6.852: Distributed Algorithms
Fall, 2009

Class 12

Today's plan

- Weak logical time and vector timestamps
- Consistent global snapshots and stable property detection.
- Applications:
  - Distributed termination.
  - Deadlock detection.
- Asynchronous shared memory model
- Reading: [Mattern], Chapter 19, Chapter 9
- Next:
  - Mutual exclusion
  - Reading: Sections 10.1-10.7

Weak Logical Time and Vector Timestamps

Weak Logical Time

- Logical time imposes a total ordering on events, assigning them values from a totally-ordered set T.
- Sometimes we don't need to order all events---it may be enough to order just the ones that are causally dependent.
- Mattern (also Fidge) developed an alternative notion of logical time based on a partial ordering of events, assigning values from a partially-ordered set P.
- Function ltime from events in α to partially-ordered set P is a weak logical time assignment if:
  1. ltimes are distinct: ltime(e_1) $\neq$ ltime(e_2) if $e_1 \neq e_2$
  2. ltimes of events at each process are monotonically increasing.
  3. ltime(send) < ltime(receive) for same message
  4. For any t, the number of events e with ltime(e) < t is finite.
- Same as for logical time, but using partial order.

Weak logical time

- In fact, Mattern’s partially-ordered set P is designed to represent causality exactly.
- Timestamps of two events are ordered in P if and only if the two events are causally related (related by the causality ordering).
- Might be useful in distributed debugging: A log of local executions with weak logical times could be observed after the fact, used to infer causality relationships among events.

Algorithm for weak logical time

- Based on vector timestamps: vectors of nonnegative integers indexed by processes.
- Each process maintains a local vector clock, called clock.
- When an event occurs at process i, it increments its own component of its clock, which is clock(i), and assigns the new clock to be the vector timestamp of the event.
- Whenever process i sends a message, it attaches the vector timestamp of the send event.
- When i receives a message, it first increases its clock to the component-wise maximum of the existing clock and the incoming vector timestamp. Then it increments its clock(i) as usual, and assigns the new vector clock to the receive event.
- A process' vector clock represents the latest known "tick values" for all processes.
- Partially ordered set P:
  - The vector timestamps, ordered based on ≤ in all components.
  - $V \leq V'$ if and only if $V(i) \leq V'(i)$ for all i.

Key theorems about vector clocks

- Theorem 1: The vector clock assignment is a weak logical time assignment.
- Essentially, if event π causally precedes event π', then the logical times are ordered, in the same order.
- Proof: LTTR.
  - Not too surprising.
  - True for direct causality, use induction on number of direct causality relationships.
- Claim this assignment exactly captures causality:
- Theorem 2: If the vector timestamp V of event π is (component-wise) $\leq$ the vector timestamp V′ of event π', then π causally precedes π'.
- Proof: Prove the contrapositive: Assume π does not causally precede π' and show that V is not $\leq$ V′.

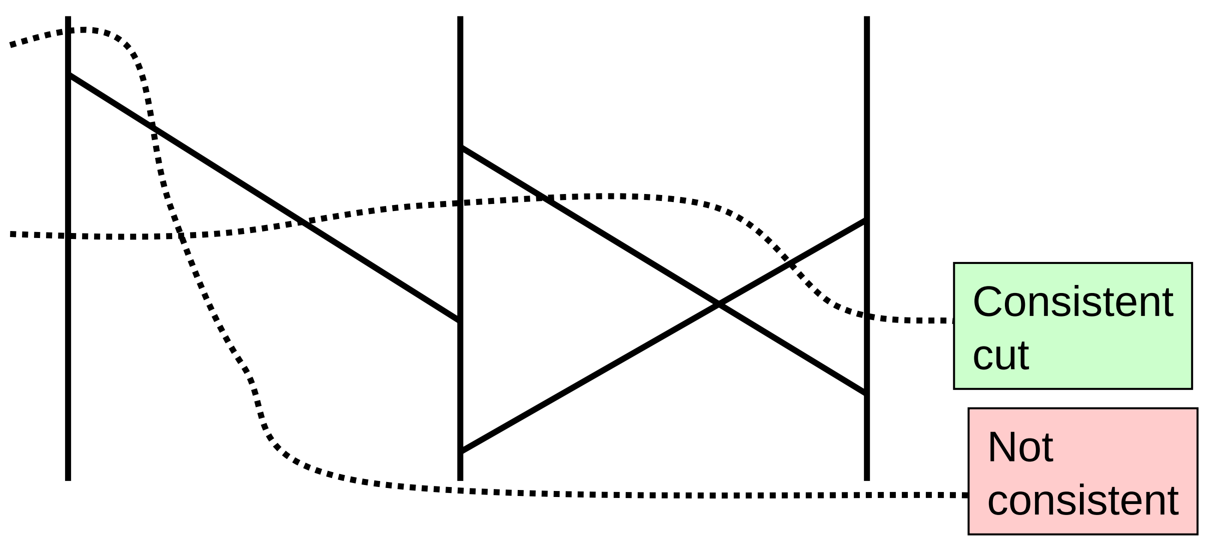
Proof of Theorem 2

- Theorem 2: If the vector timestamp V of event π is (component-wise) $\leq$ the vector timestamp V' of event π', then π causally precedes π'.

- Proof: Prove the contrapositive: Assume π does not causally precede π' and show that V is not $\leq V'$.
  - Assume π does not causally precede π'.
  - Say π is an event of process i, π' of process j.
  - We must have $j \neq i$.
  - i increases its clock(i) for event π, say to value t.
  - Without causality, there is no way for this tick value t for i to propagate to j before π' occurs.
  - So, when π' occurs at process j, j's clock(i) < t.
  - So V is not $\leq V'$.

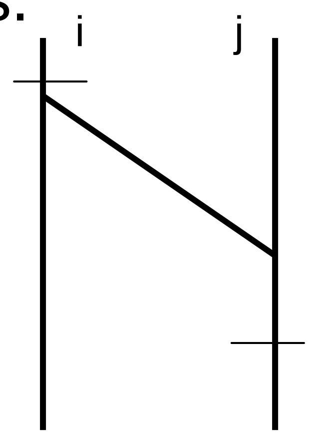
Another theorem about vector timestamps [Mattern]

- Relates timestamps to consistent cuts of causality graph.
- Cut: A point between events at each process.
  - Specify a cut by a vector giving the number of preceding steps at each node.
- Consistent cut: "Closed under causality": If event π causally precedes event π' and π' is before the cut, then so is π.
- Example:



The theorem

- Consider any particular cut.
- Let V_i be the vector clock of process i at i's cut-point.
- Then $V = \max(V_1, V_2, \ldots, V_n)$ gives the maximum information obtainable by combining everyone knowledge from their cut-points.
  - Component-wise max.
- Theorem 3: The cut is consistent iff, for every i, $V(i) = V_i(i)$.
- That is, the maximum information about i that is known by anyone at the cut is the same as what i knows about itself at its cut point.
- "No one else knows more about i than i itself knows."



- Rules out j receiving a message before its cut point that i sent after its cut point; then j would have more info about i than i had about itself.

The theorem

- Let V_i be the vector clock of process i exactly at i's cut-point, $V = \max(V_1, V_2, \ldots, V_n)$.
- Theorem 3: The cut is consistent iff, for every i, $V(i) = V_i(i)$.
- Stated slightly differently:
- Theorem 3: The cut is consistent iff, for every i and j, $V_j(i) \leq V_i(i)$.

- Q: What is this good for?

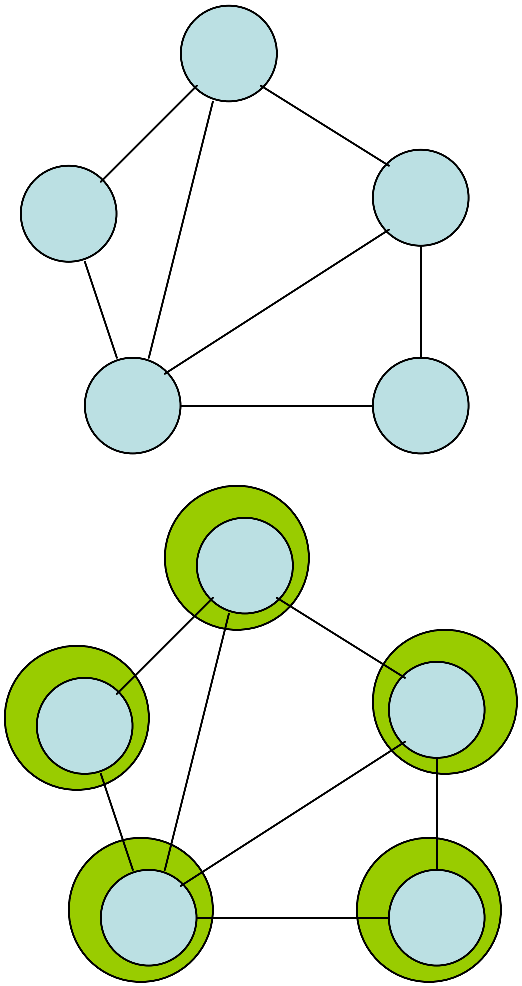
Application: Debugging

- Theorem 3: The cut is consistent iff $V_j(i) \leq V_i(i)$ for every i and j.
- Example: Debugging
  - Each node keeps a log of its local execution, with vector timestamps for all events.
  - Collect information, find a cut for which $V_j(i) \leq V_i(i)$ for every i and j. (Mattern gives an algorithm…)
  - By Theorem 3, this is a consistent cut.
  - Such a cut yields states for all processes and info about messages sent and not received.
  - Put this together, get a “consistent” global state (we will study this next).
  - Use this to check correctness properties for the execution, e.g., invariants.

