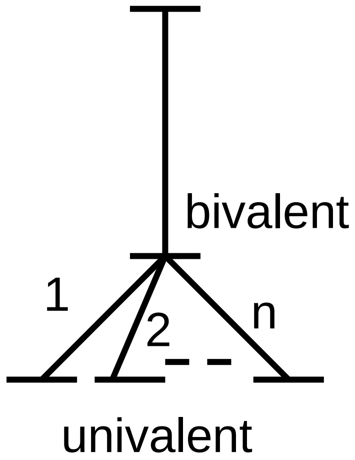
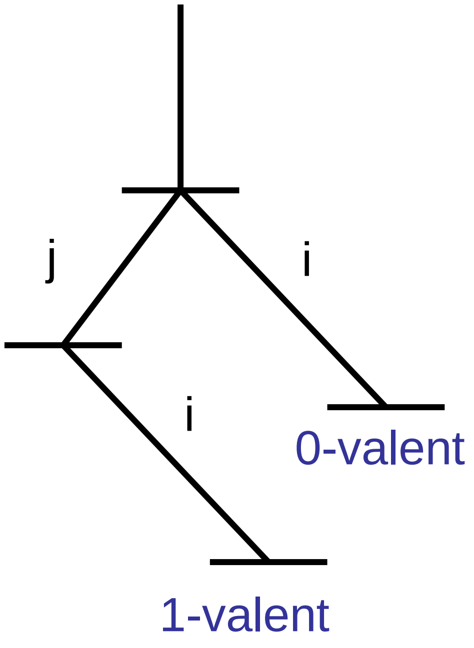
6.852: Distributed Algorithms
Fall, 2015

Class 18

Impossibility results for consensus in asynchronous shared-memory systems with failures





Impossibility of agreement

- Main Theorem [Fischer, Lynch, Paterson], [Loui, Abu-Amara]:
  - For $n \geq 2$, there is no algorithm in the read/write shared memory model that solves the agreement problem and guarantees 1-failure termination.
- A Weaker Theorem [Herlihy]:
  - For $n \geq 2$, there is no algorithm in the read/write shared memory model that solves the agreement problem and guarantees wait-free termination.

Importance of the read/write data type

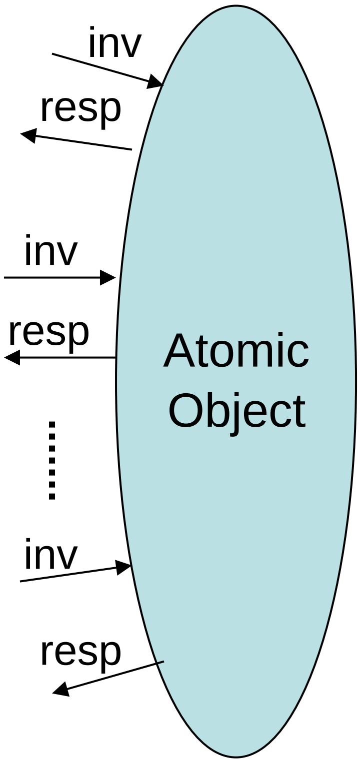
- Consensus impossibility result doesn't hold for more powerful data types.
- Example: Read-modify-write shared memory
  - Very strong primitive.
  - In one step, can read variable, do local computation, and write back a value.
  - Easy algorithm:
    - One shared variable x, value in $V \cup \{\perp\}$, initially $\perp$.
    - Each process i accesses x once.
    - If it sees:
      - $\perp$, then it changes the value in x to its own initial value and decides on that value.
      - Some v in V, then it decides on that value.
- Read/write registers are similar to asynchronous FIFO reliable channels---we'll see the precise connection later.

Today's plan

- Atomic objects:
  - Basic definitions
  - Canonical atomic objects
  - Atomic objects vs. shared variables
- Reading: Sections 13.1-13.2
- Next time:
  - Algorithms to implement atomic objects:
    - Atomic snapshots
    - Atomic read/write registers
  - Reading: Sections 13.3-13.4

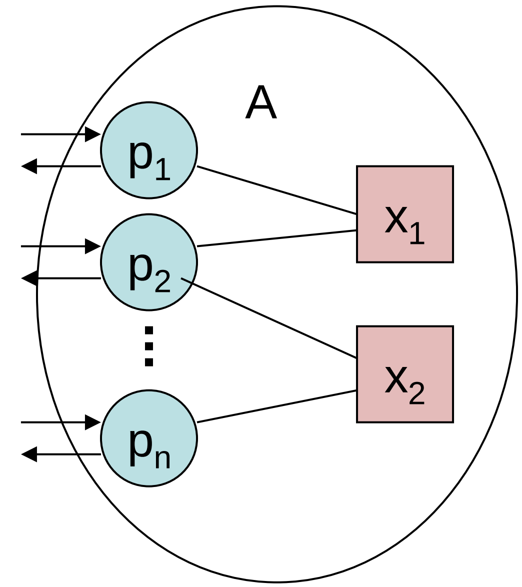
Atomic Objects

- Atomic objects are fundamental building blocks for fault-tolerant distributed systems and multiprocessor systems.
- An atomic object is a version of a shared variable, with invocation and response events separated, rather than being combined into one indivisible event.
- Also consider (stopping) failures.
- Separating invocations and responses allows us to consider lower-level implementations of these objects.
  - These can use shared memory, or distributed network algorithms.
  - For shared memory, we can “layer” the development through several levels.



Shared memory model

- A single I/O automaton with processes and variables “inside”.
  - Separation is expressed by locality restrictions on the actions and transitions.
  - Processes and variables aren’t separate automata.
  - Doesn’t exploit I/O automaton composition.
  - We can’t talk about “implementing” shared variables with lower-level distributed algorithms.



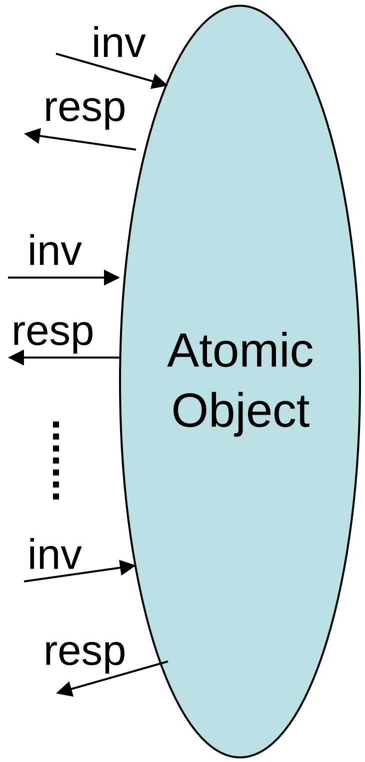
- Q: Can we model each process and variable as a separate I/O automaton?
  - Split operations on variables into separate invocation and response actions.
  - But we still want an (invocation, response) pair to “look like” an instantaneous access.

