  is a **WRITE** and  $\phi$  is a **READ**.
- Show this satisfies Conditions 1-4.
- Condition 1 follows from Condition 2 and the fact that there are no incomplete operations.
- Show Condition 2:

# Condition 2

- **Claim:** The (tag,index) pairs in any particular variable  $x(i,j)$  never decrease during  $\alpha$ .
- **Proof of Claim:**
  - $x(i,j)$  is written only by process  $j$ .
  - $j$ 's high-level operations are sequential.
  - Each operation of  $j$  involves reading row  $j$ , choosing a tag pair  $\geq$  the maximum one it sees, then writing it to column  $j$ .
  - Among the variables  $j$  reads is the diagonal  $x(j,j)$ , so  $j$ 's chosen pair is  $\geq$  the one in  $x(j,j)$ .
  - Since  $x(i,j)$  contains the same pair as  $x(j,j)$ ,  $j$ 's chosen pair is also  $\geq$  the one in  $x(i,j)$ .
  - Writes  $x(i,j)$  with this tag pair, nondecreasing.

# Condition 2

- **Condition 2:** If the response for  $\pi$  precedes the invocation for  $\phi$  in  $\beta$ , then we can't have  $\phi < \pi$ .
- **Proof:**
  - Suppose we have:  $\xrightarrow{\pi} \quad \quad \quad \xrightarrow{\phi}$
  - Then before the response event,  $\pi$  has written  $\text{tagpair}(\pi)$  to its entire column.
  - So (by Claim 1),  $\phi$  reads a  $\text{tagpair} \geq \text{tagpair}(\pi)$ .
  - Then (by the way the algorithm works),  $\phi$  chooses a tag pair,  $\text{tagpair}(\phi)$ , that is  $\geq \text{tagpair}(\pi)$ ; furthermore, if  $\phi$  is a **WRITE**, then  $\text{tagpair}(\phi) > \text{tagpair}(\pi)$ .
  - Then we can't have  $\phi < \pi$ :
    - Since  $\text{tagpair}(\phi) \geq \text{tagpair}(\pi)$ , the only way we could have  $\phi < \pi$  is if  $\text{tagpair}(\phi) = \text{tagpair}(\pi)$ ,  $\phi$  is a **WRITE** and  $\pi$  is a **READ** (by definition of  $<$ ).
    - But in this case,  $\text{tagpair}(\phi) > \text{tagpair}(\pi)$ , contradiction.

# Condition 3

- **Condition 3:** **WRITEs** are ordered with respect to each other and with respect to all **READs**.
- **Proof:**
  - Follows because all **WRITEs** get distinct tagpairs.
  - Why distinct?
    - Different ports: Different indices.
    - Same port  $i$ :
      - **WRITEs** on port  $i$  are sequential.
      - Each **WRITE** by  $i$  reads its previous tag in its own diagonal variable  $x(i,i)$  and chooses a larger tag.
- **Condition 4:** LTTR
- Apply the Lemma, implies that V-A satisfies atomicity, as needed.

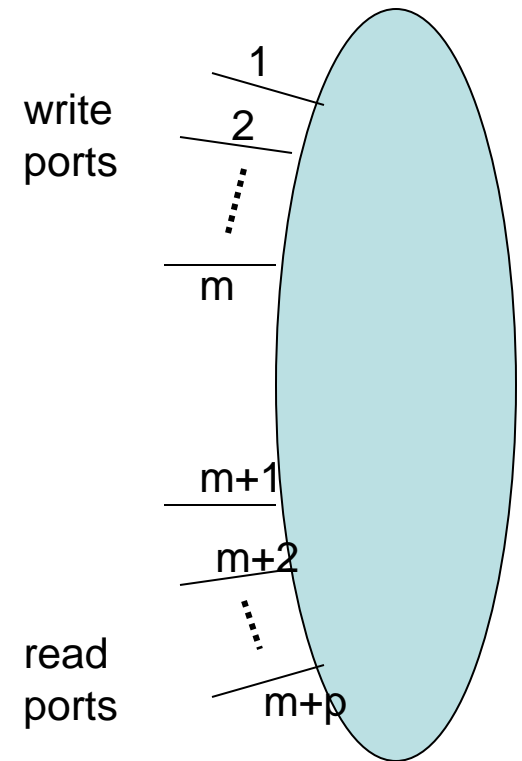


# Complexity

- Shared memory size:
  - $(m + p)^2$  variables, each of unbounded size (because of  $x(i).tag$ ).
- Time for read:
  - At most  $2(m + p)$  shared memory accesses
  - $O((m + p) l)$  time
- Time for write:
  - Also  $O((m + p) l)$

