6.852: Distributed Algorithms
Fall, 2009

Class 11

Today's plan

- Lower bound on time for global synchronization.
- Logical time
- Applications of logical time
- Weak logical time and vector timestamps
- Reading:
  - Section 16.6, Chapter 18
  - [Lamport 1978: Time, Clocks, and the Ordering of Events in a Distributed System]
  - [Mattern]
- Next:
  - Consistent global snapshots
  - Stable property detection
  - Reading: Chapter 19

Lower Bound on Time for Global Synchronization

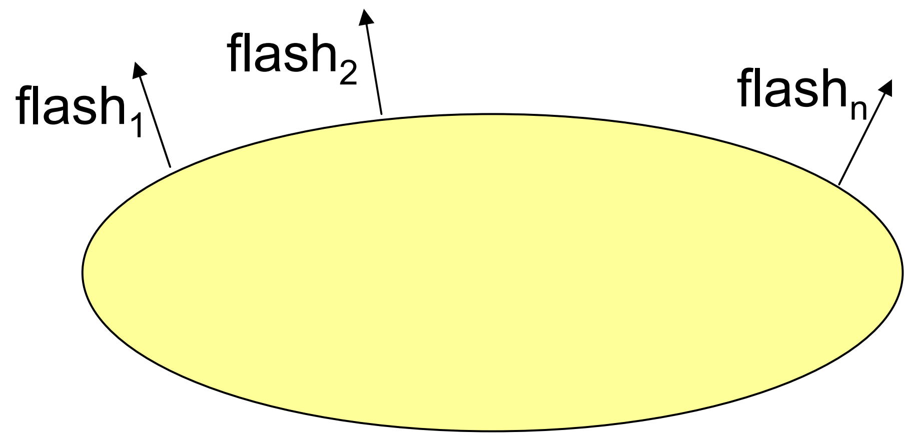
Lower bound on time

- Synchronizers emulate synchronous algorithms in a local sense:
  - Looks the same to individual users,
  - Not to the combination of all users---can reorder events at different users.
- Good enough for many applications (e.g., data management).
- Not for others (e.g., embedded systems).

- Now show that global synchronization is inherently more costly than local synchronization, in terms of time complexity.
- Approach:
  - Define a particular global synchronization problem, the k-Session Problem.
  - Show this problem has a fast synchronous algorithm, that is, a fast algorithm using GlobSynch.
    - Time O(k d), assuming GlobSynch takes steps ASAP.
  - Prove that all asynchronous distributed algorithms for this problem are slow.
    - Time Ω(k diam d).
  - Implies GlobSynch has no fast distributed implementation.
- In contrast, synchronizers yield fast distributed impls of LocSynch.

k-Session Problem

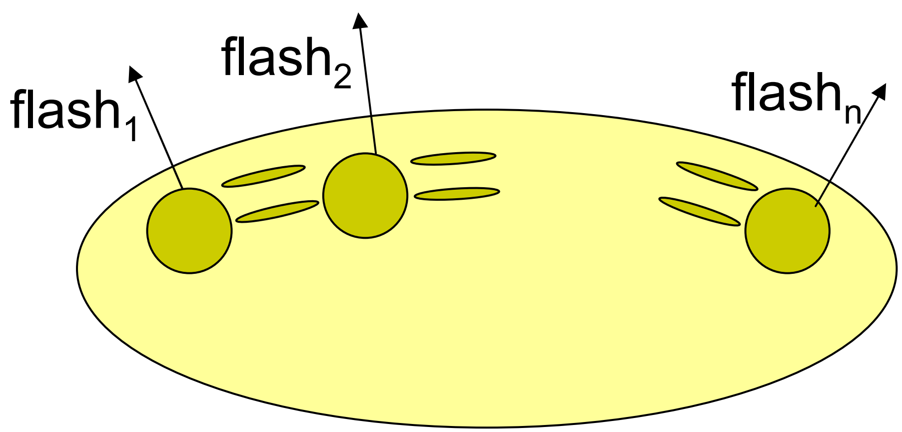
- Session:
  - Any sequence of flash events containing at least one $flash_i$ event for each location i.



- k-Session problem:
  - Perform at least k separate sessions (in every fair execution), and eventually halt.
- Original motivation:
  - Synchronization needed to perform parallel matrix computations that require enough interleaving of process steps, but tolerate extra steps.

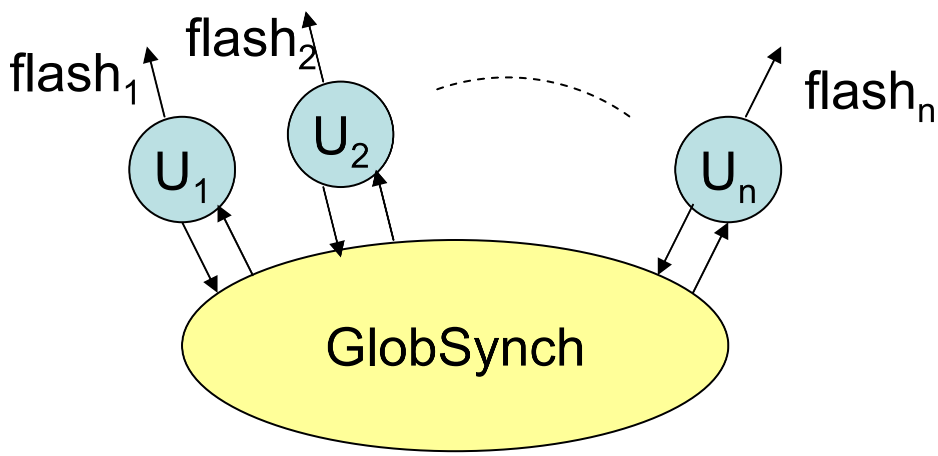
Example: Boolean matrix computation

- $n = m^3$ processes compute the transitive closure of $m \times m$ Boolean matrix M.
- $p_{i,j,k}$ repeatedly does:
  - read M(i,k), read M(k,j)
  - If both are 1 then write 1 in M(i,j)
- Each $flash_{i,j,k}$ in abstract session problem represents a chance for $p_{i,j,k}$ to read or write a matrix entry.
- With enough interleaving (O (log n) sessions), this is guaranteed to compute transitive closure.