Atomic Objects

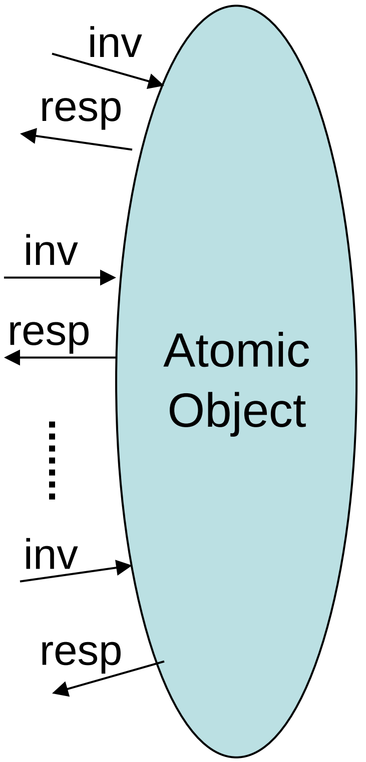
- Q: Can we model each process and variable as a separate I/O automaton?

- Define atomic objects.
- Atomic object of a given type is similar to an ordinary shared variable of that type, but it also allows concurrent accesses.
- Interface has invocation inputs and response outputs.
- Invocation/response behavior "looks like" that of an instantaneous-access shared variable.
- Looks "as if" operations occur one at a time, sequentially, in some order consistent with order of invocations and responses.

- AKA linearizable objects [Herlihy, Wing]



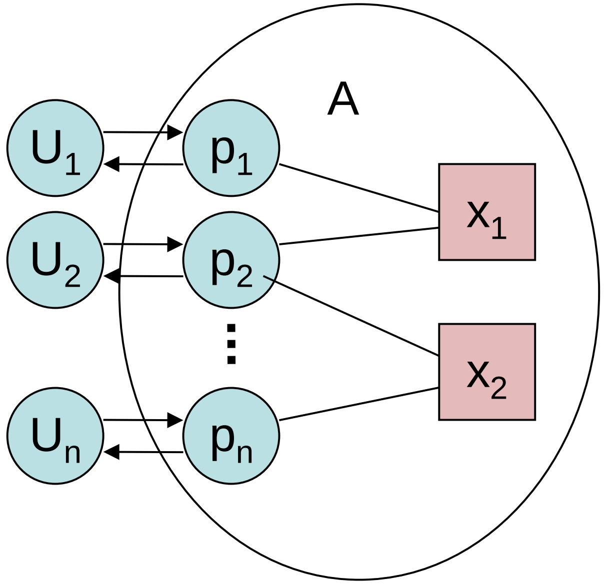
Atomic Objects

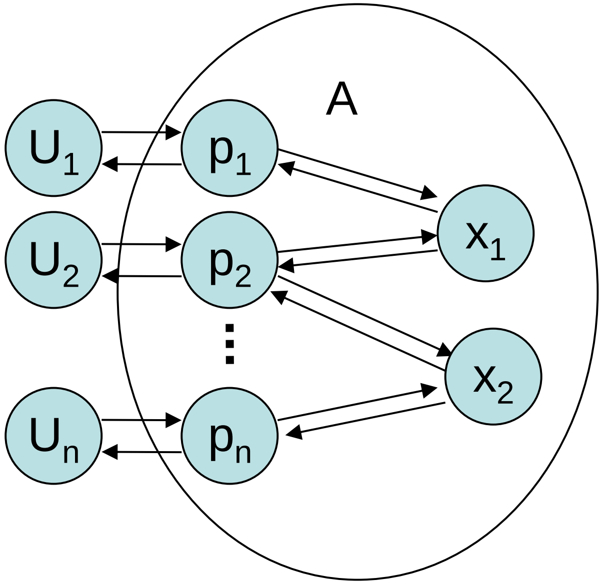


- Interface has invocation inputs and response outputs.
- Looks as if operations occur one at a time, sequentially, in some order consistent with order of invocations and responses.
- Also, consider fault-tolerance conditions (stopping failures only), as for consensus:
  - Wait-free termination
  - f-failure termination
  - Etc.
- Separating invocations and responses allows us to consider lower-level implementations of atomic objects.
  - Shared-memory algorithms, or distributed network algorithms.
  - For shared memory algorithms, we can develop algorithms hierarchically, using several levels.
- Atomic objects are important building blocks for multiprocessor systems and distributed systems.

Replacing variables with atomic objects

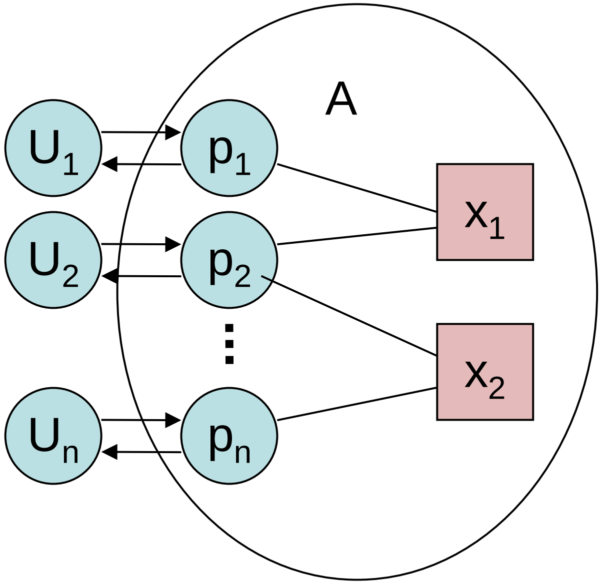
- Now processes and objects are all I/O automata, combined using ordinary automata composition.
- Interactions:
  - Processes access atomic objects via invocations, get responses.
  - Invocations are outputs of processes, inputs of objects.
  - Responses are outputs of objects, inputs of processes.
  - May be a time delay between invocation and response.