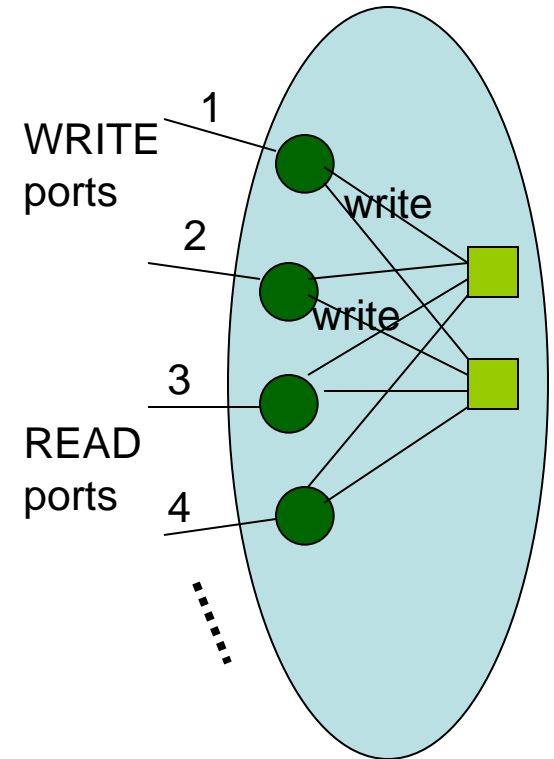
# More on read/write atomic objects

- [Vitanyi, Awerbuch] algorithm is not too costly in time, but uses unbounded variables.
- **Q:** Can we implement multi-writer multi-reader atomic objects in terms of single-writer single-reader registers, using bounded variables?
- Yes. Several published algorithms:
  - [Burns, Peterson]
  - [Dolev, Shavit]
  - [Vidyasankar]
  - ...
  - Bounded-tag algorithm in [Vitanyi, Awerbuch] incorrect.
- Fairly complicated, costly.
- Usually divide the problem into:
  - 1-writer multi-reader from 1-writer 1-reader.
  - Multi-writer multi-reader from 1-writer multi-reader.



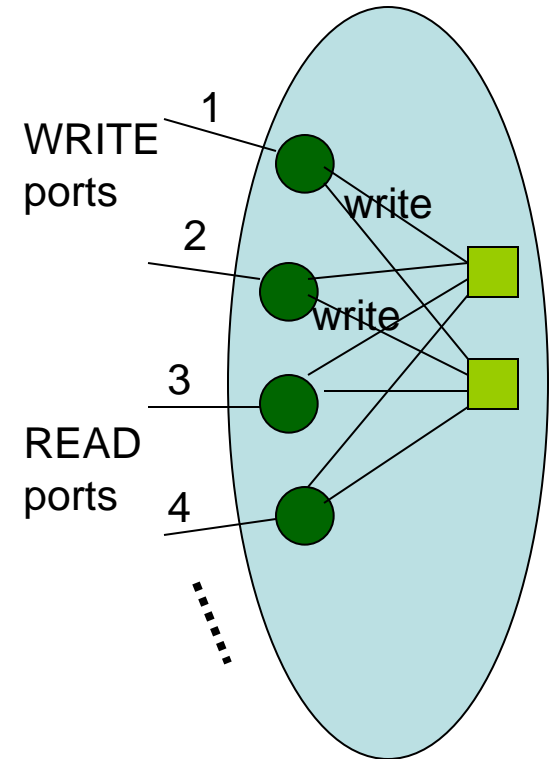
# Bloom algorithm

- A simple special case, illustrates:
  - Typical difficulties that arise
  - Interesting proof methods
- 2-writer multi-reader register from 1-writer multi-reader registers
- **Shared variables:**
  - $x(1)$ ,  $x(2)$ , with:
    - $val$  in  $V$ , initially  $v_0$
    - $tag$  in  $\{0,1\}$ , initially 0
  - $x(1)$  written by WRITER 1, read by everyone
  - $x(2)$  written by WRITER 2, read by everyone



# Bloom algorithm

- **WRITE( $v$ )<sub>1</sub>:**
  - Read  $x(2).tag$ , say  $b$
  - Write:
    - $x(1).val := v$ ,
    - $x(1).tag := 1 - b$
  - Tries to make tags unequal.
- **WRITE( $v$ )<sub>2</sub>:**
  - Read  $x(1).tag$ , say  $b$
  - Write:
    - $x(2).val := v$ ,
    - $x(2).tag := b$
  - Tries to make tags equal.
- **READ:**
  - Read both registers.
  - If **tags** are unequal then reread and return  $x(1).val$ .
  - If **tags** are equal then reread and return  $x(2).val$ .