Consistent Global Snapshots and Stable Property Detection

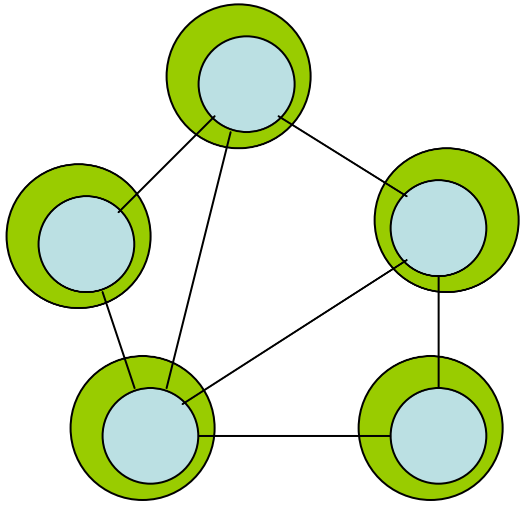
Consistent global snapshots and Stable property detection

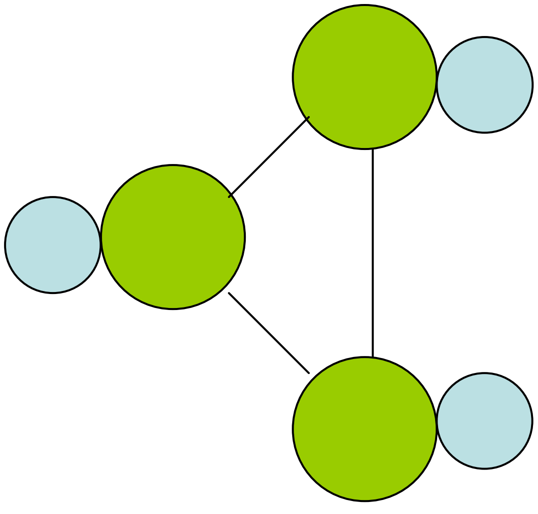
- We have seen how logical time can be used to take a “global snapshot” of a running distributed system.
- Now examine global snapshots more closely.
- General idea:
  – Start with a distributed algorithm A, on an undirected graph G = (V,E).
  – Monitor A as it runs, and determine some property of its execution, e.g.:
    - Check whether certain invariants are true.
    - Check for termination, deadlock.
    - Compute some function of the global state, e.g., the total amount of money in a banking system.
    - Produce a complete snapshot for a backup.
- Monitored version: Mon(A)



Mon(A)

- "Transformed version" of A.
- Mon(A) generally not obtained simply by composing each process A_i with a new monitor process.
- More tightly coupled.
- Monitoring process, $Mon(A)_i$, may "look inside" the corresponding A process, A_i, see the state.
- Superposition [Chandy, Misra]
  - Formalizes the permissible kinds of modifications.
  - Add new state components, new actions.
  - Modify old transitions, but only in certain permissible (nonintrusive) ways.

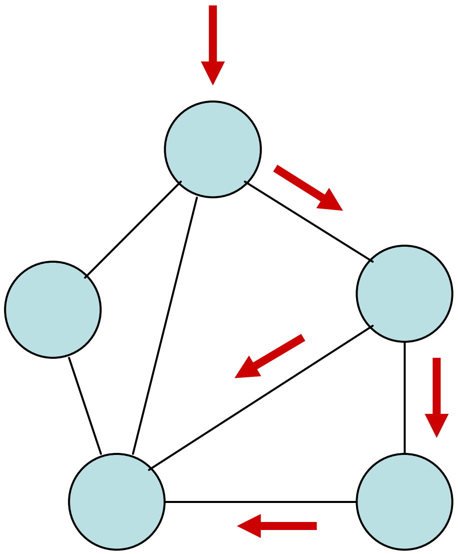




Key concepts

- Instantaneous snapshot:
  - Global state of entire distributed algorithm A, processes and channels, at some actual point in an execution.
  - Can use for checking invariants, checking for termination or deadlock, computing a function of the global state,…
- Consistent global snapshot:
  - Looks like an instantaneous snapshot, to every process and channel.
  - Good enough for checking invariants, checking for termination, …
- Stable property:
  - A property P of a global state such that, if P ever becomes true in an execution, P remains true forever thereafter.
  - E.g., termination, deadlock.
- Connection:
  - An instantaneous snapshot, or a consistent global snapshot, can be used to detect stable properties.

Termination detection [Dijkstra, Scholten]



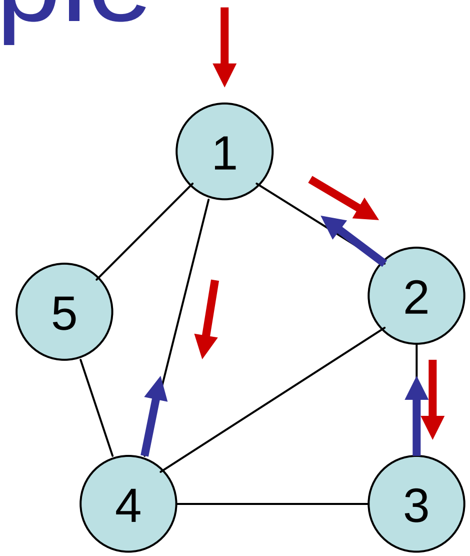
- Simple stable property detection problem.
- Connected, undirected network graph G = (V,E).
- Assume:
  - Algorithm A begins with all nodes quiescent (only inputs enabled).
  - An input arrives at exactly one node.
  - Starting node need not be predetermined.
- From there, computation can "diffuse" throughout the network, or a portion of the network.
- At some point, the entire system may become quiescent:
  - No non-input actions enabled at any node.
  - No messages in channels.
- Termination Detection problem:
  - If A ever reaches a quiescent state then the starting node should eventually output "done".
  - Otherwise, no one ever outputs "done".
- To be solved by a monitoring algorithm Mon(A).

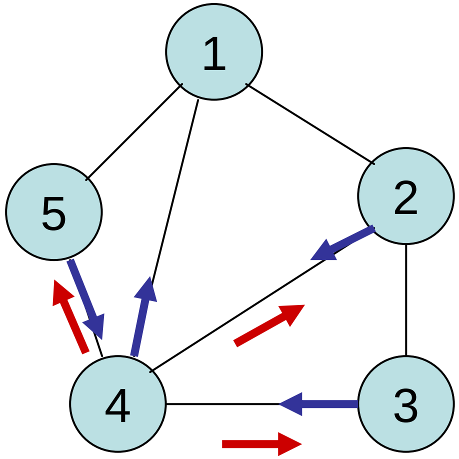
Dijkstra, Scholten Algorithm

- Augment A with extra pieces that construct and maintain a tree, rooted at the starting node, and including all the nodes currently active in A.
- Grows, shrinks, grows,…as nodes become active, quiescent, active,…
- Algorithm:
  - Execute A as usual, adding acks for all messages.
  - Messages of A treated like search messages in AsynchSpanningTree.
  - When a process receives an external input, it becomes the root, and begins executing A.
  - When any non-root process receives its first A message, it designates the sender as its parent in the tree, and begins participating in A.
  - Root process acks every message immediately.
  - Other processes ack all but the first message immediately.
  - Convergecast for termination:
    - If a non-root process finds its A-state quiescent and all its A-messages acked, then it cleans up: acks the first A-message, deletes all info about the termination protocol, becomes idle.
    - If it later receives another A message, it treats it like the first A message (defines a new parent, etc.), and resumes participating in A.
    - If root process finds A-state quiescent and all A-messages acked, reports done.

DS Algorithm, example

- First, p1 gets awakened by an external A input, becomes the root, sends A messages to p2 and p4, p2 sends an A-message to p3, all set up parent pointers and start executing A.
- Next, p4 sends A message to p3, acked immediately.
- p4 sends A message to p1, acked immediately.
- p1, p2, p3, and p4 send A messages to each other for a while, everything gets acked immediately.
- Tree remains unchanged.
- Next, p2 and p3 quiesce locally; p3 cleans up, sends ack to p2, p2 receives ack, p2 cleans up, sends ack to p1.
- Next, p4 sends A messages to p2, p3, and p5, yielding a new tree:
- Etc.





Correctness

- Claim this correctly detects termination of A: that all A-processes are in quiescent states and no A-messages are in the channels.

- Theorem 1: If Mon(A) outputs "done" then A has really terminated.
- Proof sketch:
  - Depends on key invariants:
    - If root is idle (not actively engaged in the termination protocol), then all nodes are idle, and the channels are empty.
    - If a node is idle then the part of A running at that node is quiescent.

Correctness

- Theorem 2: If A ever becomes quiescent, then eventually Mon(A) outputs “done”.

- Proof sketch: [See book]
  - Depends on key invariants:
    - If the root is not idle, then the parent pointers form a directed tree directed toward the root, spanning exactly the non-idle nodes.
    - Conservation of acks.
  - Suppose for contradiction that A quiesces, but the termination protocol does not output “done”.
  - Then the spanning tree must eventually stabilize to some final tree.
    - Because no new A-messages are sent or received, and acks are eventually finished.
  - But then any leaf node of the final tree is able to clean up and become idle.
  - Shrinks the final tree further, contradicting stability.
  - Implies that the root must output “done”.