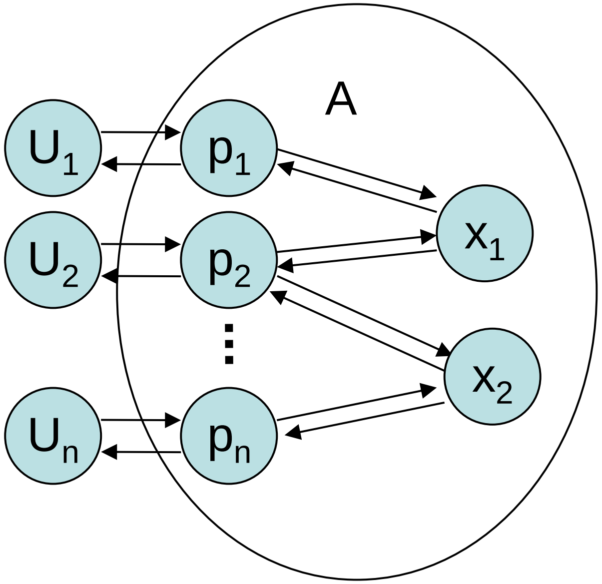




Replacing variables with atomic objects

- Locality constraints are now expressed by the I/O automata decomposition.
- More complicated than shared variables:
  - More actions (invocations/responses instead of complete accesses).
  - Algorithms have more steps, more bookkeeping.
  - More stuff to reason about.
- More realistic system model.

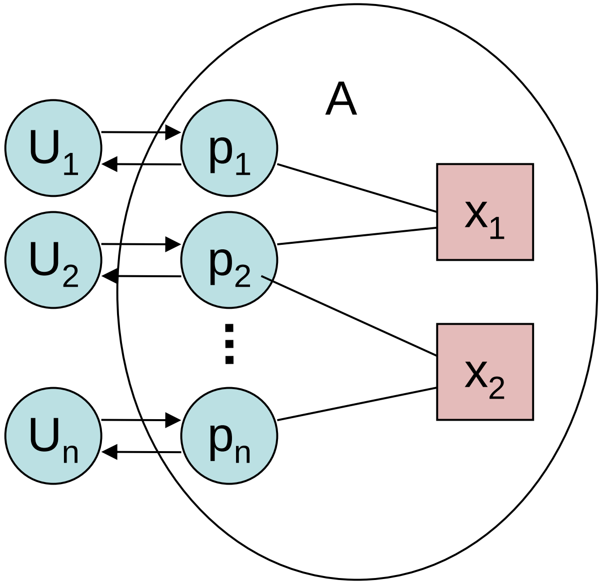


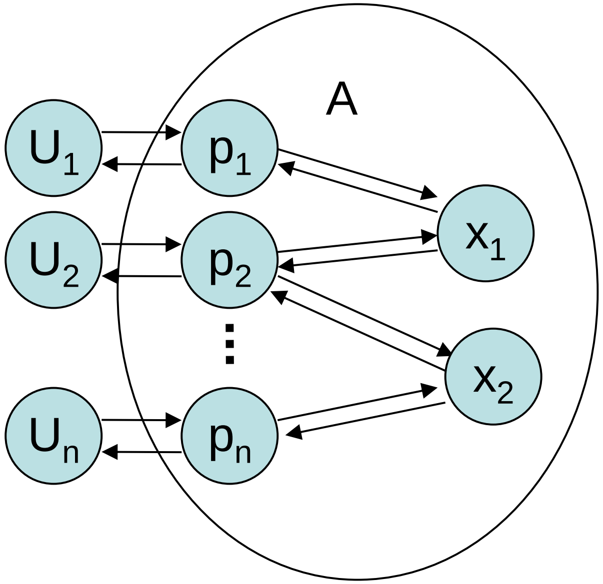


Replacing variables with atomic objects

- Q: But how do we know that this replacement doesn't introduce new behaviors?
- We need some restrictions to get "equivalence".

- Q: What can we say about failure behavior, when we make this replacement?





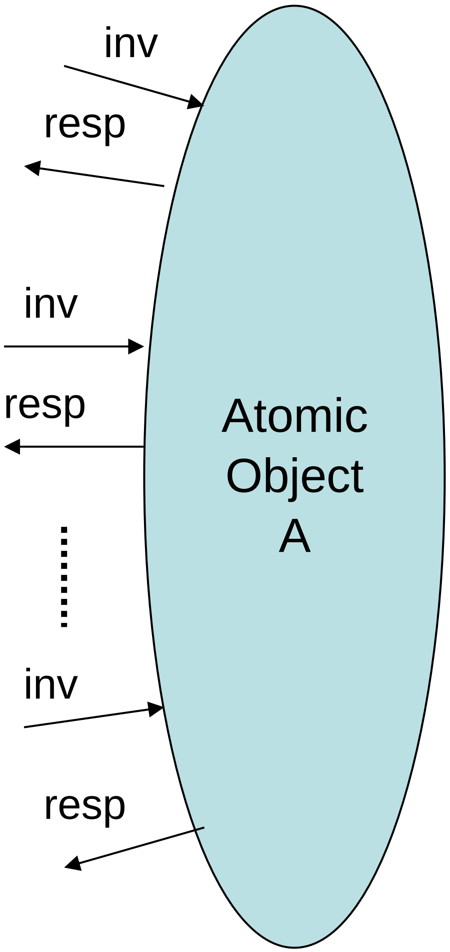
Atomic Objects:
Basic Definitions

Variable Types

- Variable type: $(V, v_0, invs, resps, f)$
  - V: Set of values
  - v_0: Initial value
  - invs: Set of invocations
  - resps: Set of responses
  - f: $invs \times V \rightarrow resps \times V$
    - Describes responses to an invocation and changes to the variable.
- AKA State machine [Lamport]
- AKA Sequential specification [Herlihy]
- Execution: $v_0, a_1, b_1, v_1, a_2, b_2, v_2, a_3, b_3, v_3, a_4, b_4, v_4, \ldots$
  - v_i is value; a_i is invocation; b_i is response
  - Ends with value (if finite).
  - $(b_i, v_i) = f(a_i, v_{i-1})$ for $i > 0$.
- Trace: $a_1, b_1, a_2, b_2, a_3, b_3, a_4, b_4, \ldots$ (i.e., just invocations and responses, hide the variable values)

Atomic objects

- Atomic object A of a given variable type is an I/O automaton with a particular kind of interface, satisfying some conditions:
  - Well-formedness
  - Atomicity
  - Liveness (termination)
- External interface:
  - Assume “ports” 1, 2, ..., n (one for each process).
  - May restrict so that some invocations are allowed on only some of the ports.
  - Also allow stop inputs on all ports, as before.
- Compose with users U_i, assumed to preserve well-formedness (alternating invocations and responses at each port, starting with an invocation).