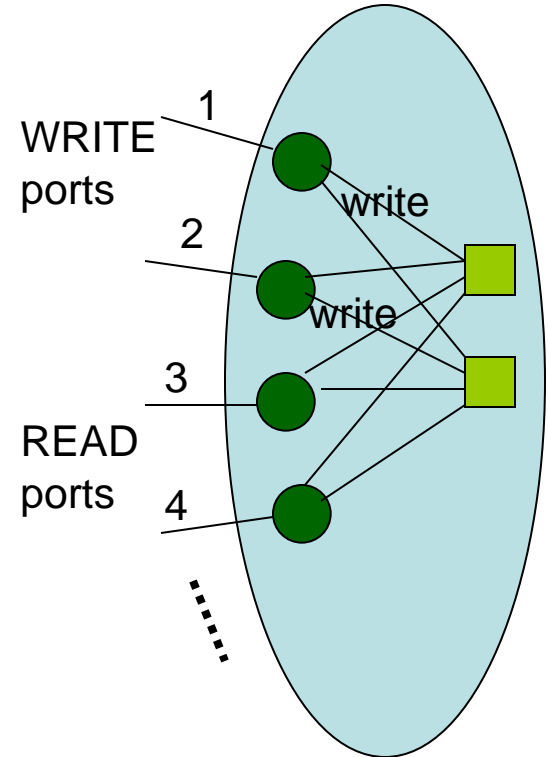


# Correctness

- Well-formedness, wait-freedom: Clear
- Atomicity:
  - Could use:
    - Explicit serialization points, or
    - Partial-order lemma
  - Instead, use a **simulation relation**, mapping the algorithm to a simpler unbounded-tag version

# Unbounded-tag algorithm

- Shared variables:
  - $x(1), x(2)$ , with:
    - $val$  in  $V$ , initially  $v_0$
    - $tag$ , a natural number; initially  $x(1).tag = 0, x(2).tag = 1$
- **WRITE( $v$ )<sub>1</sub>:**
  - Read  $x(2).tag$ , say  $t$
  - Write  $x(1).val := v, x(1).tag := t + 1$
- **WRITE( $v$ )<sub>2</sub>:**
  - Read  $x(1).tag$ , say  $t$
  - Write  $x(2).val := v, x(2).tag := t + 1$
- **READ:**
  - Read both registers, get  $tags$   $t_1$  and  $t_2$ .
  - If  $|t_1 - t_2| \leq 1$  then reread the register  $x(i)$  with the higher  $tag$  and return  $x(i).val$ .
  - Else reread and return either (choose nondeterministically)



# Why the nondeterministic choice?


- The extra generality provided by the nondeterministic choice is needed to make a simulation relation from the Bloom algorithm work correctly.
- The integer algorithm works even with the nondeterministic choice.
- The nondeterminism doesn't significantly complicate the integer algorithm.
- Doesn't complicate the proof at all; in fact, makes it easier to see what's needed.

# Proof for the integer algorithm

- **Invariant:**
  - $x(1).tag$  is always even
  - $x(2).tag$  is always odd
  - $|x(1).tag - x(2).tag| = 1$
- **Well-formedness, wait-freedom:** Clear
- **Atomicity:**
  - E.g., use the **partial-order lemma**.
  - Define the partial order  $<$  using the tags:
    - Order **WRITEs** by the tags they write.
    - Break ties (sequential operations by the same WRITER) in temporal order.
    - Insert each **READ** just after the **WRITE** whose value it gets.
  - Check Conditions 1-4 of the partial order lemma.



## E.g., Condition 2

- **Condition 2:** If the response for  $\pi$  precedes the invocation for  $\phi$  in  $\beta$ , then we can't have  $\phi < \pi$ .
- **Proof:**
  - Suppose we have: 
  - Consider cases based on the types of  $\pi$  and  $\phi$ .
  - Most interesting case:  $\pi$  is a **WRITE**,  $\phi$  is a **READ**.
  - Suppose **WRITE**  $\pi$  is done by WRITER  $i$ , writes tag  $t$ .
  - Must show we can't have  $\phi < \pi$ .
  - That is, we must show that **READ**  $\phi$  must return either the result written by **WRITE**  $\pi$  or one by some other **WRITE**  $\psi$  with  $\pi < \psi$ .