Complexity

- Messages:
  - 2m, where m is the number of messages sent in A.
- Time from quiescence of A until output "done":
  - O(m d), where d = upper bound on message delay, ignore local processing time
  - Time to clean up the spanning tree.
- Bounds are most interesting if m << n.
  - E.g., for algorithms that involve only a limited computation in a small portion of a large network.

Application: Asynchronous BFS

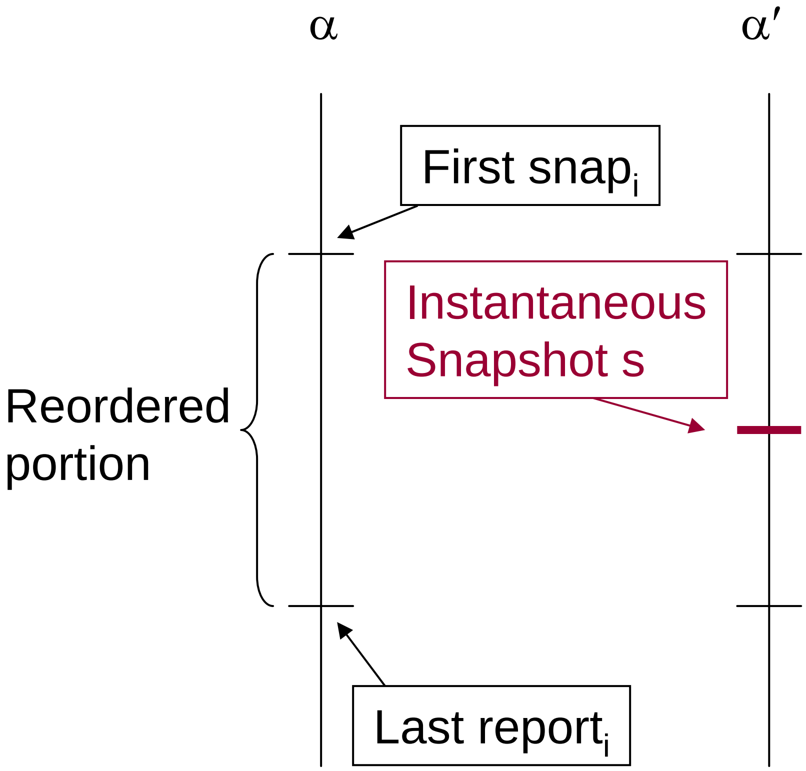
- Recall Asynchronous Breadth-First Search algorithm (AsynchBFS).
- Allows corrections.
- Doesn't terminate on its own; described ad hoc termination strategy earlier.
- It's a diffusing algorithm:
  - Wakeup input at the root node.
- So we can apply [Dijkstra, Scholten] to get a simple terminating version.
- Similarly for AsynchBellmanFord shortest paths.

Consistent Global Snapshots [Chandy, Lamport]

- Connected, undirected network graph G = (V,E).
- A is an arbitrary asynchronous distributed network algorithm.
- Mon(A) is supposed to take a "snapshot".
- Any number ($\geq$ 1) of nodes may receive $snap_i$ inputs, triggering the snapshot.
- Every node i should output $report_i$ containing:
  - A state for A_i.
  - States for all of A_i's incoming channels.
- Combination is a global state s.
- Must satisfy: If α is the actual underlying execution of A, then there is another execution, α', of A such that:
  - α and α' are indistinguishable to each individual A_i.
  - α and α' are identical up to the first snap and after the last report.
  - s is the actual global state at some point in α' in the snapshot interval.
- Implies the algorithm returns a Consistent Global Snapshot of A,
  - One obtained by reordering only the events occurring during the snapshot interval, and taking an instantaneous snapshot of the reordered execution, at some time during the snapshot interval.

Consistent Global Snapshot problem

- If α is the actual underlying execution of A, then there is another execution, α', of A such that:
  - α and α' are indistinguishable to each individual A_i.
  - α and α' are identical up to the first snap and after the last report.
  - s is the actual global state at some point in α' in the snapshot interval.



Chandy-Lamport algorithm

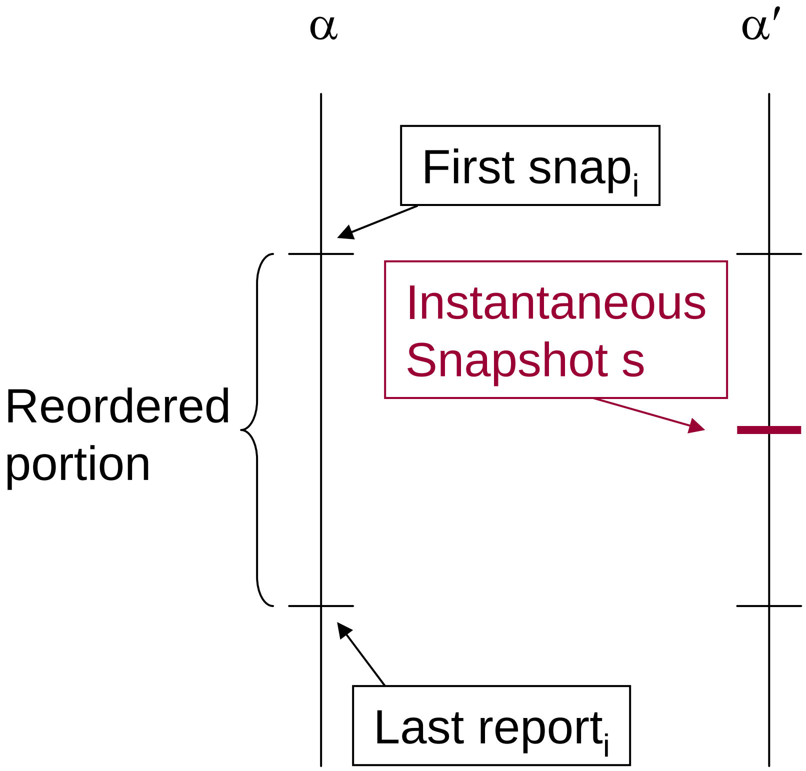
- Recall logical-time-based snapshot algorithm
  - Gets snapshot at a particular logical time t.
  - Depends on finding a good value of t.
- Chandy-Lamport algorithm can be viewed as running the same algorithm, but without explicitly using a particular logical time t.
- Instead, use marker messages to indicate where the logical time of interest occurs:
  - Put marker messages between messages sent at logical time $\leq t$ and those sent at logical times $> t$.
  - Relies on FIFO property of channels.

Chandy-Lamport algorithm

- Algorithm:
  - When not-yet-involved process i receives $snap_i$ input:
    - Snaps A_i's state.
    - Sends marker on each outgoing channel, thus marking the boundary between messages sent before and after the $snap_i$.
    - Thereafter, records all messages arriving on each incoming channel, up to the marker.
  - When process i receives first marker message without having previously received $snap_i$:
    - Snaps A_i's state, sends out markers, and begins recording messages as before.
    - Channel on which it got the marker is recorded as empty.

Correctness

- Termination: Easy to see
  - All snap eventually, because of either snap input or marker message.
  - Markers eventually sent and received on all channels.
- Returns a correct global state:
  - Let α be the underlying execution of A.
  - We must produce α', show that the returned state is an instantaneous snapshot of α'.



Returns a correct global state

- Let α be the underlying execution of A.
- Divide events of α into:
  - S_1: Those before the snap at their processes
  - S_2: Those after the snap at their processes
- Every event of α belongs to some process, so is in S_1 or S_2.
- Obtain α' by reordering events of α between first snap and last report, putting all S_1 events before all S_2 events, preserving causality order.
  - Causality: Orders events at each process and sends vs. receives.
- Q: How do we know we can do this?
- Claim that no send appears in S_2 whose corresponding receive is in S_1.
- In other words, for every send in S_2, the corresponding receive is also in S_2.
- The points between S_1 and S_2 at all processes form a consistent cut.

Returns a correct global state

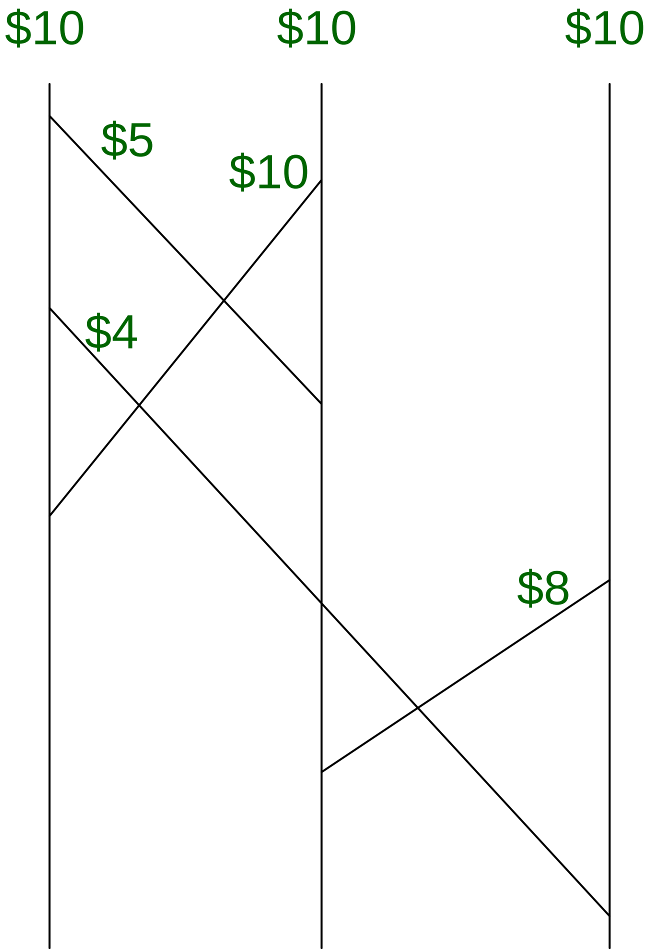
- Divide events of α into: S_1 (before snap) and S_2 (after snap).
- Obtain α' by reordering events of α between first snap and last report, putting all S_1 events before all S_2 events, preserving causality order.
- Can do this because no send appears in S_2 whose corresponding receive appears in S_1:
  - Follows from the marker discipline.
  - A send in S_2 occurs after the local snap, so after the marker is sent.
  - So the send produces a message that follows the marker on its channel.
  - Recipient snaps when it receives the marker (or sooner), so before receiving the message.
  - So the receive event is also in S_2.
- Returned state is exactly the global state of α' between the S_1 and S_2 events, that is, after all the pre-snap events and before all the post-snap events.
- Thus, returned state is an instantaneous snapshot of α'.

