# 6.852: Distributed Algorithms
Fall, 2009

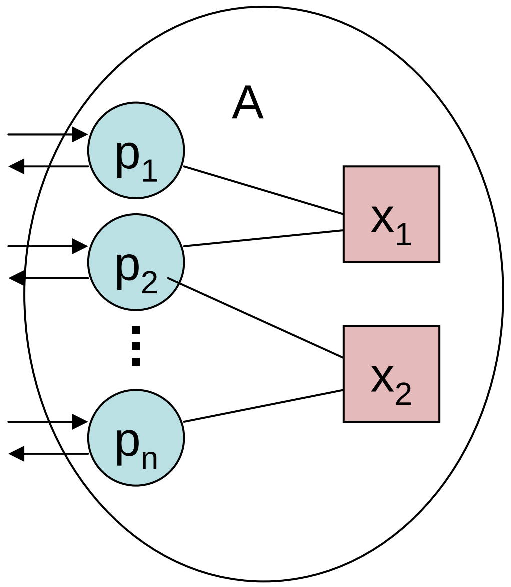
Class 17

## Today’s plan

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

# Shared memory model

- Single I/O automaton with processes and variables "inside".
  - Separation expressed by locality restrictions on the actions and transitions.
  - Processes and variables aren't separate automata.
  - Doesn't exploit I/O automaton (de)composition.
  - Can't talk about implementing shared variables with lower-level distributed algorithms.
- Q: Could 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 to "look like" an instantaneous access.

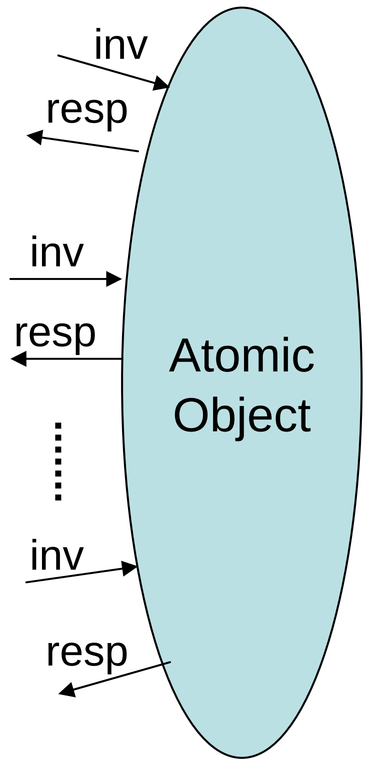


- Define atomic objects:
  - Interface has invocation inputs and response outputs.
  - Invocation/response behavior "looks like" that of an instantaneous-access shared variable.
  - AKA linearizable objects [Herlihy, Wing]

# Atomic objects

- Interface has invocation inputs and response outputs.
- Invocation/response behavior “looks like” that of an instantaneous-access shared variable.

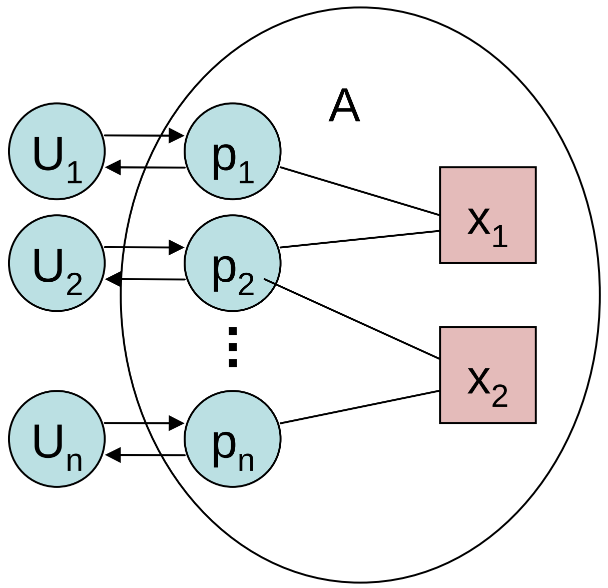
- Atomic object of a given type is similar to an ordinary shared variable of that type, but it allows concurrent accesses by different processes.
- Still looks “as if” operations occur one at a time, sequentially, in some order consistent with order of invocations and responses.
- Fault-tolerance conditions,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

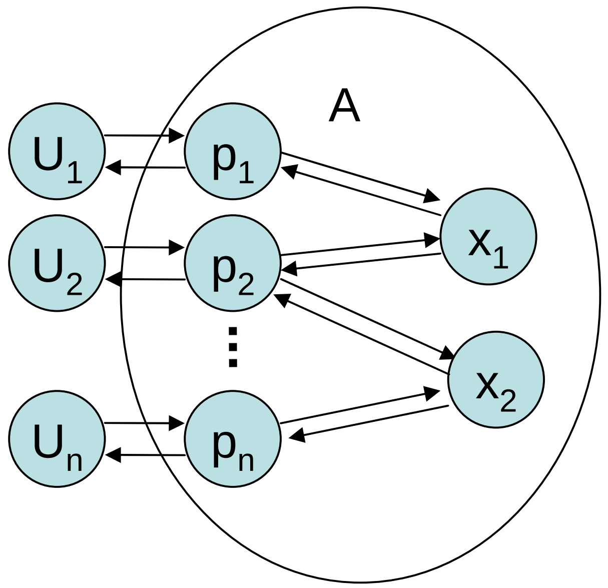


- Separating invocations and responses allows us to consider lower-level implementations of these objects.
  - Shared-memory algorithms, or distributed network algorithms.
  - For shared memory algorithms, can develop algorithms hierarchically, using several levels.
- Atomic objects are important building blocks for multiprocessor systems and distributed systems.

