6.852 Lecture 17

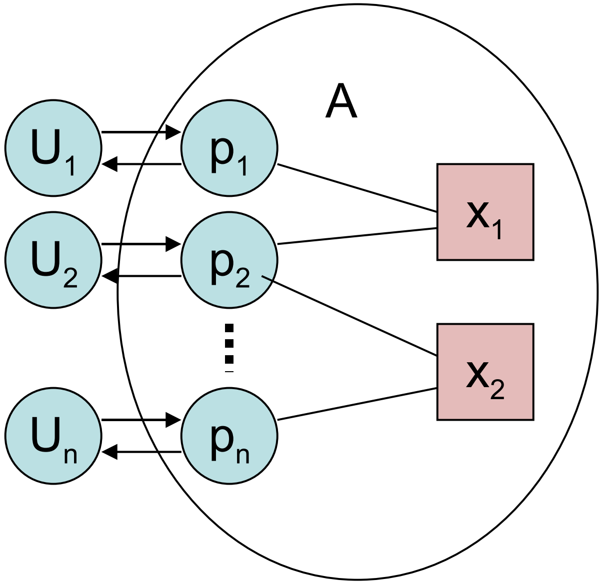
- Atomic objects
- Reading: Chapter 13
- Next lecture: Atomic snapshot, read/write register

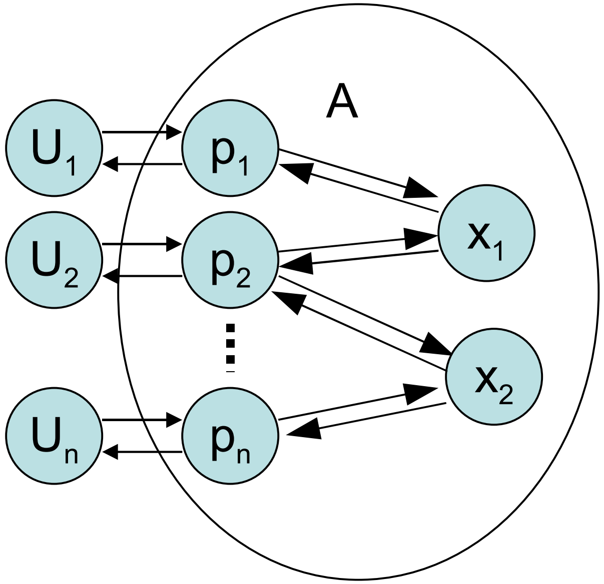
Shared-memory model

- Single I/O automaton with “locality restrictions”
  - doesn't exploit I/O automaton (de)composition for locality
  - can't abstract implementation of a single shared variable
- “More natural” to model each process and variable as separate I/O automata (Chapter 9.1)
  - split operations on variables into invocation/response actions
    - but how to get “instantaneous” access?
      - restrict executions to ones in which inv/resp are consecutive?
      - special automaton (with new composition operation) for variables?
- Atomic objects
  - allow split, and define behavior; require “atomicity”
  - “looks like” instantaneous-access shared variables

Atomic objects

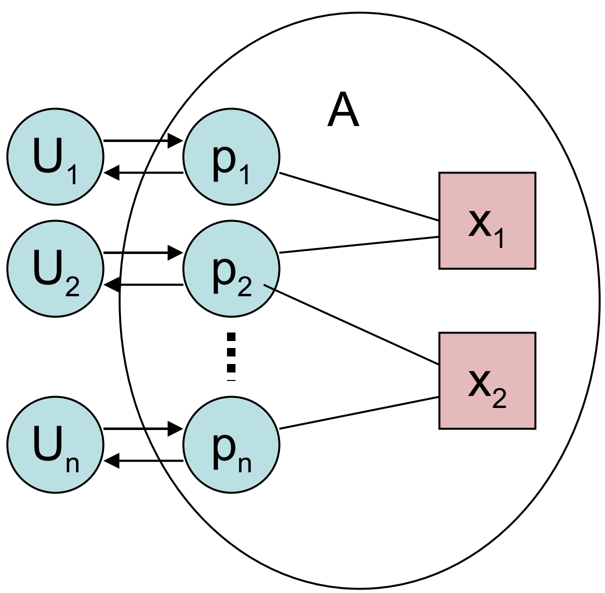
- Replace variables with atomic objects
  - everything is an I/O automaton: normal composition
  - processes access atomic objects via invocations; get responses
    - may be a gap between invocation and response: what is allowed?