Remark

- Algorithm works in strongly-connected digraphs, as well as undirected graphs.

Example: Bank audit

- Distributed bank, money sent in reliable messages.
- Audit problem:
  - Count the total money in the bank.
  - While money continues to flow around.
  - Assume total amount of money is conserved (no deposits or withdrawals).

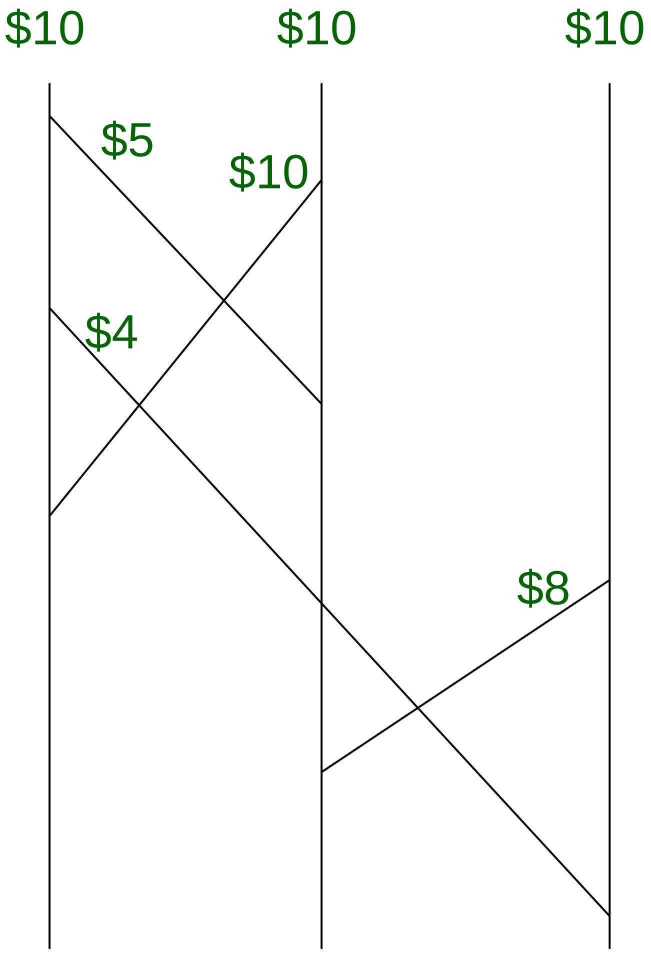


Trace

- Nodes 1,2,3 start with $10 apiece.

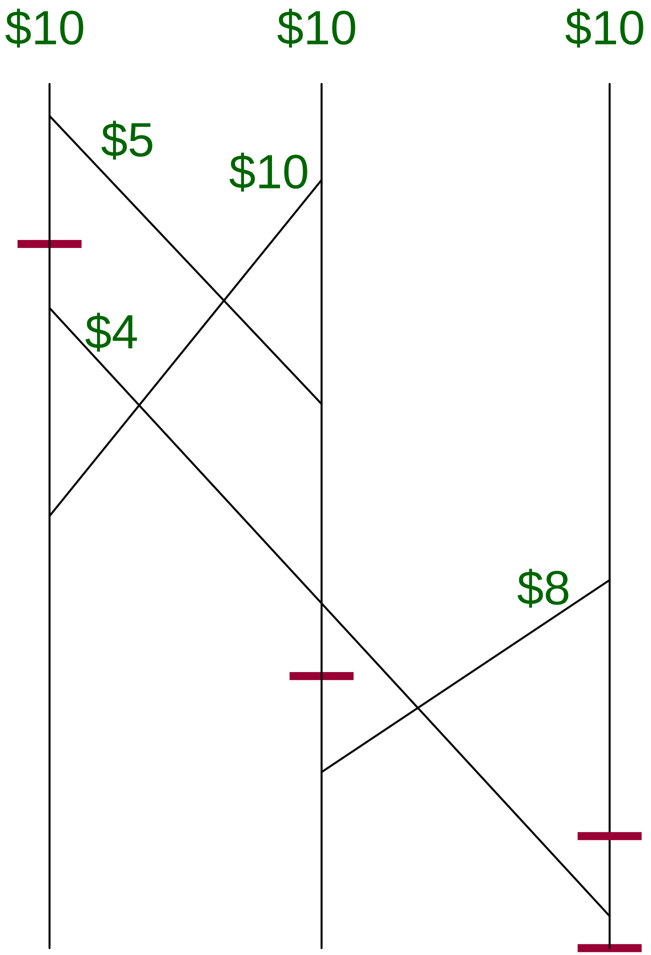
- Node 1 sends $5 to node 2.
- Node 2 sends $10 to node 1.
- Node 1 sends $4 to node 3.
- Node 2 receives $5 from node 1.
- Node 1 receives $10 from node 2.
- Node 3 sends $8 to node 2.
- Node 2 receives $8 from node 3.
- Node 3 receives $4 from node 1.

- Count the money?



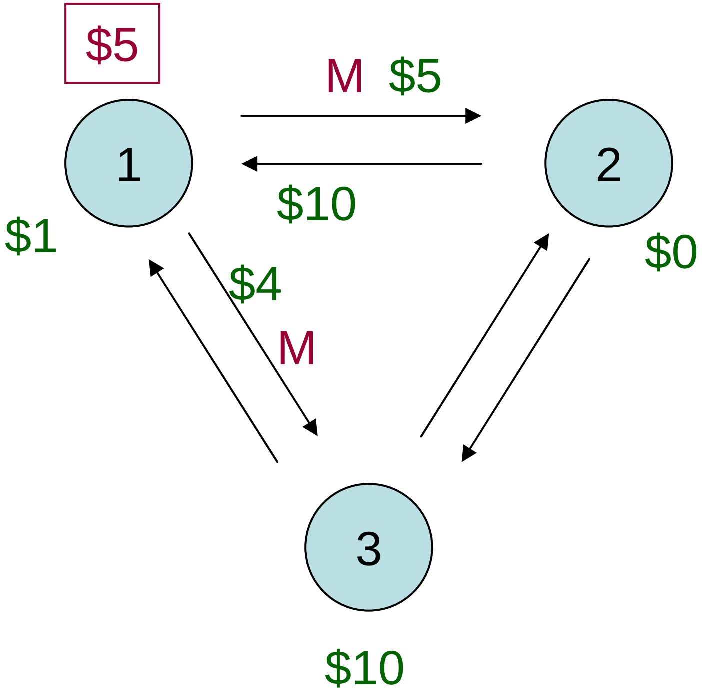
Chandy-Lamport audit

- Add snap input events:
- Q: Will local snapshots actually occur at these points?
- No, node 3 will snap before processing the $4 message, since it will receive the marker first.
- So actual local snapshot points are:



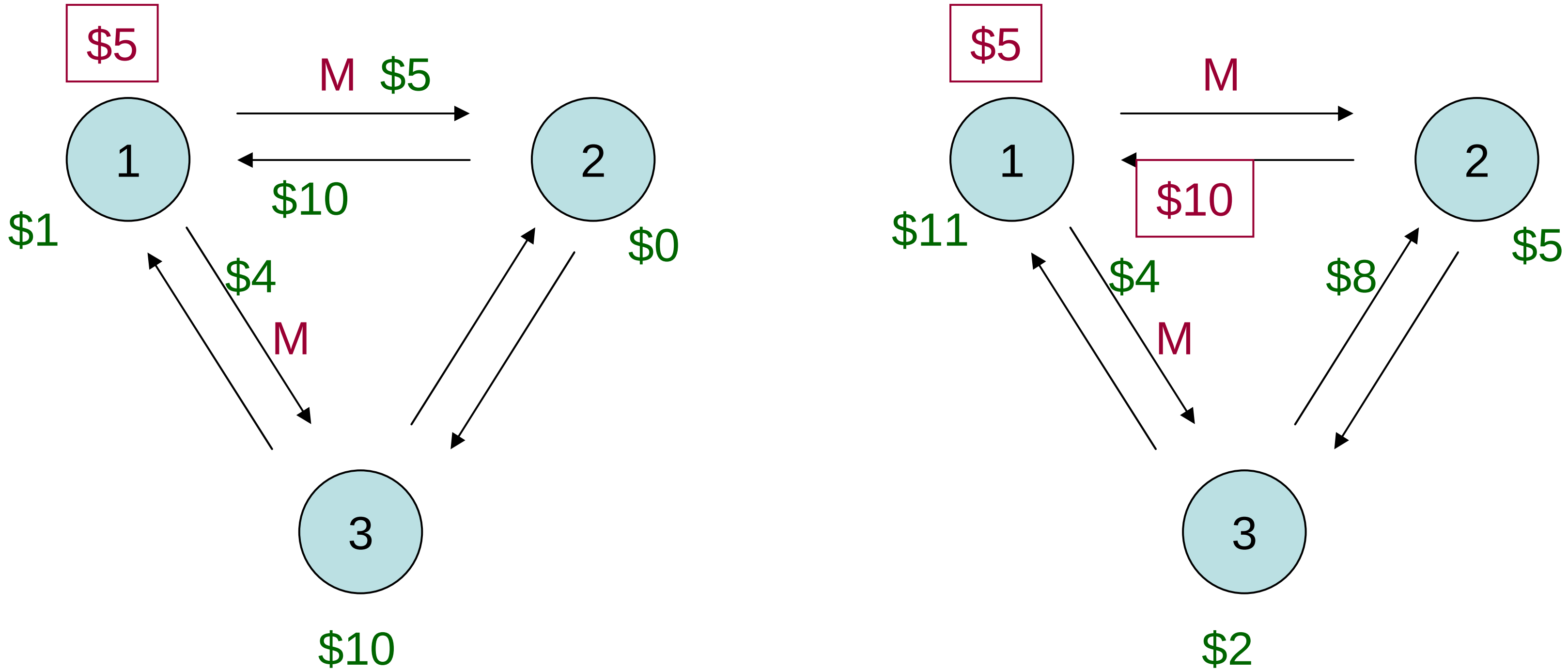
Trace, with snapshots

- Node 1 sends \$5 to node 2.
- Node 2 sends \$10 to node 1.
- Node 1 receives snap input, takes a snapshot, records state of A_1 as \$5, sends markers.
- Node 1 sends \$4 to node 3.



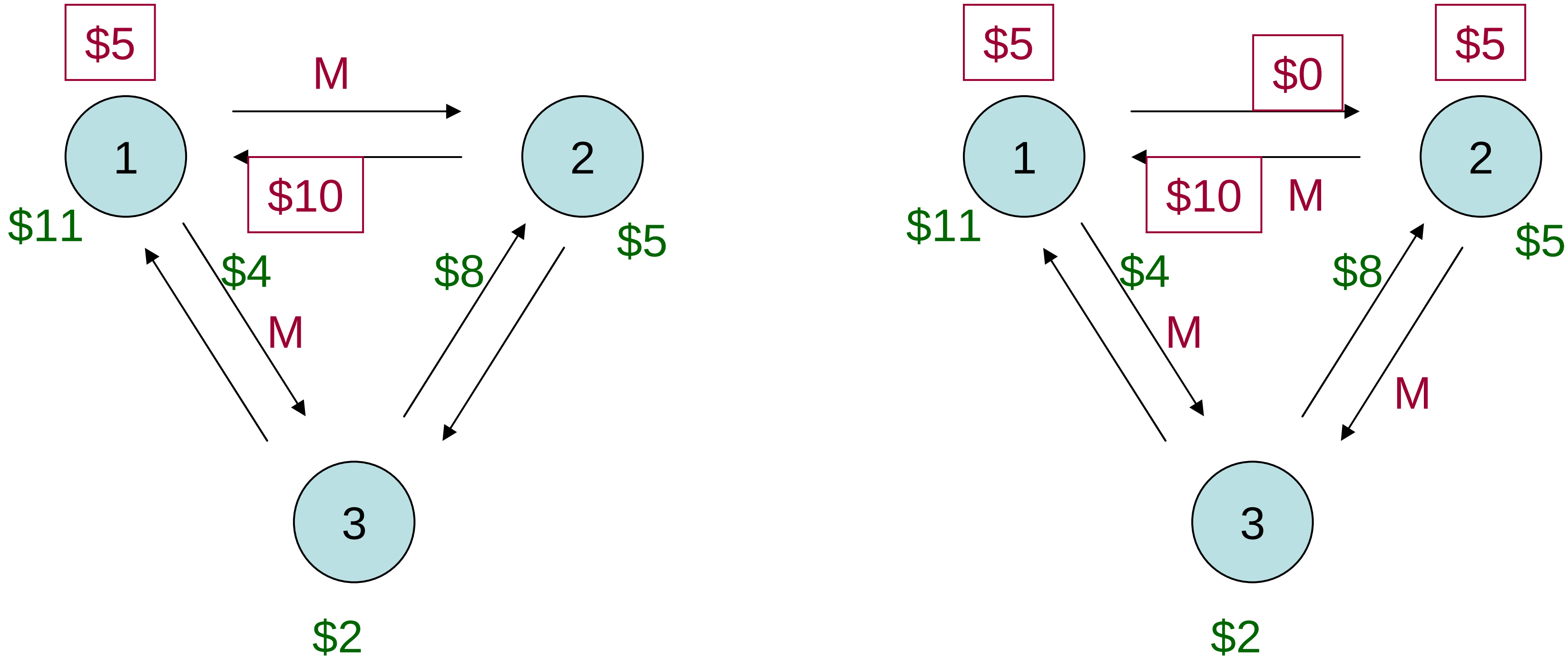
Trace, cont'd

- Node 2 receives \$5 from node 1.
- Node 1 receives \$10 from node 2, accumulates it in its count for $C_{2,1}$.
- Node 3 sends \$8 to node 2.



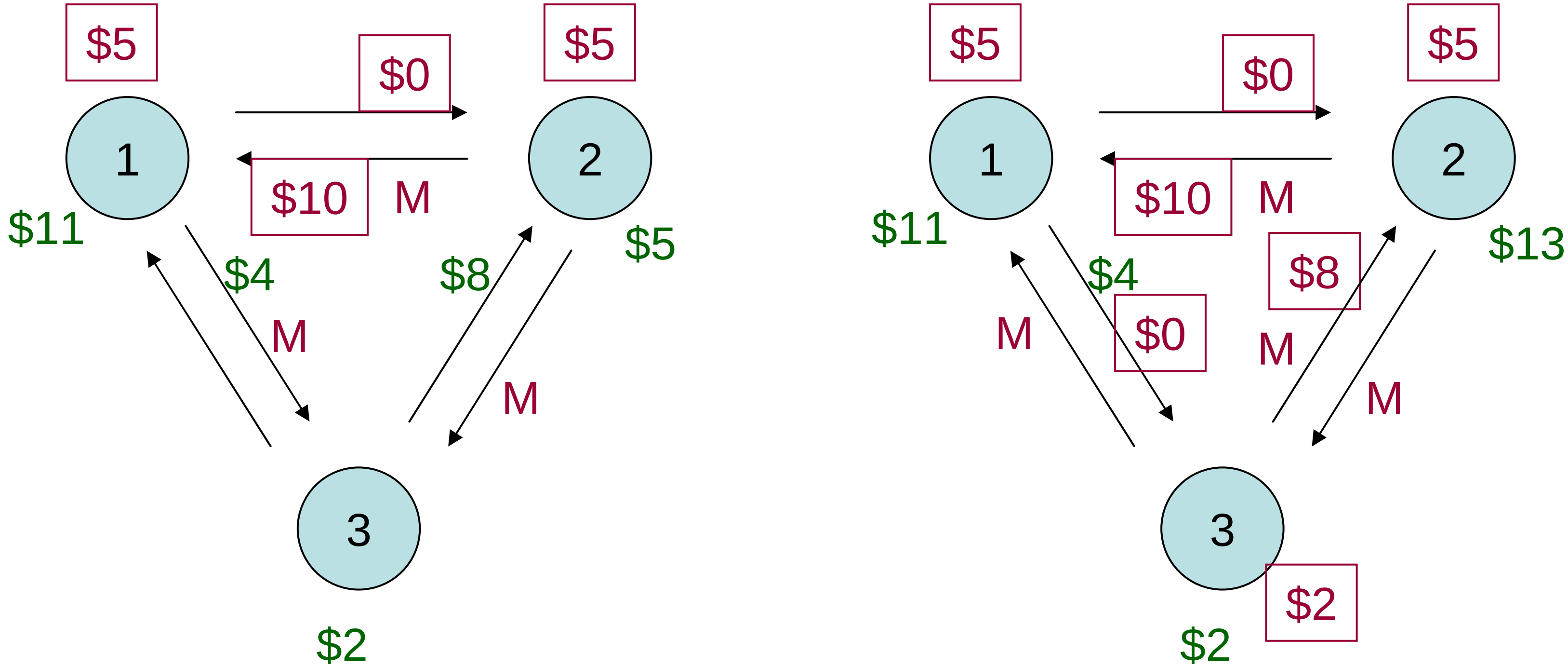
Trace, cont'd

- Node 2 receives M from node 1, takes a snapshot, records state of A_2 as \$5, records state of $C_{1,2}$ as \$0, sends markers.