# Replacing variables with atomic objects

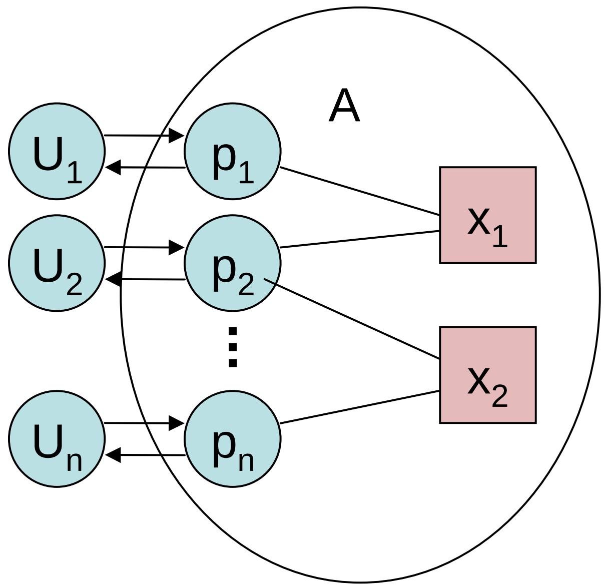
- Now processes and objects are all I/O automata, combined using ordinary automata composition.
- Interactions:
  - Processes access atomic objects via invocations, get responses.
  - Invocations are outputs of processes, inputs of objects.
  - Responses are outputs of objects, inputs of processes.
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

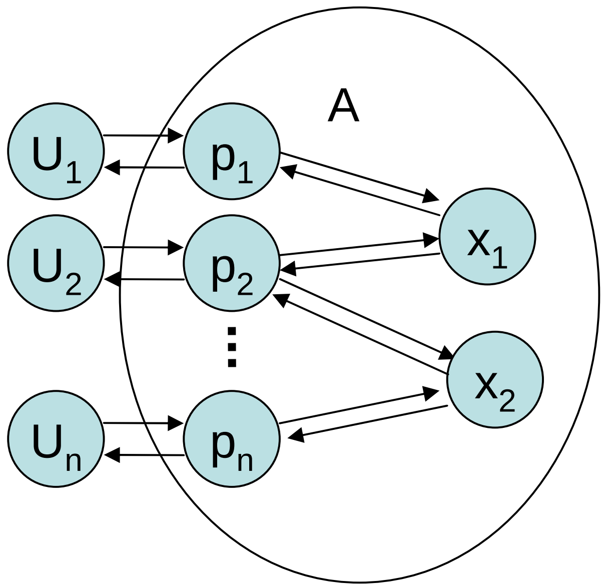




## Replacing variables with atomic objects

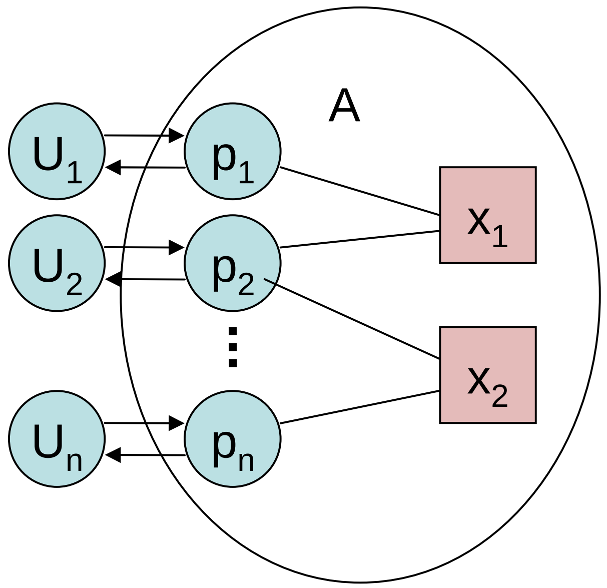
- "Locality" is now automatic from I/O automata composition.
- More complicated than shared variables:
  - More actions (invocations/responses instead of entire accesses).
  - Algorithms have more steps, more bookkeeping.
  - More stuff to reason about.
- More realistic system model.