Conditions satisfied by A

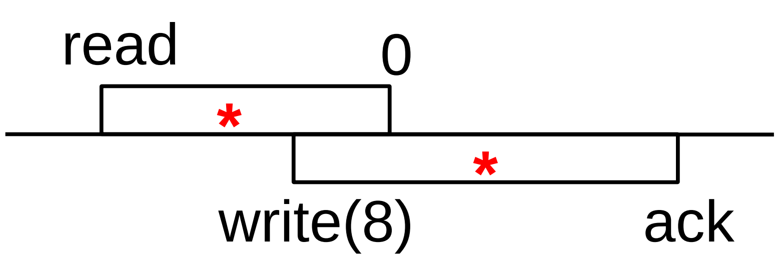
- Preserves well-formedness (alternating invocations and responses at each port, starting with an invocation).
- Atomicity:
  - First define when a well-formed sequence β of invocations and responses (at all ports) is atomic.
  - Then A satisfies atomicity iff all well-formed executions of $A \times U$, where $U = \Pi U_i$ (for any users), have atomic traces.
- First suppose that all invocations have matching responses (that is, the sequence β is complete).
- Then we say β is atomic provided that it's possible to insert a serialization point (dummy event) somewhere between each invocation and matching response, such that, if all the invs and resps are moved to their serialization points, the result is a trace of the (serial) variable type.

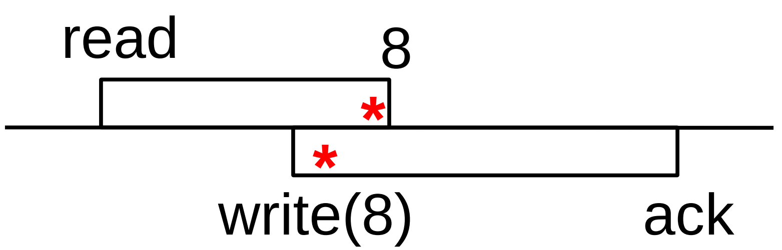
Atomicity for complete sequences

- Suppose β is a complete well-formed sequence of invocations and responses.

  Then β is atomic provided that one can insert a serialization point between each invocation and matching response, such that, if all the invs and resps are moved to their serialization points, the result is a trace of the (serial) variable type.

- Examples: Initial value 0.





- read, 0, write(8), ack is correct for serial specification.
- write(8), ack, read, 8 is also correct.

Alternative definition [Herlihy]

- Suppose β is a complete well-formed sequence of invocations and responses. Then β is atomic provided that it can be reordered to a trace of the variable type, while preserving:
  – The order of events at each process, and
  – The order of any response and following invocation (anywhere).
- Equivalent.

Complication: Incomplete operations

- Q: What about sequences β containing some incomplete operations? Which ops should get serialization points?
- We can't require that we include serialization points for all such operations (operation might fail right after invocation).
- We can't require that we exclude all such operations (operation might fail just before returning).
- So, we leave it optional…
- Require that it's possible to:
  - Insert serialization points for all complete operations (between invocations and responses).
  - Select some arbitrary subset Φ of incomplete operations.
  - For each operation in Φ, insert a serialization point somewhere after the invocation, and make up a response.
  - In such a way that moving all matched invocations and responses to their serialization points (and removing other invocations) yields a trace of the variable type.

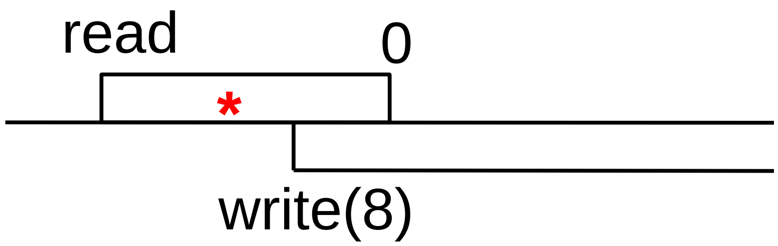
Atomic sequences, in general

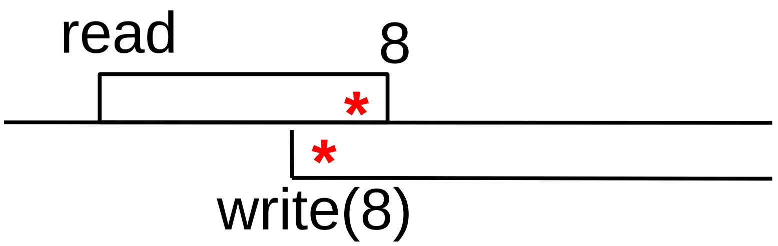
- Suppose β is any well-formed sequence of invocations and responses.

  Then β is atomic provided that one can
  - Insert serialization points for all complete operations.
  - Select a subset Φ of incomplete operations.
  - For each operation in Φ, insert a serialization point somewhere after the invocation, and make up a response.
  - In such a way that moving all matched invs and their resps to the serialization points (and removing other invs) yields a trace of the variable type.

More atomicity examples

- Initial value 0.

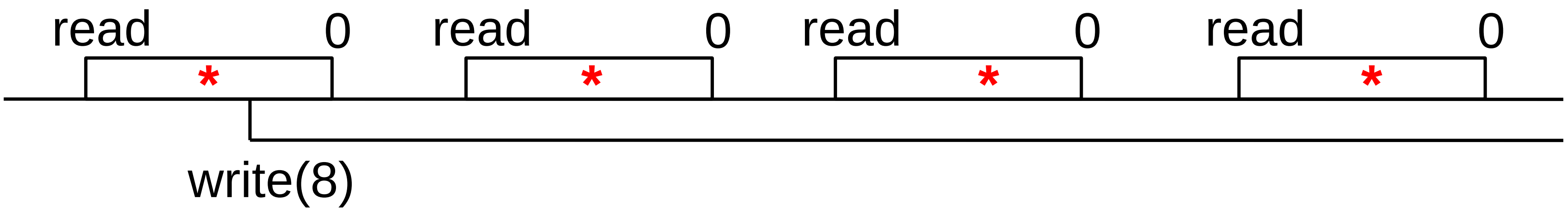




- read, 0 is correct for serial specification.
- write(8), ack, read, 8 is correct.