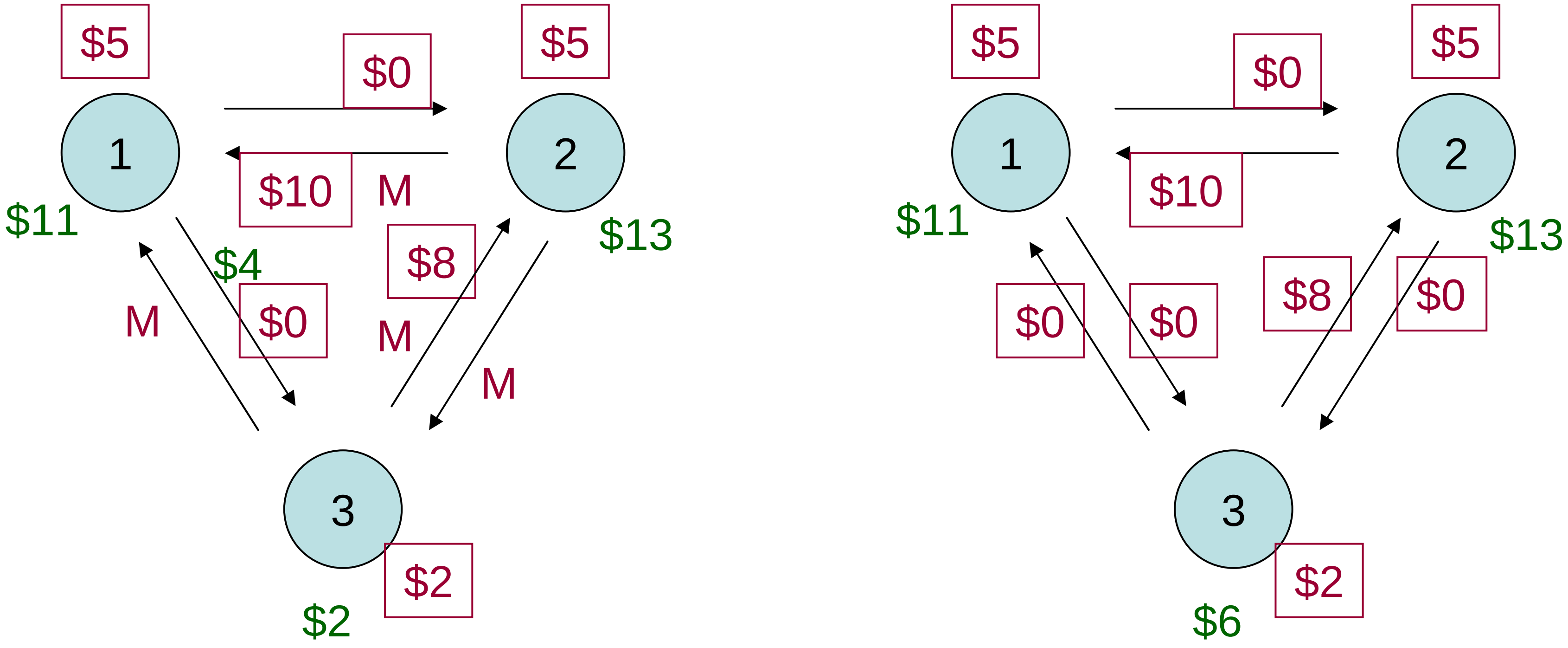
Trace, cont'd

- Node 2 receives $8 from node 3, accumulates it in its count for $C_{3,2}$.
- Node 3 receives M from node 1, takes a snapshot, records state of A_3 as $2, records state of $C_{1,3}$ as $0, sends markers.



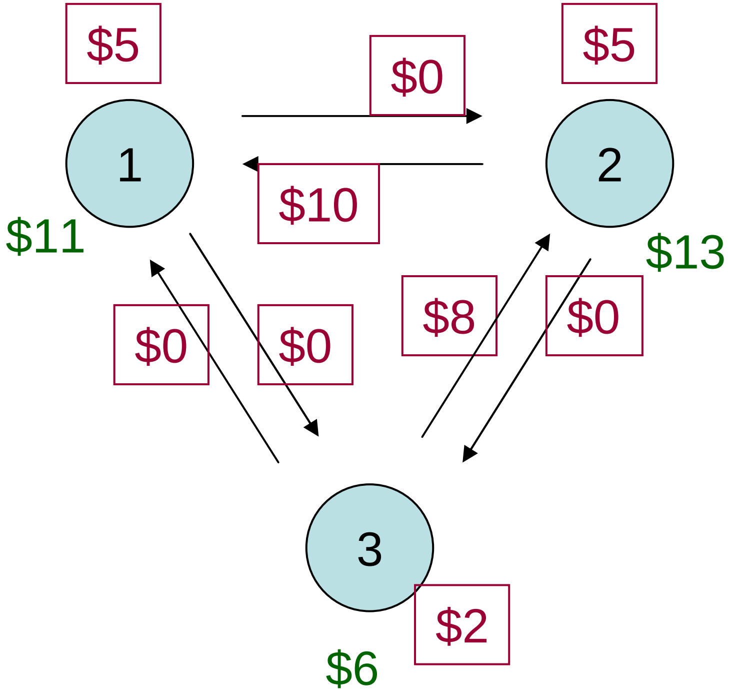
Trace, cont'd

- Node 3 receives $4 from node 1, ignored by snapshot algorithm.
- Remaining markers arrive, finalizing the counts for the remaining channels.



Total amount of money

- At beginning: $10 at each node = $30
- At end: $11 + $13 + $6 = $30
- In the snapshot:
  - Nodes: $5 + $5 + $2 = $12
  - Channels: $0 + $10 + $0
    + $8 + $0 + $0 = $18
  - Total: $30

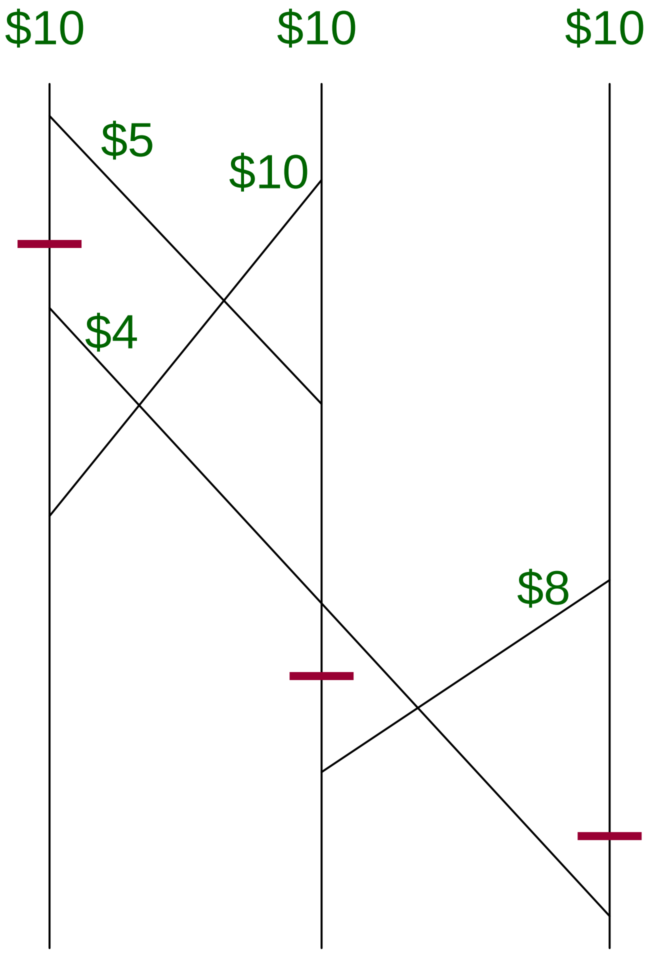


- Note:
  - The snapshot state never actually appears in the underlying execution α of the bank.
  - But it does appear in an alternative execution α' obtained by reordering events, aligning the local snapshots.

Original execution α

- Nodes 1,2,3 start with $10 apiece.

- Node 1 sends $5 to node 2.
- Node 2 sends $10 to node 1.
- Node 1 snaps.
- Node 1 sends $4 to node 3.
- Node 2 receives $5 from node 1.
- Node 1 receives $10 from node 2.
- Node 3 sends $8 to node 2.
- Node 2 snaps.
- Node 2 receives $8 from node 3.
- Node 3 snaps.
- Node 3 receives $4 from node 1.



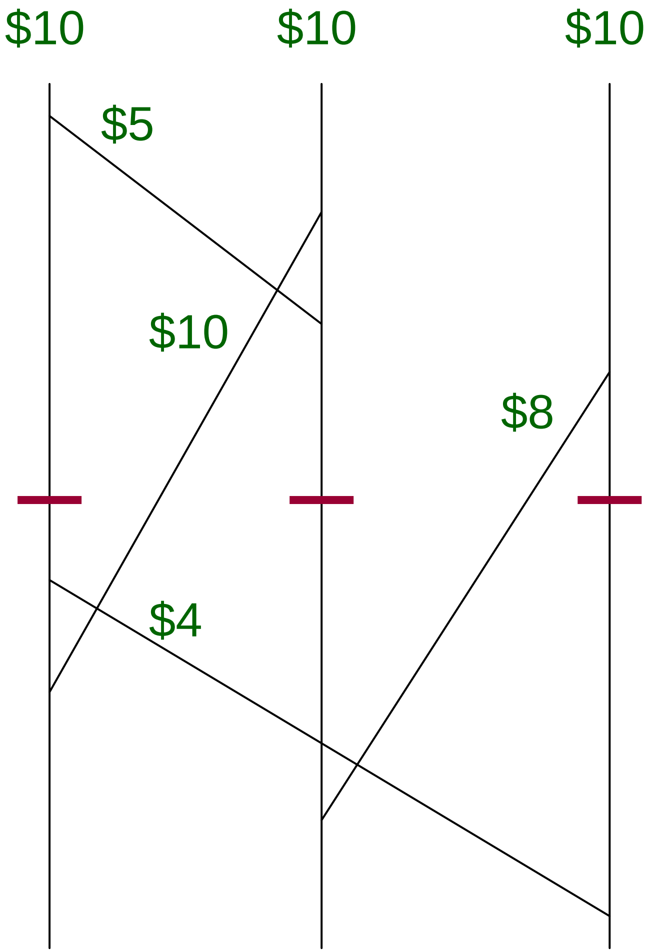
Reordered execution α'

- Nodes 1,2,3 start with $10 apiece.