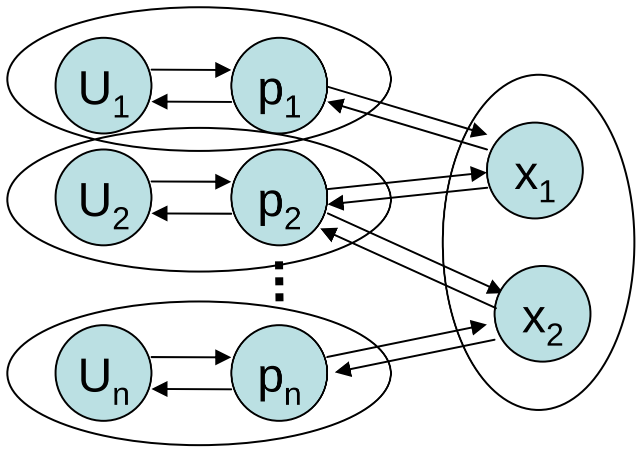




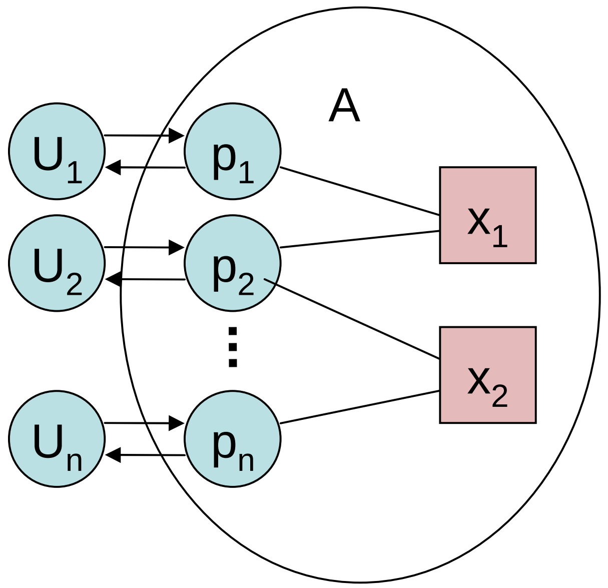
## Atomic objects

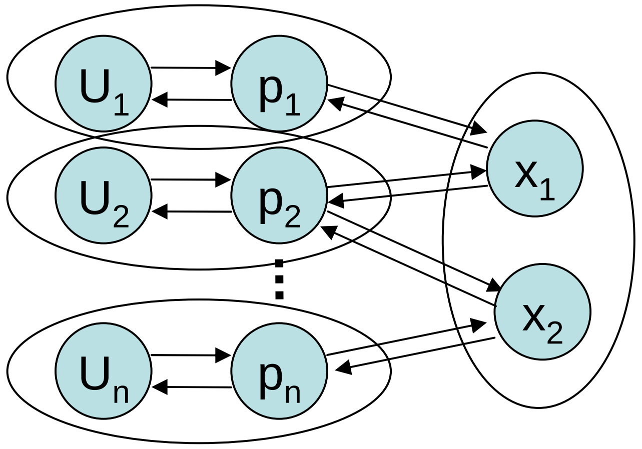
- Replace variables with atomic objects
  - can decompose system in different ways
    - what a process is depends on your point of view
    - can compose objects into larger objects





- but we need some restrictions to get “equivalance”
- handling failures, in particular, is tricky
  - delay for later in lecture





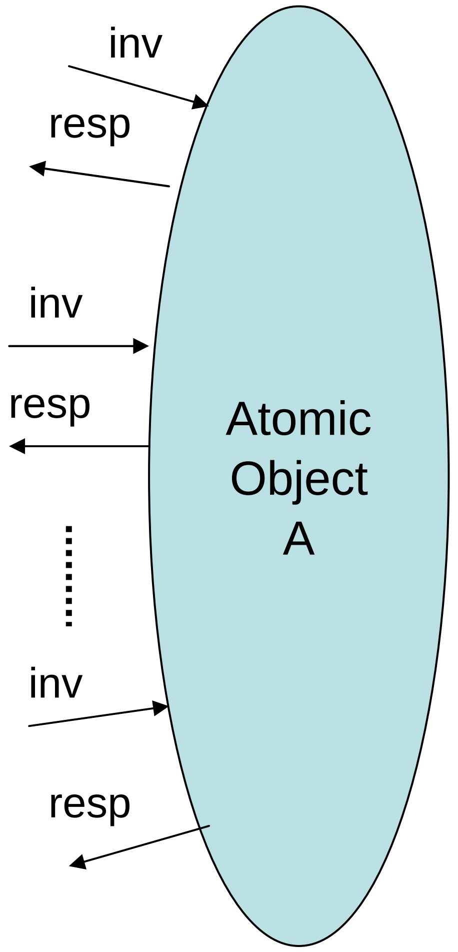
## Atomic objects:
## Basic definitions

# Atomic object definitions

- Variable type: $(V, v_0, \text{invs}, \text{resps}, f)$
  - V: Set of values
  - $v_0$: Initial value
  - invs: Set of invocations
  - resps: Set of responses
  - $f: \text{invs} \times V \rightarrow \text{resps} \times V$
    - Describes responses to an invocation and associated changes to the variable.
- AKA Sequential specification [Herlihy], State machine [Lamport]
- 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, but no variable values)

# Atomic objects

- Atomic object A of a given 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 some of the ports, not all.
  - 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 invocation).