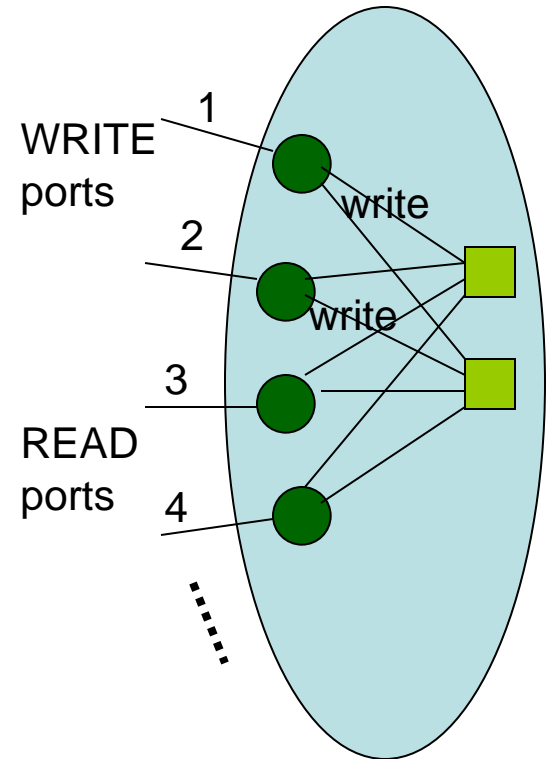
# Proof of Condition 2, cont'd



- Show  $\phi$  must return either the result written by  $\pi$  or one by some other **WRITE**  $\psi$  with  $\pi < \psi$ .
- When **READ**  $\phi$  is invoked,  $x(i).tag \geq t$ , by monotonicity.
- At that point,  $x(2-i).tag \geq t - 1$ , by invariant.
- Only possible problem:  $\phi$  returns the value of a **WRITE** with  $tag = t - 1$ .
- Suppose it does; then  $\phi$  must see  $x(2-i).tag = t - 1$  on its initial read of  $x(2-i)$ , and also on the third read.
- What might  $\phi$  see for  $x(i).tag$  on its initial read of  $x(i)$ ?
- 2 possibilities:
  - $\phi$  sees  $x(i).tag = t$ .
    - Then it would reread  $x(i)$ , contradiction.
  - $\phi$  sees  $x(i).tag > t$ .
    - Then by the time it sees this,  $x(2-i).tag$  is already  $> t - 1$ .
    - So  $\phi$  couldn't see  $x(2-i).tag = t-1$  on the third read, contradiction.

# Where are we?

- Integer version of Bloom algorithm (IB) implements a 2-writer multi-reader atomic object from 1-writer multi-reader registers.
- Now show that the original Boolean Bloom algorithm (BB) implements the integer version.
- Use a simulation relation from BB to IB.
- **Q:** What is the simulation relation?



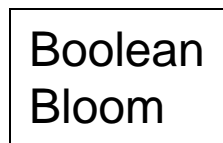
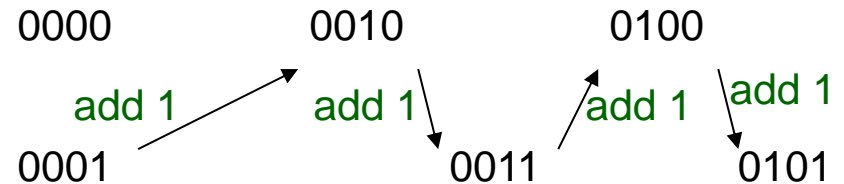
# Simulation relation from BB to IB

- If  $s$  is a state of Boolean Bloom system,  $u$  a state of IntegerBloom system, then define  $(s,u)$  in  $R$  exactly if:
  - Each occurrence of a **tag** in BB is exactly the **second low-order bit** of the corresponding **tag** occurrence in IB.
  - All other state components are identical in the two systems.
- Note this is multivalued: Each state of BB corresponds to many states of IB.
- Example:**



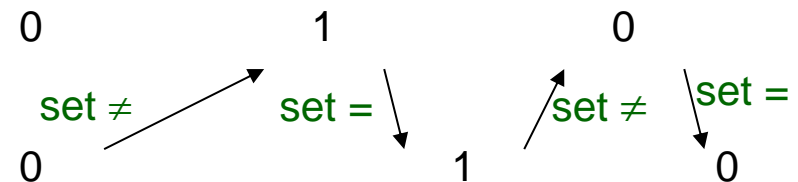
$x(1).tag:$

$x(2).tag:$



$x(1).tag:$

$x(2).tag:$



# R is a simulation relation

- **Proof:**
  - **Start states related:**
    - Second low-order bit of 0000 is 0
    - Second low-order bit of 0001 is 0
  - **Step condition:**
    - For any step  $(s, \pi, s')$  in BB, and any state  $u$  of IB such that  $(s, u)$  in R, the corresponding step of IB is almost the same:
      - Same kind of action, same process, same register...
    - Must show:
      - The IB step is enabled, and
      - The state correspondence is preserved.
    - Key points:
      - The write step of a **WRITE** preserves the state correspondence.
      - The third read of a **READ** is enabled in IB (for the same register).