Another atomicity example

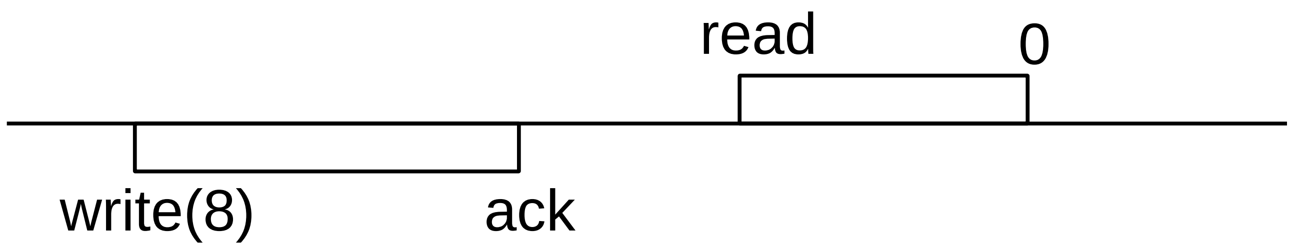
- Initial value 0.



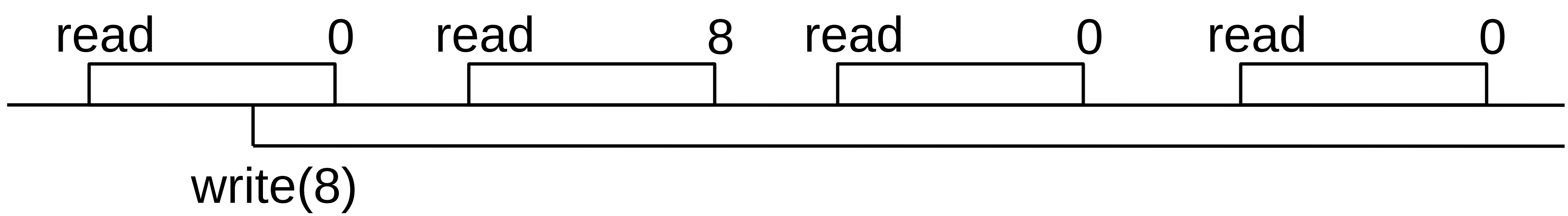
- read, 0, read, 0,…(forever) is correct.
- The write does not (cannot) get a serialization point.

Some non-atomic sequences

- Write not seen:



- Out-of-order reads



Note on the atomicity property

- [Well-formedness + atomicity] is a safety property.
- More precisely, let P be the trace property, for sequences of invocations and responses, expressing:
  - Well-formedness for every port, plus
  - Atomicity.
- Then P is a safety property.
- In other words, if this combination doesn't hold, the violation occurs at some particular point in the sequence.
- Plausible, but not completely obvious---proved in book, p. 405.
  - Uses Konig's Lemma to show limit-closure.
  - That is, if we can assign serialization points correctly to successively-extended finite sequences, then there is a way to assign them to their infinite limiting sequence.

Back to the conditions satisfied by an atomic object A…

- Preserves well-formedness.
- Atomicity:
  - We just defined when a well-formed sequence β of invocations and responses (at all ports) is atomic.
  - Then A satisfies atomicity iff all well-formed executions of $A \times U$, where $U = \Pi\, U_i$ (for any users) have atomic traces.
- Liveness (termination):

Liveness

- Failure-free termination (basic requirement for atomic objects):
  - In any fair failure-free execution of $A \times U$, every invocation has a matching response.
  - “Fair” here refers to fairness in the underlying I/O automata model---A keeps taking steps.
- Definition: Automaton A (with the right interface) is an atomic object if it satisfies well-formedness, atomicity, and failure-free termination (for all U).

Other liveness conditions

- As for consensus, we sometimes consider other liveness conditions, expressing fault-tolerance properties.
- Wait-free termination: In any fair execution of $A \times U$, every invocation on a non-failing port gets a response.
- f-failure termination, $0 \leq f \leq n$: In any fair execution of $A \times U$ in which failures occur on $\leq f$ ports, every invocation on a non-failing port gets a response.

Example: A wait-free atomic object

- Variable type:
  - Natural numbers, initial value 0.
  - read and increment operations.
- Atomic object supports read and increment operations on all ports.
- Implement with an n-process shared-memory system.
- Shared read/write registers
  - $x(i)$, $1 \leq i \leq n$, natural number, initially 0.
  - $x(i)$ writable by i, readable by all.
- To implement $increment_i$:
  - Process i increments its own variable $x(i)$.
  - Can do this using a write operation, by remembering the previous value written.
- To implement $read_i$:
  - Process i reads all the shared variables, one at a time, in any order, and returns the sum.
- Q: Why does this work?

Read/Increment algorithm

- increment_i: Increment $x(i)$.
- read_i: Read all the shared variables, one at a time, in any order, and return the sum.
- Proof:
  - Well-formed, wait-free: Immediate.
  - Atomic: Say where to put the serialization points.
    - For an increment: At the actual write step.
    - For a complete read:
      - Must be somewhere between invocation and response.
      - Read returns a value v such that sum of the x's at the beginning $\leq$ v $\leq$ sum at the end.
      - Since the sum increases by one each time, there is some point where sum of the x's = v.
      - Put the serialization point there.
    - For an incomplete read: Don't bother.
- Correctness depends on the particular kinds of operations.

Canonical Atomic Object Automata

Canonical atomic object automaton

- Express the set of traces acceptable for a wait-free atomic object as the fair traces of a particular canonical object automaton; see Section 13.1.2.
- Could generalize to f-failure termination (we'll see this later).
- Canonical object automaton keeps internal copy of the variable, plus delay buffers for invocations and responses.
- Behavior: 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.
- Internal perform step is convenient, even though we're interested only in specifying external behavior.
- Perform step corresponds to serialization point.

Canonical atomic object automaton

- Liveness:
  - One task for each port i.
  - Use the usual I/O automata convention that tasks keep getting turns to take steps.
  - To model the effects of failures, we include a specially $dummy_i$ action in each task i, which gets enabled when $stop_i$ occurs.