## Conditions satisfied by A

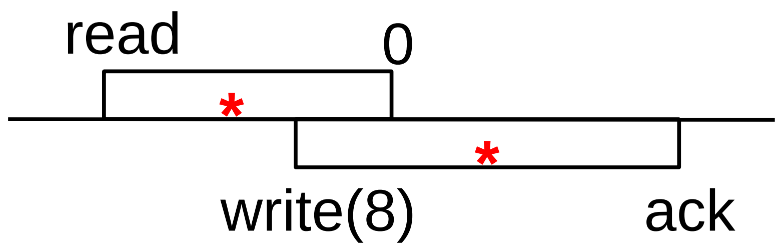
- Preserves well-formedness (alternating invocations and responses at each port, starting with invocation).
- Atomicity:
  - First define when a well-formed sequence $\beta$ of invocations and responses (at all ports) is atomic.
  - Then A satisfies atomicity iff all well-formed executions of $A \times U$, where $U = \Pi U_i$ (for any users) have atomic traces.
- First suppose that all invocations have matching responses (that is, the sequence $\beta$ is complete).
- Then we say $\beta$ 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.

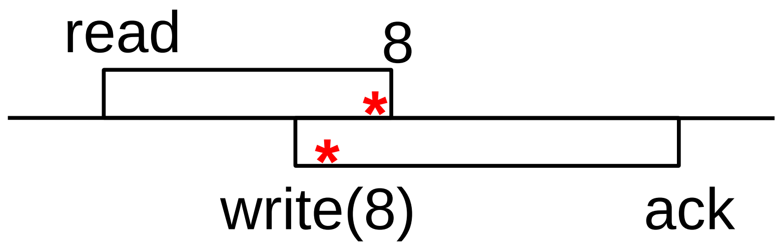
# Atomicity for complete sequences

- 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 $\beta$ is a complete well-formed sequence of invocations and responses. Then $\beta$ 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 $\beta$ 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.
  - Select some subset $\Phi$ of incomplete operations (arbitrary).
  - 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 invs and their resps to the serialization points yields a trace of the variable type.

# Atomic sequences, in general

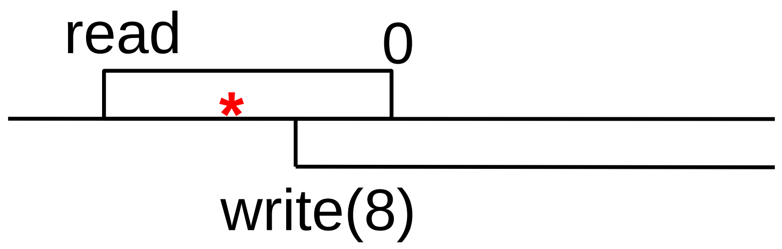
- Suppose $\beta$ is any well-formed sequence of invocations and responses.

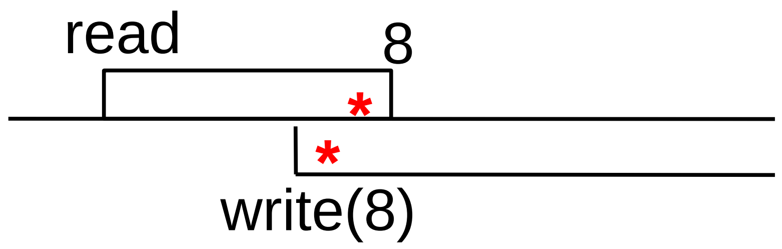
  Then $\beta$ is atomic provided that one can
  - Insert serialization points for all complete operations.
  - Select a subset $\Phi$ of incomplete operations.
  - For each operation in $\Phi$, insert a serialization point somewhere after the invocation, and make up a response.

  In such a way that moving all matched invs and their resps to the serialization points yields a trace of the variable type.

# More atomicity examples

- Initial value 0.





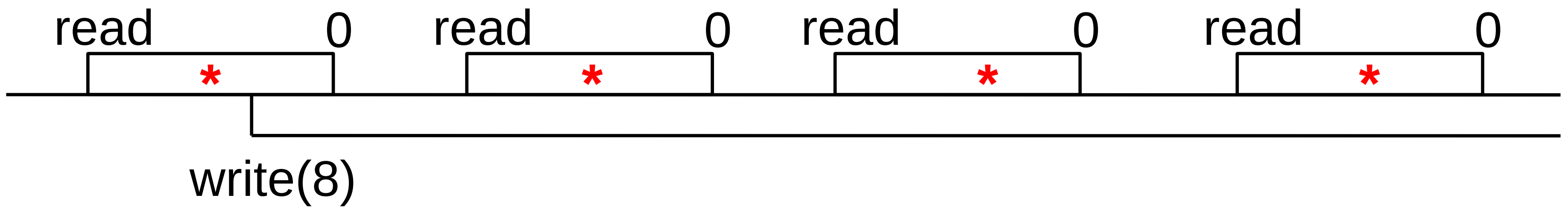
- read, 0 is correct for serial specification.
- write(8), ack, read, 8 is correct.