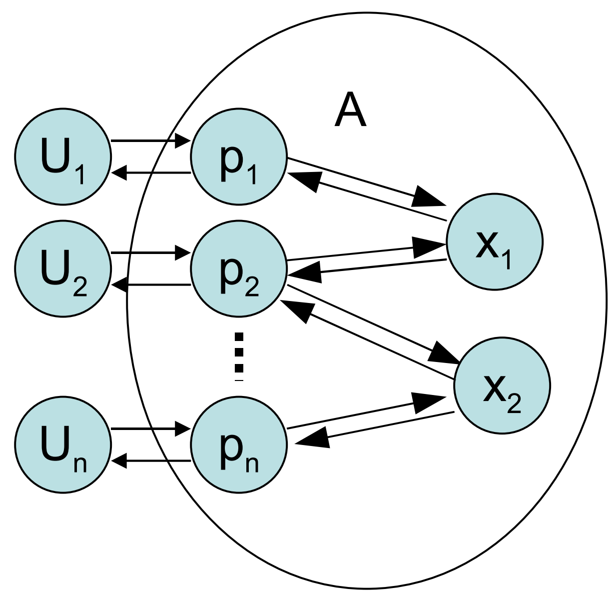




Atomic objects

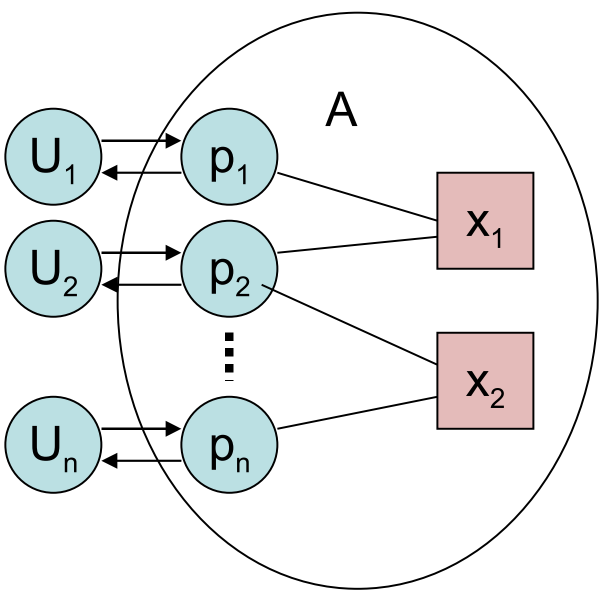
- Replace variables with atomic objects
  - more actions (invocations/responses)
  - more bookkeeping (to track invocations/responses)/state
  - more stuff to reason about