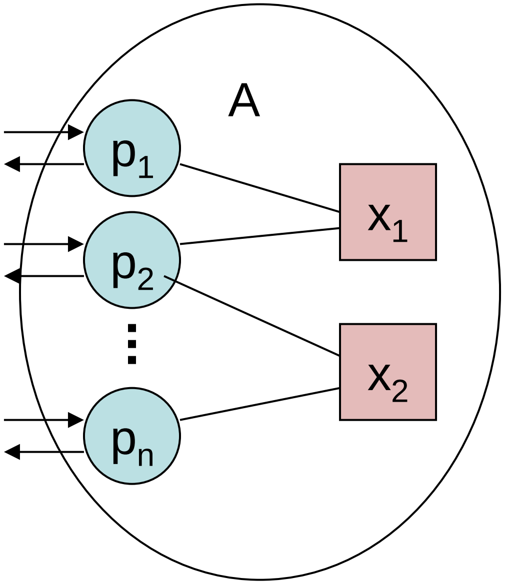
Canonical atomic object automaton

- Equivalent to the original serialization-point-based 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 to prove safety.
- 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

Atomic objects vs. shared vars

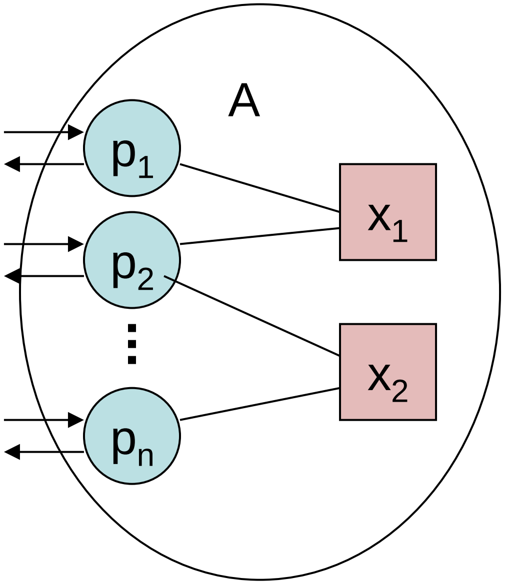
- Atomic objects aren't the same as shared variables.
- But an important basic result says we can substitute atomic objects for shared variables in a shared-memory system, and the resulting system "behaves the same".
- Enables hierarchical construction of shared-memory systems.



- This is not completely obvious, although the research literature just assumes that it works.

Atomic objects vs. shared vars

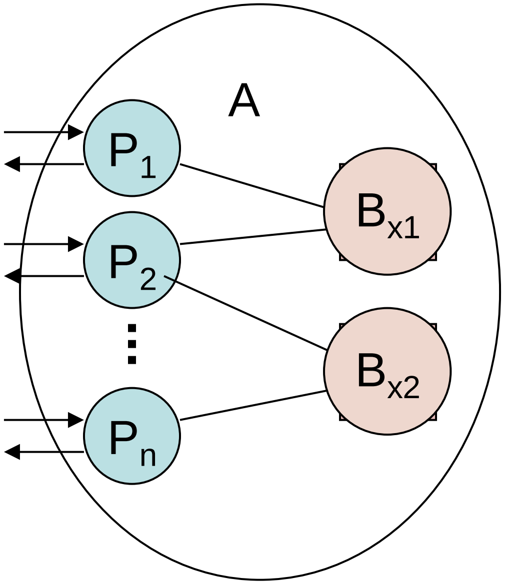
- Atomic objects aren't the same as shared variables.
- But an important basic result says we can substitute atomic objects for shared variables in a shared-memory system, and the resulting system "behaves the same".
- Enables hierarchical construction of shared-memory systems.



- The substitution:
  - Given A, a shared-memory system, and
  - For each shared variable x of A, given an atomic object B_x (same type, interface corresponding to the allowed connections).
  - Trans(A) is a composition of I/O automata, one for each process and one for each shared variable.

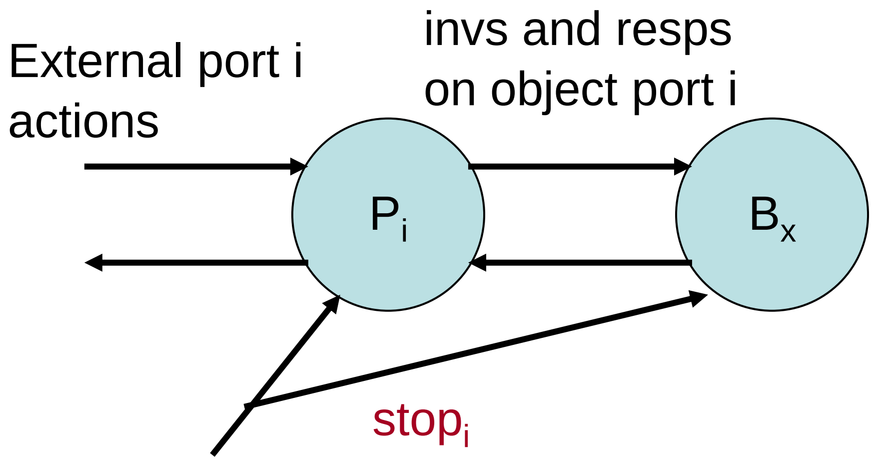
Atomic objects vs. shared vars

- Given shared-memory system A, and for each shared variable x of A, given atomic object B_x.
- Trans(A) is a composition of I/O automata, one for each process and one for each shared variable:
  - For variable x, use atomic object B_x.
  - For process i, use automaton P_i, where:
    - Inputs of P_i are:
      - Inputs of A on port i,
      - Responses of all the B_xs on port i,
      - $stop_i$.
    - Outputs of P_i are:
      - Outputs of A on port i, and
      - Invocations to all the B_xs on port i.
    - Steps of P_i simulate those of process i of A directly, except:
      - When process i of A accesses x, then P_i invokes the operation on B_x and pauses, waiting for a response. When a response arrives, P_i resumes simulating process i.
      - When $stop_i$ occurs, all tasks of P_i (which correpond to tasks of process i in A) are disabled.



Atomic objects vs. shared vars

- A note on failure actions:
  - $stop_i$ is an input both to P_i, and to all objects B_x that P_i is connected to.



What is preserved by this transformation?