# Atomic objects

- Define acceptable behavior using trace properties
  - well-formedness (for port i)
    - alternating invocation/response (beginning with invocation) for i
    - whole trace is well-formed if well-formed for every port
  - sequential
    - alternating invocation/response for whole trace
    - trace for the variable type
  - complete
    - every invocation has matching response
      - invocation+matching response = complete operation
      - invocation without matching response = incomplete/pending operation

# Another atomicity example

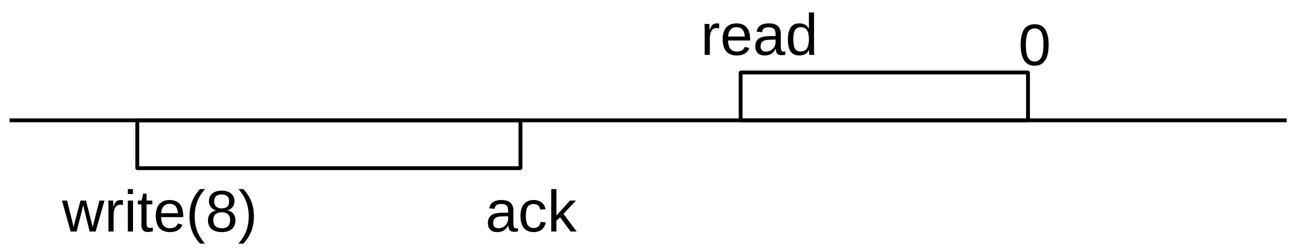
- Initial value 0.



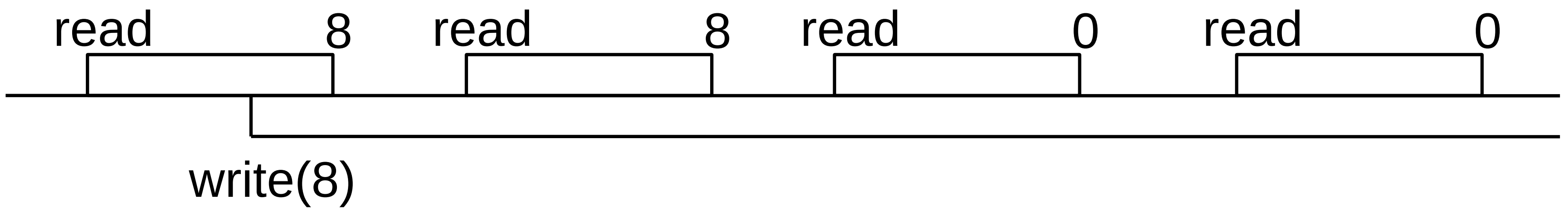
- read, 0, read, 0,…(forever) is correct.
- The write does not (cannot) get a serialization point.

# 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 $\beta$ 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: A 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 ops 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

- $\text{increment}_i$: Increment x(i).
- $\text{read}_i$: 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 increment: At the actual write step.
    - For complete read:
      - Must be somewhere between invocation and response.
      - Returns a value v such that $v \geq$ the sum of the x's at the beginning, but $v \leq$ the sum at the end.
      - Since the sum increases by one at a time, there is some point where sum of the x's = v.
      - Put the serialization point there.
    - For incomplete read: Don't bother.
- Correctness depends on restricted form of the operations.

## Canonical Atomic Object Automata

# Canonical atomic object automaton

- Describe 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 (see 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 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.

## Canonical atomic object automaton

- An equivalent definition as an automaton C
  - external actions as before
  - internal actions: perform(a,i)
  - state variables:
    - val: V, initially $v_0$
    - inv_buffer: set of (i,a), initially empty
    - resp_buffer: set of (i,b), initially empty
  - transitions:
    - inv(a,i) adds (i,a) to inv_buffer
    - perform(a,i) removes (i,a) from inv_buffer, applies a to val, and puts (i,b) into resp_buffer, where b is the response from applying a to val
    - resp(b,i) takes (i,b) removes resp_buffer
  - one task for each i

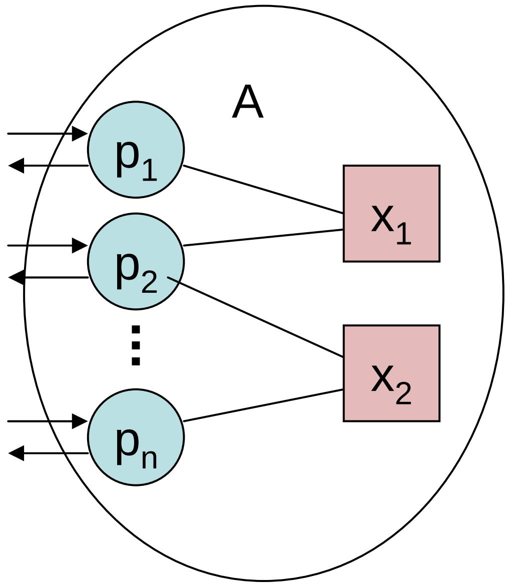
# Canonical atomic object automaton

- For C and U as defined previously:
  - ® □ traces(C×U) iff ® is well-formed and atomic
  - ® □ fairtraces(C×U) iff ® is well-formed, atomic and complete
- Proof
  - well-formedness
  - atomicity
  - completeness
  - need to show both directions
- An automaton A is atomic if it implements C.