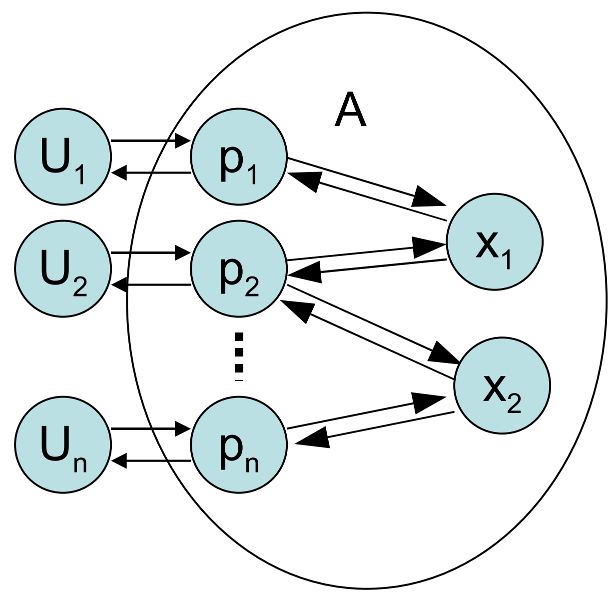




Atomic objects

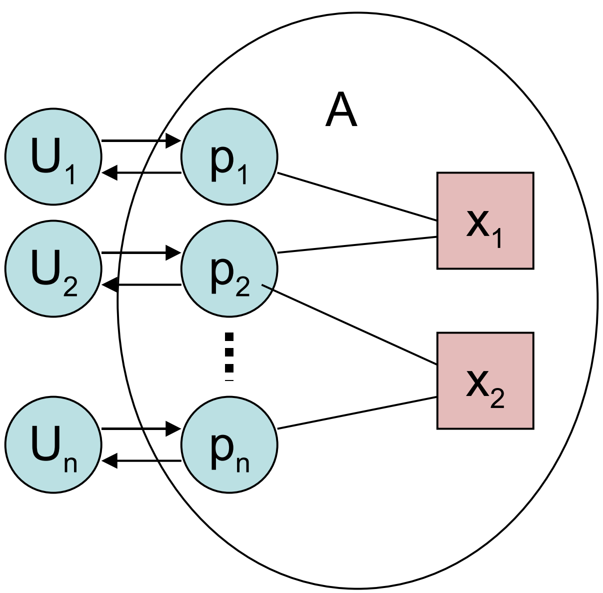
- Replace variables with atomic objects
  - "locality" immediate from I/O automaton composition
  - encapsulate complex "variable" implementations
  - enable hierarchical proofs (and other I/O automata theory)
  - more faithful model of system (but same observable behavior)

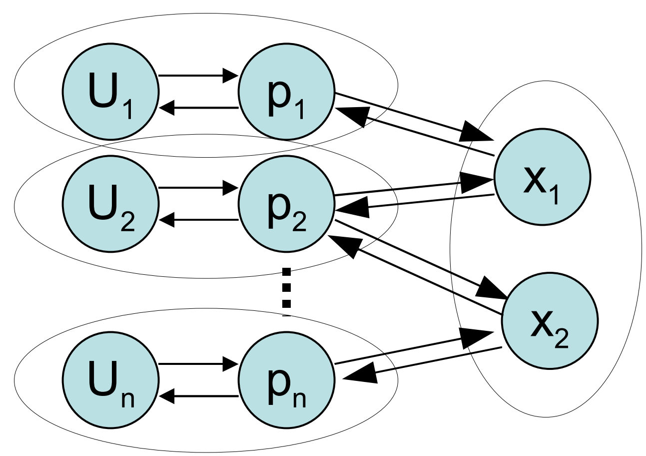




Atomic objects

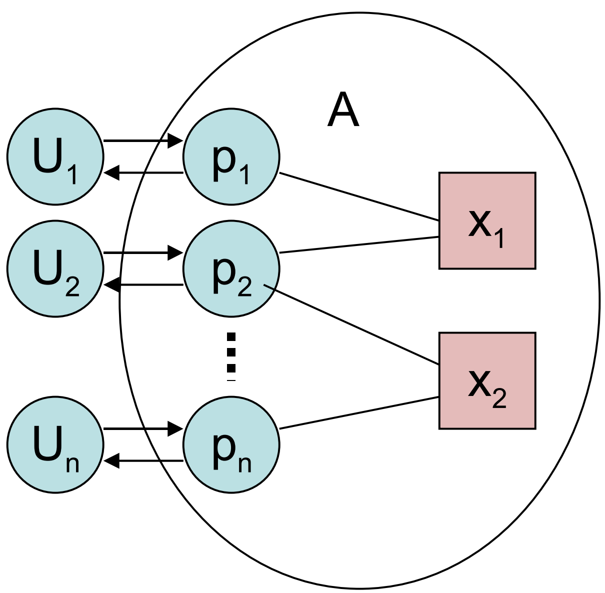
- Replace variables with atomic objects
  - can decompose system in different ways
    - what a process is depends on your point of view
    - can compose objects into larger objects