- Node 1 sends $5 to node 2.
- Node 2 sends $10 to node 1.
- Node 2 receives $5 from node 1.
- Node 3 sends $8 to node 2.

- Everyone snaps.

- Node 1 sends $4 to node 3.
- Node 1 receives $10 from node 2.
- Node 2 receives $8 from node 3.
- Node 3 receives $4 from node 1.



Complexity

- Messages: O(|E|)
  - Traverse all edges, unlike [Dijkstra, Scholten]
- Time:
  - O(diam d), ignoring local processing time and pileups.

Applications of global snapshot

- Bank audit: As above.
- Checking invariants:
  - Global states returned are reachable global states, so any invariant of the algorithm should be true in these states.
  - Can take snapshot, check invariant (before trying to prove it).
- Checking requires some work:
  - Collect entire snapshot in one place and test the invariant there.
  - Or, keep the snapshot results distributed and use some distributed algorithm to check the property.
  - For "local" properties, this is easy:
    - E.g., consistency of values at neighbors: send-count$_i$ = receive-count$_j$ + number of messages in transit on channel from i to j.
  - For global properties, harder:
    - E.g., no global cycles in a "waits-for" graph, expressing which nodes are waiting for which other nodes.
    - Requires another distributed algorithm, for a static graph.

Stable Property Detection

- Stable property:
  - A property P of a global state such that, if P ever becomes true in an execution, P remains true forever thereafter.
  - Similar to an invariant, but needn't hold in all reachable states; rather, once it's true, it remains true.
- Example: Termination
  - Assume distributed algorithm A has no external inputs, but need not start in a quiescent state.
  - Essentially, inputs in initial state.
  - Terminated when:
    - All processes are in quiescent states, and
    - All channels are empty.
- Example: Deadlock
  - A set of processes are waiting for each other to do something, e.g., release a needed resource.
  - Cycle in a waits-for graph.

Snapshots and Stable Properties

- Can use [Chandy, Lamport] consistent global snapshots to detect stable properties.
- Run [CL], check whether stable property P is true in the returned snapshot state.
- Q: What does this show?
  - If P is true in the snapped state, then it is true in the real state after the final report output, and thereafter.
  - If P is false in the snapped state, then false in the real state just before the first snap input, and every state before that.
- Proof: Reachability arguments.

- Q: How can we be sure of detecting a stable property P, if it ever occurs?
- Keep taking snapshots.

Application: Asynchronous BFS

- Again recall AsynchBFS.
  - Allows corrections.
  - Doesn't terminate on its own; described ad hoc termination strategy earlier.
  - Diffusing algorithms, so we can apply [Dijkstra, Scholten] to get a simple terminating version.
- Alternatively, can use [Chandy, Lamport] algorithm to detect termination, using repeated snapshots.
- Eventually AsynchBFS actually terminates, and any snapshot thereafter will detect this.

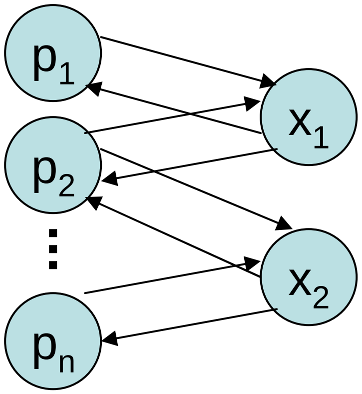
- Similarly for AsynchBellmanFord shortest paths.

Asynchronous Shared-Memory Systems

Asynchronous Shared-Memory systems

- We've covered basics of non-fault-tolerant asynchronous network algorithms:
  – How to model them.
  – Basic asynchronous network protocols---broadcast, spanning trees, leader election,…
  – Synchronizers (running synchronous algorithms in asynch networks)
  – Logical time
  – Global snapshots
- Now consider asynchronous shared-memory systems:

- Processes, interacting via shared objects, possibly subject to some access constraints.
- Shared objects are typed, e.g.:
  – Read/write (weak)
  – Read-modify-write, compare-and-swap (strong)
  – Queues, stacks, others (in between)



Asynch Shared-Memory systems

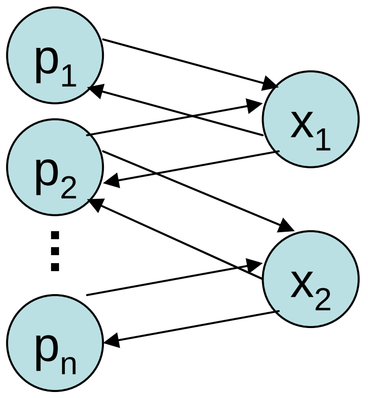
- Theory of ASM systems has much in common with theory of asynchronous networks:
  - Similar algorithms and impossibility results.
  - Even with failures.
  - Transformations from ASM model to asynch network model allow ASM algorithms to run in asynchronous networks.
    - “Distributed shared memory”.
- Historically, theory for ASM started first.
- Arose in study of early operating systems, in which several processes can run on a single processor, sharing memory, with possibly-arbitrary interleavings of steps.
- Currently, ASM models apply to multiprocessor shared-memory systems, in which several processes can run on separate processors and share memory.

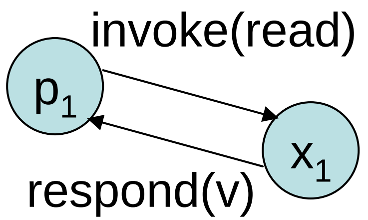
Topics

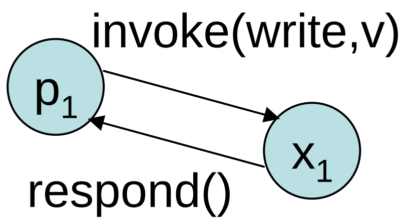
- Define the basic system model, without failures.
- Use it to study basic problems:
  - Mutual exclusion.
  - Other resource-allocation problems.
- Introduce process failures into the model.
- Use model with failures to study basic problems:
  - Distributed consensus
  - Implementing atomic objects:
    - Atomic snapshot objects
    - Atomic read/write registers
- Wait-free and fault-tolerant computability theory
- Modern shared-memory multiprocessors:
  - Practical issues
  - Algorithms
  - Transactional memory

Basic ASM Model, Version 1

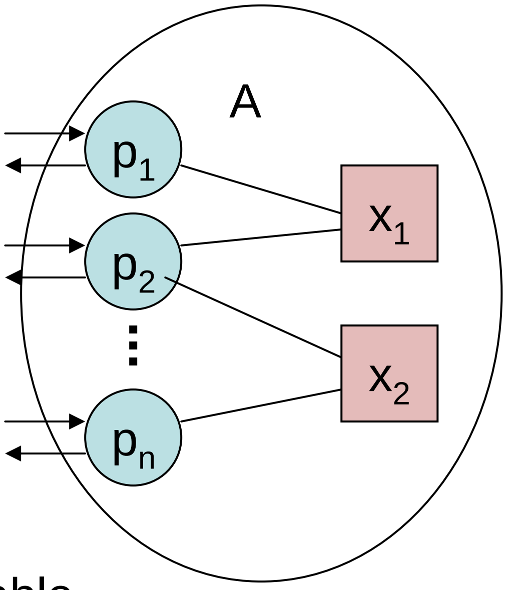
- Processes + objects, modeled as automata.
- Arrows:
  - Represent invocations and responses for operations on the objects.
  - Modeled as input and output actions.
- Fine-granularity model, can describe:
  - Delay between invocation and response.
  - Concurrent (overlapping) operations:
    - Object could reorder.
    - Could allow them to run concurrently, interfering with each other.
- We'll begin with a simpler, coarser model:
  - Object runs ops in invocation order, one at a time.
  - In fact, collapse each operation into a single step.
- Return to the finer model later.





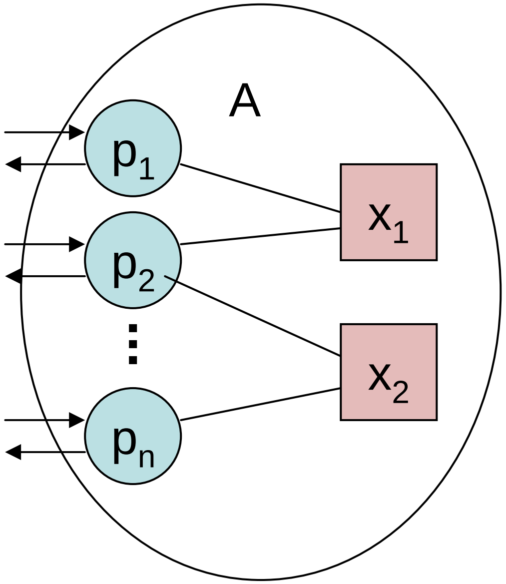


Basic ASM Model, Version 2



- One big shared memory system automaton A.
- External actions at process "ports".
- Each process i has:
  - A set $states_i$ of states.
  - A subset $start_{i,}$ of start states.
- Each variable x has:
  - A set $values_x$ of values it can take on.
  - A subset $initial_x$ of initial values.
- Automaton A:
  - States: State for each process, a value for each variable.
  - Start: Start states, initial values.
  - Actions: Each action associated with one process, and some also with a single shared variable.
  - Input/output actions: At the external boundary.
  - Transitions: Correspond to local process steps and variable accesses.
    - Action enabling, which variable is accessed, depend only on process state.
    - Changes to variable and process state depend also on variable value.
    - Must respect the type of the variable.
  - Tasks: One or more per process (threads).