# First key point

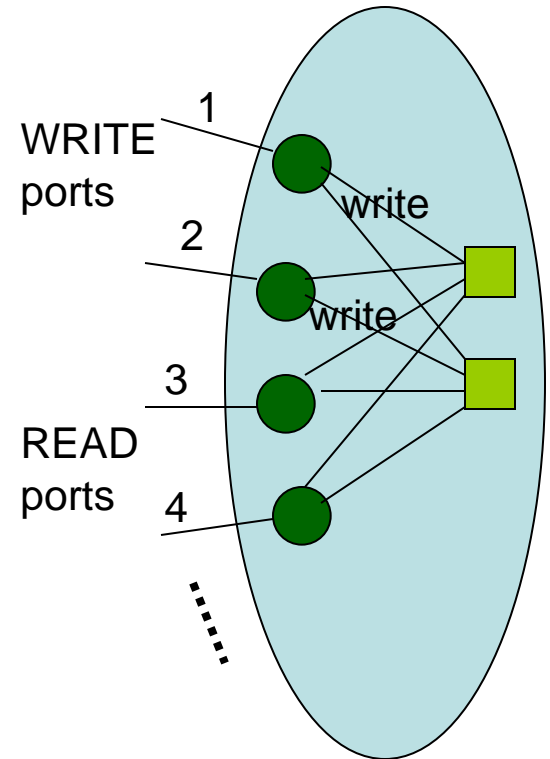
- The write step of a **WRITE** operation preserves the state correspondence.
- **Proof:**
  - E.g., a **WRITE** by process 1.
  - Writes to  $x(1).tag$ :
    - $1-b$ , where  $b$  is the value read from  $x(2).tag$ , in BB.
    - $t+1$ , where  $t$  is the value read from  $x(2).tag$ , in IB.
  - By relation  $R$  on the pre-states,  $b$  is the second low-order bit of  $t$ .
  - We need to show that  $1-b$  is the second low-order bit of  $t+1$ .
  - Follows because:
    - $t$  is odd (by the invariant, process 2's **tag** is always odd), and
    - Incrementing an odd number always flips the second low-order bit.
  - **Example:**
    - $t = 101, \quad b = 0$
    - $t + 1 = 110, \quad 1-b = 1$
  - Argument for process 2 is similar.

# Second key point

- IB allows reading the same third register as BB.
- Proof:
  - Choice of register is based on the **tags** read in the first two reads.
    - In BB: Read  $x(1).tag = b_1$ ,  $x(2).tag = b_2$ .
    - In IB: Read  $x(1).tag = t_1$ ,  $x(2).tag = t_2$ .
    - By state correspondence,  $b_1$  and  $b_2$  are the second low-order bits of  $t_1$  and  $t_2$ , respectively.
  - Consider cases:
    - $t_1 = t_2 + 1$ 
      - Then IB reads from  $x(1)$  on third read.
      - Since  $t_1$  is even and  $t_2$  is odd, second low-order bits are unequal.
      - Thus,  $b_1 \neq b_2$ , and so BB also reads from  $x(1)$  on third read.
    - $t_2 = t_1 + 1$ 
      - Symmetric, both read from  $x(2)$  on the third read.
    - Neither of these holds.
      - Then IB allows either to be read (nondeterministic).

# Now where are we?

- Argued simulation reln. from BB to IB.
- Implies every trace of BB is a trace of IB.
- Showed that IB satisfies atomicity.
- Trace inclusion implies that BB also satisfies atomicity.
- **Theorem:** The Bloom algorithm implements a 2-writer multi-reader atomic object from 1-writer multi-reader registers.
- Unfortunately...
- This algorithm doesn't immediately extend to three or more writers.
- Algorithms exist for this, but they (and their proofs) are much more complicated.





# More Atomic Objects

- Important in both shared-memory and distributed settings.
- Multiprocessor memories (next time)
- Lamport's state machines
  - Implemented using replicated state machine (RSM) approach, based on logical time.
- Fault-tolerant data servers
  - Replication-based approaches (later this term)
  - Coding-based approaches
- Sometimes settle for weaker consistency conditions.

# Next time...

- Wait-free computability
- The wait-free consensus hierarchy
- Reading:
  - [Herlihy, Wait-free synchronization],
  - [Attiya, Welch, Chapter 15]