## Atomic Objects
## vs.
## Shared Variables

# Atomic objects vs. shared vars

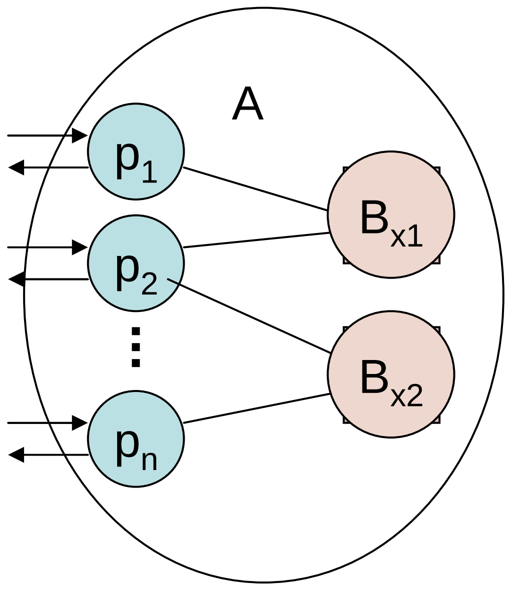
- Atomic objects aren’t the same as shared variables.
- But an important basic result says we can substitute atomic objects for shared variables in a shared-memory system, and the resulting system “behaves the same”.
- Enables hierarchical construction of shared-memory systems.



- 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 is the composition of I/O automata, one for each process and variable.

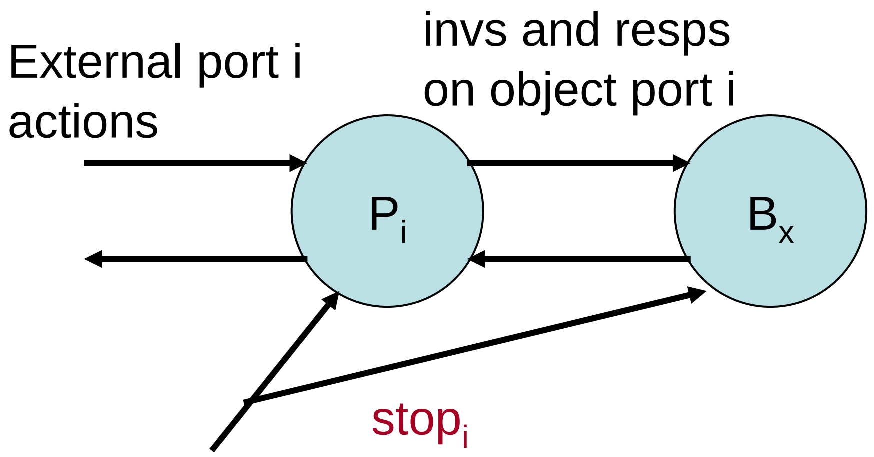
# Atomic objects vs. shared vars

- Given shared-memory system A, and for each shared variable x of A, given atomic object $B_x$.
- Trans is the composition of I/O automata, one for each process and 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 $B_x$s on port i, and $stop_i$.
    - Outputs of $P_i$ are outputs of A on port i and invocations to all the $B_x$s 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 $B_x$, then blocks, waiting for a response. When response arrives, $P_i$ resumes simulating process i.



# 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 $\alpha$ of Trans $\times$ U, there is an execution $\alpha'$ of A $\times$ U (that is, of the original shared-memory system) such that:
  - $\alpha \mid U = \alpha' \mid U$ (looks the same to the users), and
  - $stop_i$ events occur for the same i in $\alpha$ and $\alpha'$ (the same processes fail).
- Technicality: Need a little assumption about A---that at any point, for each i, either process i or the user at i is enabled to do something, but not both.
- Proof: Given $\alpha$, construct $\alpha'$:
  - Introduce serialization points and responses for operations of $B_x$ in $\alpha$, 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 $\alpha$ of Trans $\times$ U, there is an execution $\alpha'$ of A $\times$ U such that:
  - $\alpha \mid U = \alpha' \mid U$ and
  - $stop_i$ events occur for the same i in $\alpha$ and $\alpha'$.
- Proof: Given $\alpha$, construct $\alpha'$:
  - Introduce ser. pts. and responses for operations of $B_x$ in $\alpha$.
  - Commute invocation and responses events with other events until they appear next to their serialization points.
  - OK as far as the $B_x$s are concerned.
  - What about the $P_i$s? We aren't allowed to reorder events of the same $P_i$.
  - But no such reordering happens, because:
    - $P_i$ blocks when it performs invocations, and
    - No inputs arrive at $P_i$ from U while $P_i$ is waiting for a response to an invocation (by the technical assumption---it's the system's turn).
  - Result is still an execution of Trans $\times$ U (using composition results), but now it's one with all invocations and responses occurring in consecutive pairs.
  - Now replace the pairs with single access steps.