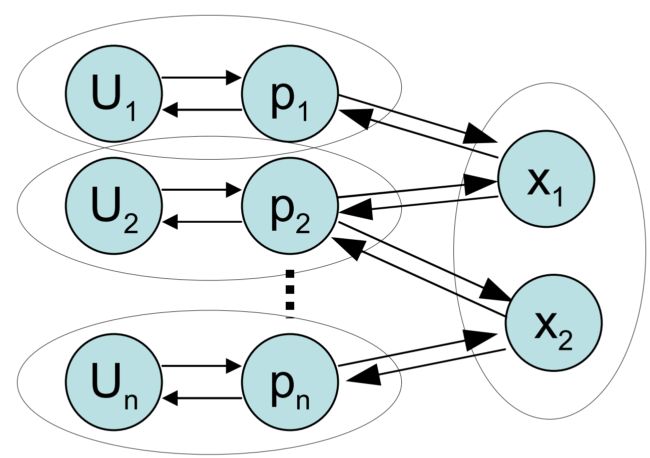




Atomic objects

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





Atomic objects

- Variable type
  - V: set of values
  - v_0: initial value
  - invs: set of invocations
  - resps: set of responses
  - f: invs × V → resps × V
  - 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,...
    - v_i is value; a_i is invocation; b_i is response
    - ends in value if finite
    - $(b_i, v_i) = f(a_1, v_{i-1})$ for $i > 0$
  - trace: a_1, b_1, a_2, b_2, a_3, b_3, a_4, b_4,... (i.e., drop values)

Atomic objects

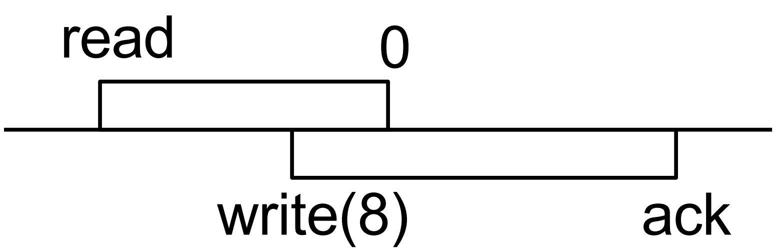
- Shared-memory model: processes and variables
  - state consists of processes' local states plus values of variables
  - each action associated with one process, possibly one variable
  - if no variable, only local state changes; only based on local state
  - if associated with variable, must be an invocation
    - new value of variable determined by invocation and previous state
    - new local state based on response and local state
- Atomic objects:
  - assume “ports” 1, 2, ..., n (one for each “process”)
  - external actions for invocation and response: inv(a,i), resp(b,i)
  - but operation should appear to occur atomically

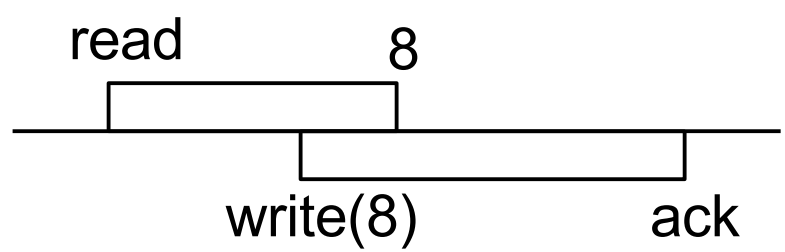
Atomic objects

- 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
  - atomic

Atomicity property

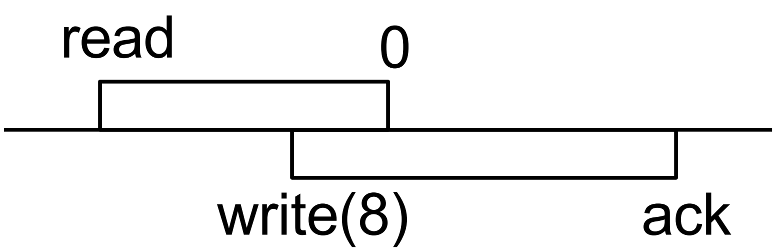
- Atomicity for complete traces
  - each operation has a **serialization point**
    - operation "really happens" at its serialization point
    - between its invocation and response
  - a trace is **atomic** if sequence resulting from moving every invocation/response pair to its serialization point is sequential