Basic ASM Model



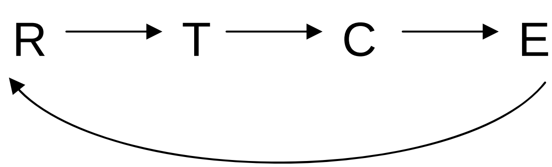
- Execution of A:
  - As specified by general definitions of executions, fair executions for I/O automata.
  - By fairness definition, each task gets infinitely many chances to take steps.
  - Model environment as a separate automaton, to express restrictions on environment behavior.
- Commonly-used variable types:
  - Read/write registers: Most basic primitive.
    - Allows access using separate read and write operations.
  - Read-modify-write: More powerful primitive:
    - Atomically, read variable, do local computation, write to variable.
  - Compare-and-swap, fetch-and-add, queues, stacks, etc.
- Different computability and complexity results hold for different variable types.

The Mutual Exclusion Problem

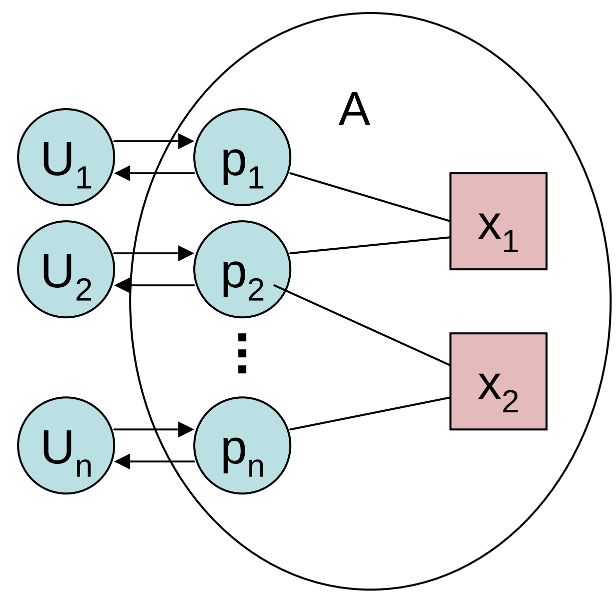
- Share one resource among n user processes, $U_1, U_2, \ldots, U_n$.
  - E.g., printer, portion of a database.
- U_i has four “regions”.
  - Subsets of its states, described by portions of its code.
  - C critical; R remainder; T trying; E exit

Protocols for obtaining and relinquishing the resource

- Cycle:

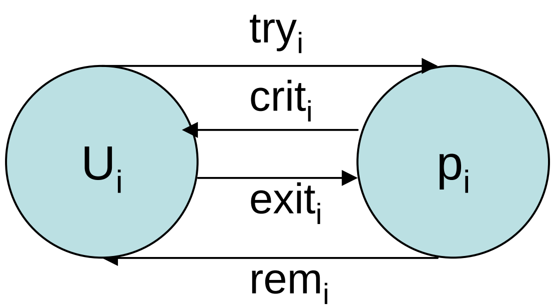


- Architecture:
  - U_is and A are IOAs, compose.

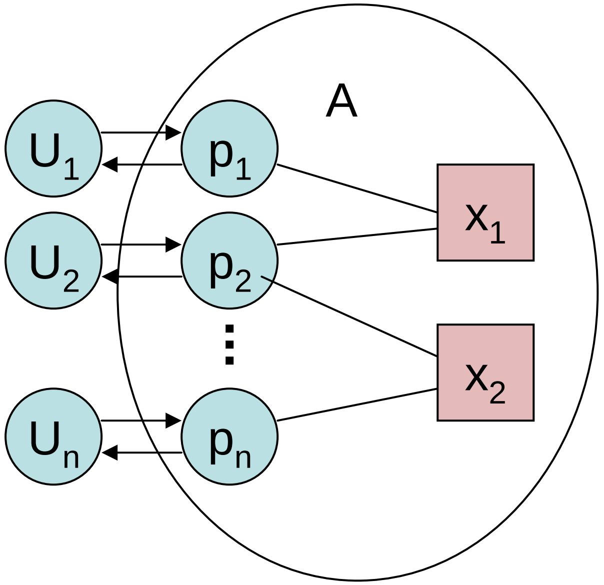


The Mutual Exclusion Problem

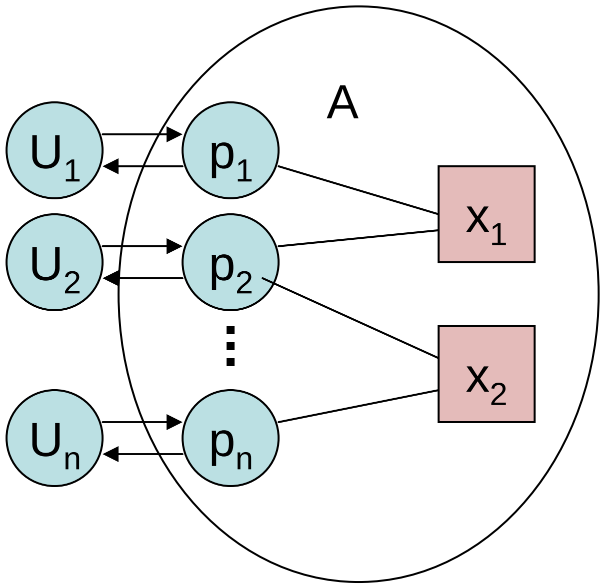
- Actions at user interface:
  - Connect U_i to P_i
  - p_i is U_i's "agent"



- Correctness conditions:
  - Well-formedness (Safety):
    - System also obeys cyclic discipline.
    - E.g., doesn't grant resource when it wasn't requested.
  - Mutual exclusion (Safety):
    - System never grants to > 1 user simultaneously.
    - Trace safety property.
    - Or, there's no reachable system state in which >1 user is in C at once.
  - Progress (Liveness):
    - From any point in a fair execution:
      - If some user is in T and no user is in C then at some later point, some user enters C.
      - If some user is in E then at some later point, some user enters R.



The Mutual Exclusion Problem

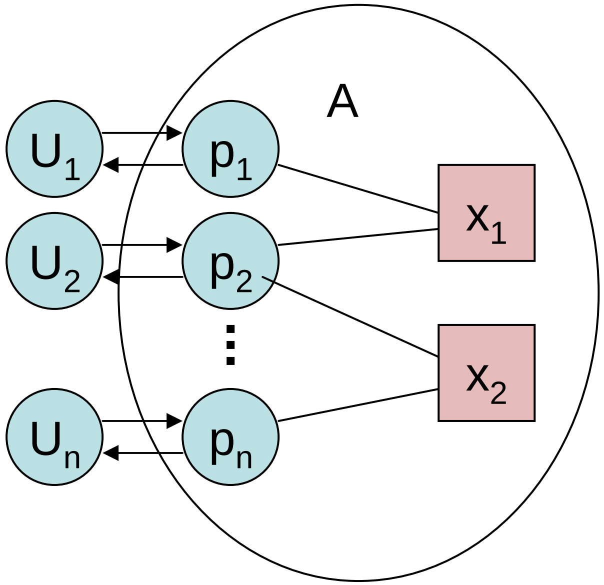


- Well-formedness (Safety):
  - System obeys cyclic discipline.
- Mutual exclusion (Safety):
  - System never grants to > 1 user.
- Progress (Liveness):
  - From any point in a fair execution:
    - If some user is in T and no user is in C then at some later point, some user enters C.
    - If some user is in E then at some later point, some user enters R.

- Conditions all constrain the system automaton A, not the users.
  - System determines if/when users enter C and R.
  - Users determine if/when users enter T and E.
  - We don't state any requirements on the users, but except that users respect well-formedness.

The Mutual Exclusion Problem

- Well-formedness (Safety):
- Mutual exclusion (Safety):
- Progress (Liveness):
  - From any point in a fair execution:
    - If some user is in T and no user is in C then at some later point, some user enters C.
    - If some user is in E then at some later point, some user enters R.



- Fairness assumption:
  - Progress condition requires fairness assumption (all process tasks continue to get turns to take steps).
  - Needed to guarantee that some process enters C or R.
  - In general, in the asynchronous model, liveness properties require fairness assumptions.
  - Contrast: Well-formedness and mutual exclusion are safety properties, don't depend on fairness.

One more assumption…

- No permanently active processes.
  - Locally-controlled actions enabled only when user is in T or E.
  - No always-awake, dedicated processes.
  - Motivation:
    - Multiprocessor settings, where users can run processes at any time, but are otherwise not involved in the protocol.
    - Avoid "wasting a processor".

Next time…

- The mutual exclusion problem.
- Mutual exclusion algorithms:
  - Dijkstra’s algorithm
  - Peterson’s algorithms
  - Lamport’s Bakery Algorithm
- Reading: Sections 10.1-10.7

6.852J / 18.437J Distributed Algorithms
Fall 2009