- Theorem: For any execution α of Trans(A) $\times$ U, there is an execution α' of A $\times$ U (that is, of the original shared-memory system) such that:
  - $\alpha | U = \alpha' | U$ (looks the same to the users), and
  - $stop_i$ events occur for the same i in α and α' (the same processes fail).
- Technicality: Need a little assumption about A: At any point, for each i, we don't have both process i and the user at i enabled to perform locally-controlled actions.
- Proof: Given α, construct α':
  - Introduce serialization points and responses for operations of B_x in α, as guaranteed by the atomicity definition.
  - Then commute the invocation and responses events with other events until they appear next to their serialization points.

What is preserved?

- Theorem: For any execution α of Trans(A) $\times$ U, there is an execution α' of A $\times$ U such that:
  - α | U = α' | U and
  - $stop_I$ events occur for the same i in α and α'.
- Proof: Given α, construct α':
  - Add serialization points, responses for operations of B_x.
  - Commute invocation and responses events with other events until they appear next to their serialization points.
  - OK as far as the B_xs are concerned.
  - What about the P_is? We aren't allowed to reorder events of the same P_i.
  - But no such reordering happens, because:
    - P_i pauses when it performs invocations, and
    - No inputs arrive at P_i from U while P_i is waiting for a response to an inv (follows from the technical assumption---it's the system's turn)
  - Result is still an execution of Trans(A) $\times$ U (using composition results), but now all invocations and responses occurring in consecutive pairs.
  - Now replace each pair with a single access step.

Liveness

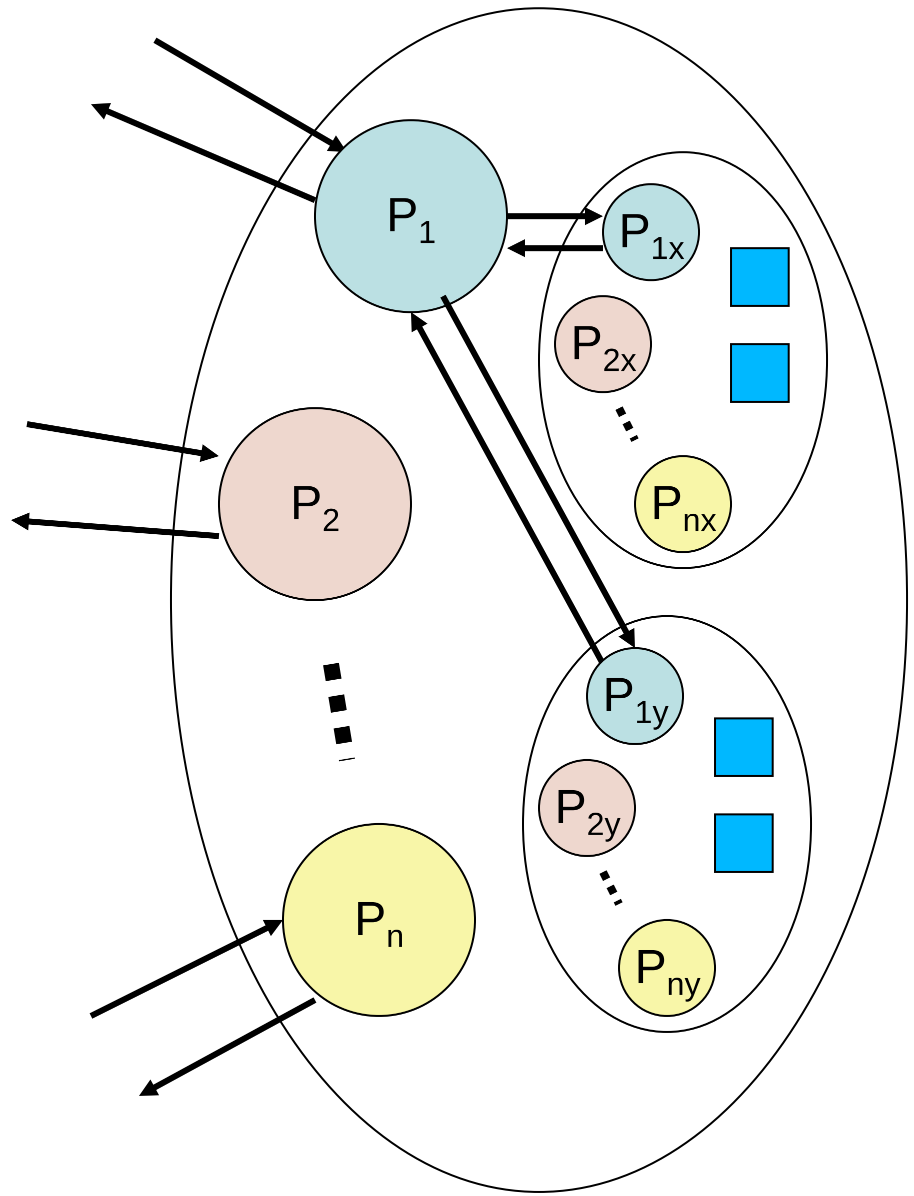
- Construction also preserves some liveness properties:
- We want to show that if α is fair then α' is fair, i.e., any fair execution of Trans(A) $\times$ U emulates a fair execution of A $\times$ U.
- A difficulty: Objects sometimes don't respond to invocations, whereas shared variable accesses always return. So the objects could possibly introduce new blocking.
- We need an assumption that implies that the objects don't introduce new blocking.
- E.g., we could assume that the B_x objects are wait-free.
- E.g., we could assume that at most f failures occur in α and each B_x guarantees f-failure termination (Theorem 13.7).
  - "The failures that happen are tolerated by the objects."
  - That's good enough to ensure that the objects always respond to non-failed processes.

Application 1 of Trans results

- Implementing atomic objects using other atomic objects:
  - Suppose A is an atomic object implementation, using shared memory.
  - Say A and all the B_xs guarantee f-failure termination.
  - Then Trans(A) also implements an atomic object (of the same type), and guarantees f-failure termination.
  - See Corollary 13.9, p. 417, for details.