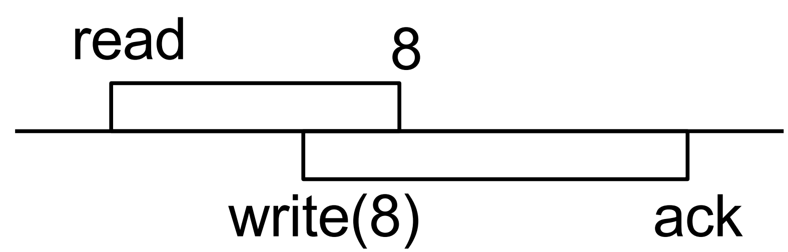




Atomicity property

- Atomicity for complete traces
  - each operation has a **serialization point**
    - operation “really happens” at its serialization point
    - between its invocation and response
  - a trace is **atomic** if sequence resulting from moving every invocation/response pair to its serialization point is sequential

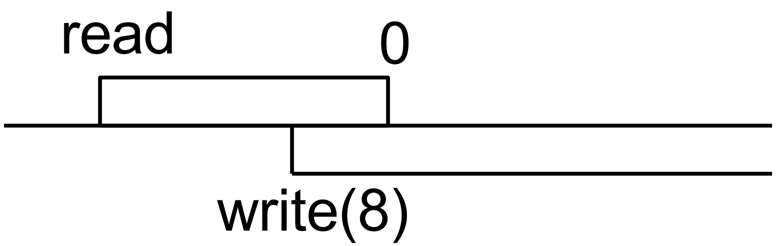


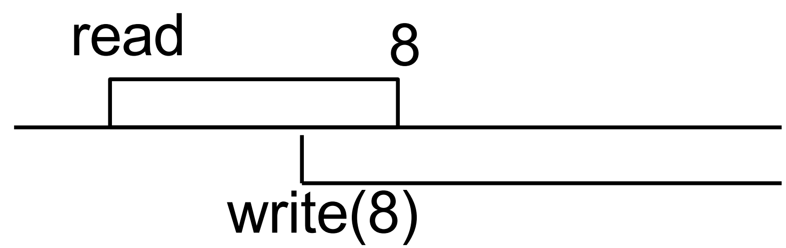


Alternate definition (Herlihy):
A complete trace (history) is atomic if it can be reordered to a sequential history that preserves per-process order and preserves the order of any response followed by an invocation.

Atomicity property

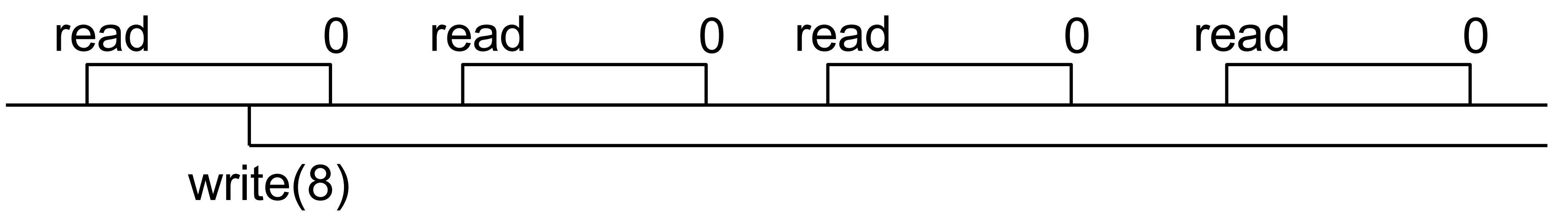
- Atomicity for incomplete traces
  - if there is an atomic "completion" of the trace





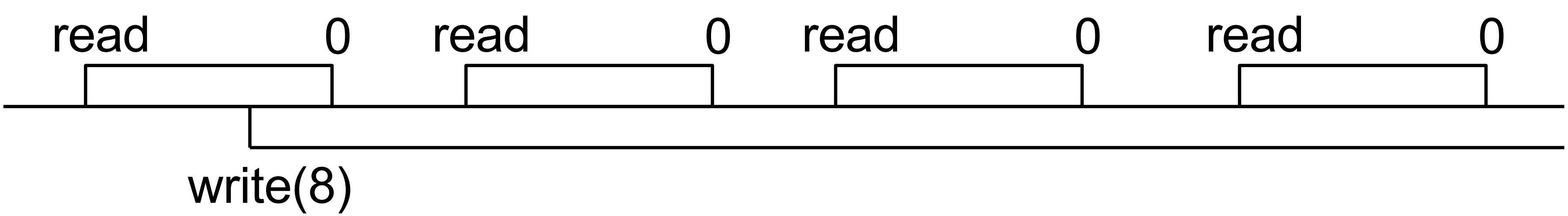
Atomicity property

- Atomicity for incomplete traces
  - if there is an atomic “completion” of the trace
  - but this is not enough!