## Liveness

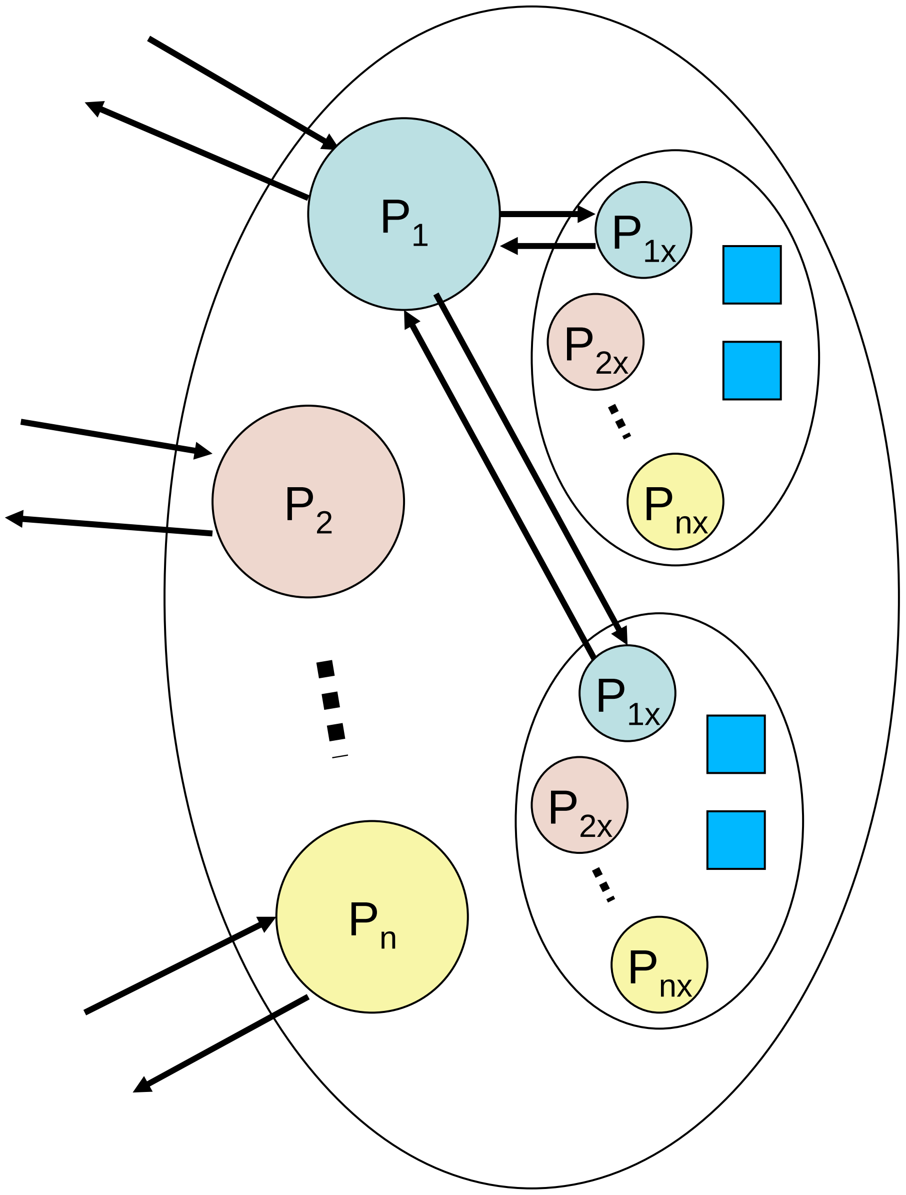
- Construction also preserves liveness:
- Can show that $\alpha$ fair implies $\alpha'$ fair, that is, that a fair execution of Trans $\times$ U emulates a fair execution of A $\times$ U.
- The difficulty: Objects sometimes don't respond to invocations, whereas shared variable accesses always return. So the objects could introduce new blocking.
- We need an assumption that implies that the objects don't introduce new blocking.
- E.g., can assume that the $B_x$ objects are wait-free.
- E.g., can assume that at most f failures occur in $\alpha$ and each $B_x$ guarantees f-failure termination.
  - "The failures that happen are tolerated by the objects."
  - Ensures that the objects always respond to non-failed processes.

# Application 1 of Trans results

- Implementing atomic objects using other atomic objects:
  - Suppose A is itself an atomic object implementation, using shared memory.
  - Say A and all the $B_x$s guarantee f-failure termination.
  - Then Trans also implements an atomic object (of the same type), and guarantees f-failure termination.