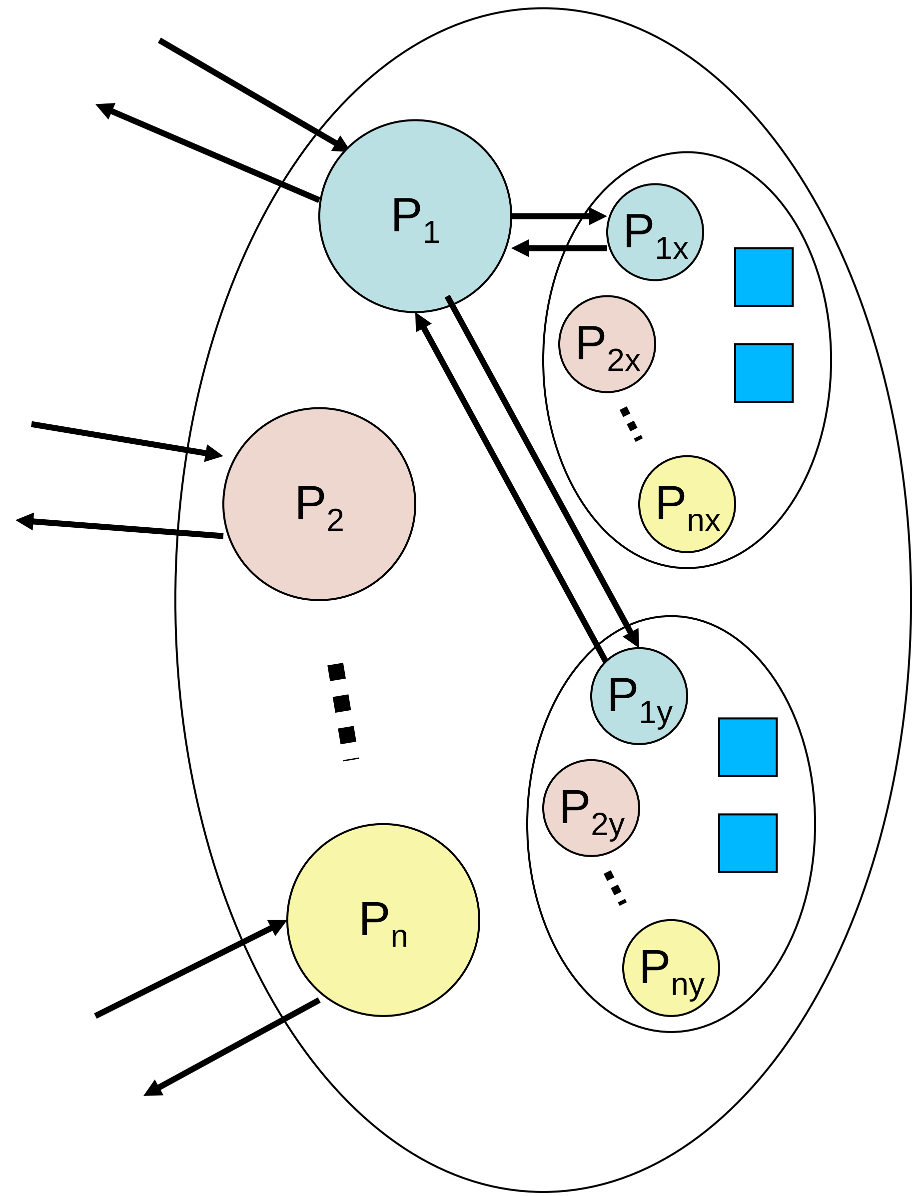
Application 2 of Trans results

- Building shared-memory systems hierarchically.
  - Suppose the B_xs are themselves shared-memory systems.
  - Then Trans(A) yields a 2-level system:
  - If we compose each P_i at the top level with all the i-port agent processes within the B_x implementations, we get an actual shared-memory system (consisting of individual processes and variables).



Combining the two applications

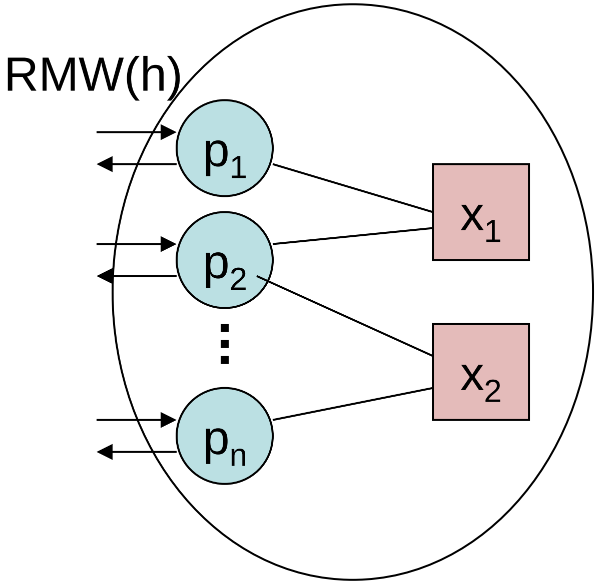
- Building shared-memory implementations of atomic objects hierarchically.
  - Like Application 2, but where:
    - Low-level systems implement atomic objects B_x, and
    - High-level system implements an atomic object, as in Application 1.
  - Shows how to combine shared-memory implementations of atomic objects at two levels to get a single shared-memory implementation of the high-level atomic object.
  - Used implicitly throughout the research literature.



Algorithms to implement RMW atomic objects

Read-Modify-Write Atomic Object

- Can we implement a general RMW atomic object using just read/write shared variables?
- Non-fault-tolerant implementation:
  - Implement the RMW variable using a single shared read/write register.
  - Access the register only within a critical region, using two operations, a read followed by a write.
  - To implement the critical region, use a lockout-free mutual exclusion algorithm, e.g., one of Peterson's.
- Q: Fault-tolerant implementation?



Read-Modify-Write Atomic Object

- Fault-tolerant implementation?
- Say,1-failure termination.
- Theorem: There is no shared memory system using only read/write shared variables that implements a general RMW atomic object and guarantees 1-failure termination.
- Proof: By contradiction.
  - Suppose there is, system B.
  - Let A be any agreement algorithm that uses shared RMW variables and guarantees 1-failure termination.
    - Earlier, we saw how to guarantee even wait-free termination.
  - Substitute B for each of the RMW shared variables in A.
  - The resulting system solves agreement in the read/write shared-memory model, with 1-failure termination.
  - Contradicts impossibility result for consensus in the read/write shared-memory model.

Next time:

- More algorithms to implement atomic objects:
  - Atomic snapshots
  - Atomic read/write registers
- Reading: Sections 13.3-13.4