Atomicity property

- Atomicity for incomplete traces
  - if there is an atomic "completion" of the trace
  - but this is not enough!
    - can drop some (any arbitrary set of) pending operations



Atomicity property

- A trace is **atomic** (for a given variable type) if
  - after dropping some incomplete operations,
  - the remaining incomplete operations can be completed so that
  - there exists a serialization point for each (complete) operation
  - such that if the invocation and response actions are moved to the serialization point (invocation before response),
  - the resulting trace is sequential.
- Let U be the most general well-formed user automaton.
- An automaton A is an atomic object (for a given variable type) if every trace of A×U is well-formed and atomic
  - and in every fair trace, every operation is complete.

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:
  - $\beta \in$ traces(C×U) iff β is well-formed and atomic
  - $\beta \in$ fairtraces(C×U) iff β 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.

Atomicity is a safety property

- Suppose automaton A satisfies the following properties:
  - unique start state
  - unique post-state for any pre-state and enabled action
  - from any reachable state, there are finitely many execution fragments containing only internal actions beginning with that state.
- Then traces(A) is a safety property
  - Proof: Nonemptiness and prefix-closure are obvious. To prove limit-closure, given an infinite sequence β all of whose prefixes are traces of A, construct infinite tree labeling each node with a state and each edge with either an execution fragment with only internal actions or an action in β such that the actions encountered walking from root to any node n yields an extended step from start state (which labels the root) to the state labeling n. By the conditions above, this tree has finite branching. By Konig's Lemma, there is an infinite path in the tree. This must yield β.

Atomic object examples

- Variable type supports read, increment; initial value 0
- Atomic object supports both operations on all ports
  - state variables: x(i) for each i (written only by i, read by all)
  - $increment_i$ increments x(i)
  - $read_i$ reads all x(i) (any order) and returns sum
- Why does this work?

Atomic object examples

- Read/modify/write from read/write and mutex
  - single read/write register (read and written by all)
  - mutex object
- When RMW is invoked, try to get mutex
- When critical,
  - read register
  - do computation locally
  - write back results
  - exit critical section
- Not fault-tolerant
  - it can't be! (why not?)

Atomic objects vs. shared variables

- Replacing shared variables with atomic objects
- For any shared-memory system A, define Trans(A)
  - one automaton P_i for each process i
    - input actions: input actions of A plus responses from the B_x's
    - output actions: output actions of A plus invocations to the B_x's
    - internal actions as before except for access of shared variables
  - one automaton B_x for each shared variable x
    - input actions: invocations of x's variable type
    - output actions: responses of x's variable type
  - replace access of shared variables with invocation, then block
  - must assume users block also (turn variable)
  - traces the same if we hide atomic object invocations/responses

Atomic objects vs. shared variables

- Two ways to show equivalence
  - use canonical automaton: replace perform with variable access, discard invocations and responses
  - use atomicity property: access variable at serialization point, discard invocations and responses
  - in other direction, replace variable access with invocation-response pair.
  - Why do we need the users to block?
- Hierarchical decomposition of shared memory systems
  - atomic object may be implemented by a shared-memory system

Fault-tolerance

- Model stopping faults with $stop_i$ action
  - external action, but not by users
  - disables all tasks of processes
  - input actions for atomic objects
- f-failure termination
  - in any fair execution, if $stop_i$ on at most f ports then every operation on nonfailing port is complete
- I-failure termination (for set of ports I)
  - similar, except guarantee when failures on subset of I
- Generalize these by having a set of sets of ports

Fault-tolerance

- Two important special cases:
  - 1-failure termination
  - wait-free termination (n-failure termination)

Fault-tolerance

- Modifying the canonical automaton to be wait-free
  - add $stop_i$ actions
  - add $dummy_i$ actions for each port i
    - enabled by $stop_i$ action
    - in same task as $perform_i$ and $resp_i$ actions
  - fair executions of modified automaton are wait-free
- Shared variables vs atomic objects
  - no longer get nice trace equivalence because stop actions may need to be moved (why?)