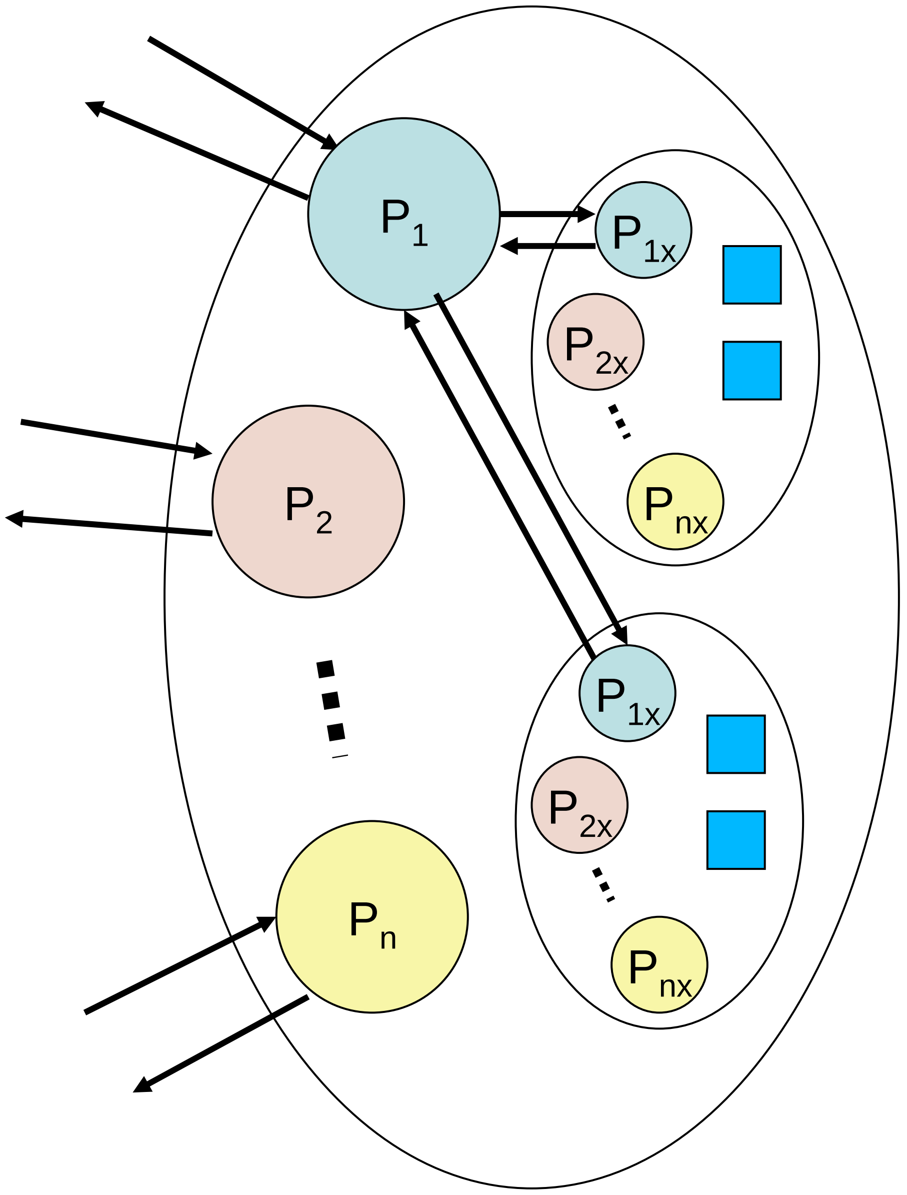
# Application 2 of Trans results

- Building shared-memory systems hierarchically.
  - Suppose the $B_x$s are themselves shared-memory systems implementing atomic objects.
  - Then Trans 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 (processes and variables).



# Combining the two applications

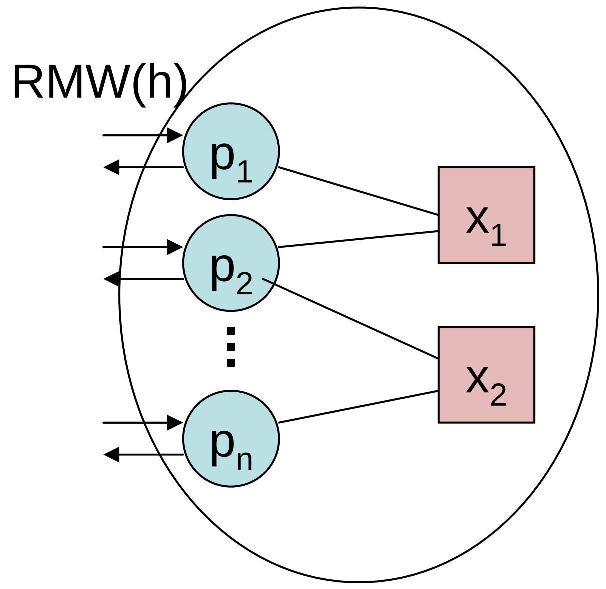
- Building shared-memory implementations of atomic objects hierarchically.
  - Same as Application 2, but top level system is itself an atomic object implementation, as in Application 1.
  - Shows how to combine shared-memory implementations of atomic objects at two levels to get a single shared-memory implementation of the top-level atomic object.
  - Used implicitly in the research literature.



## Algorithms to implement 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:
  - Use lockout-free mutex algorithm, e.g., one of Peterson's.
  - Simulate the RMW variable using a read/write register.
  - Access the register only within critical region, using a read followed by a write.
- 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 a RMW-based agreement algorithm that uses 1 shared RMW variable and guarantees 1-failure termination.
    - Earlier, we saw how to guarantee wait-free termination.
  - Substitute B for the RMW shared variables in A.
  - Resulting system solves agreement in read/write model, with 1-failure termination.
  - Contradicts impossibility result for consensus.

## Next time:

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