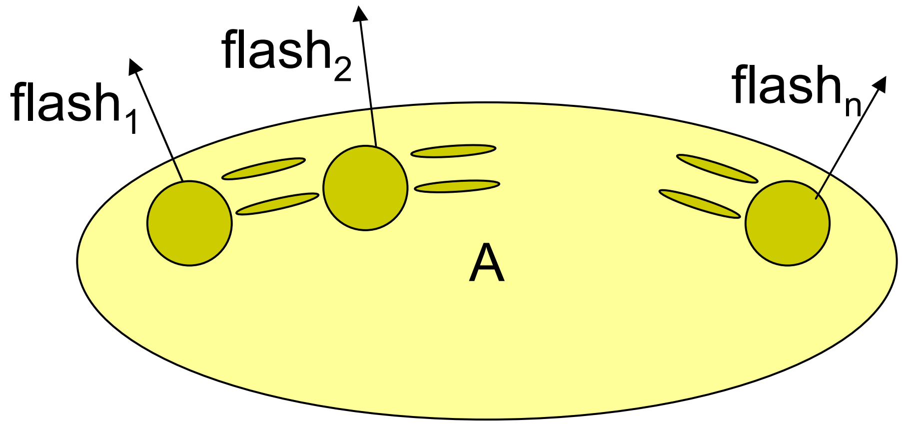
Synchronous solution

- Fast algorithm using GlobSynch:
  - Just flash once at every round.
  - k sessions done in time O(k d), assuming GlobSynch takes steps ASAP.



Asynchronous lower bound

- Consider distributed algorithm A that solves the k-session problem.
- Consists of process automata and FIFO send/receive channel automata.



- Assume:
  - d = upper bound on time to deliver any message (don't count pileups)
  - l = local processing time, l << d
- Define time measure T(A):
  - Timed execution α: Fair execution with times labeling events, subject to upper bound of d on message delay, l for local processing.
  - T(α) = time of last flash in α
  - T(A) = supremum, over all timed executions α, of T(α).

Lower bound

- Theorem 2: If A solves the k-session problem then $T(A) \geq (k-1)$ diam d.
- Factor of diam worse than the synchronous algorithm.

- Definition: Slow timed execution: All message deliveries take exactly the upper bound time d.

- Proof: By contradiction.
  - Suppose $T(A) < (k-1)$ diam d.
  - Fix α, any slow timed execution of A.
  - α contains at least k sessions.
  - α contains no flash event at a time $\geq (k-1)$ diam d.
  - So we can decompose $\alpha = \underbrace{\alpha_1 \alpha_2 \ldots \alpha_{k-1}}_{\alpha'} \alpha''$, where:
    - Time of last event in α' is $< (k-1)$ diam d.
    - No flash events occur in α''.
    - Difference between the times of the first and last events in each α_r is $<$ diam d.

Lower bound, cont'd

- Now reorder events in α, while preserving dependencies:
  - Events of same process.
  - Send and corresponding receive.
- Reordered execution will have < k sessions, contradiction.
- Fix processes, j_0 and j_1, with dist(j_0,j_1) = diam (maximum distance apart).
- Reorder within each α_r separately:
  - For α_1: Reorder to $\beta_1 = \gamma_1 \delta_{1,}$ where:
    - γ_1 contains no event of j_0, and
    - δ_1 contains no event of j_1.
  - For α_2: Reorder to $\beta_2 = \gamma_2 \delta_{2,}$ where:
    - γ_1 contains no event of j_1, and
    - δ_1 contains no event of j_0.
  - Alternate thereafter.

Lower bound, cont'd

- If the reordering yields a fair execution of A (ignore timing here), then we get a contradiction, because it contains $\leq$ k-1 sessions:
  - No session entirely within γ_1, (no event of j_0).
  - No session entirely within $\delta_1 \gamma_2$ (no event of j_1).
  - No session entirely within $\delta_2 \gamma_3$ (no event of j_0).
  - …
  - Thus, every session must span some γ_r - δ_r boundary.
  - But, there are only k-1 such boundaries.

- So, it remains only to construct the reordering.

Constructing the reordering

- WLOG, consider α_r for r odd.
- Need $\beta_r = \gamma_r \delta_r$, where γ_r contains no event of j_0, δ_r no event of j_1.

- If α_r contains no event of j_0 then don't reorder, just define $\gamma_r = \alpha_r$, $\delta_r = \lambda$.
- Similarly if α_r contains no event of j_1.
- So assume α_r contains at least one event of each.
- Let π be the first event of j_0, φ the last event of j_1 in α_r.

- Claim: φ does not depend on π.
- Why: Insufficient time for messages to travel from j_0 to j_1:
  - Execution α is slow (message deliveries take time d).
  - Time between π and φ is < diam d.
  - j_0 and j_1 are diam apart.

- Then, we can reorder α_r to β_r, in which π comes after φ.
- Consequently, in β_r, all events of j_1 precede all events of j_0.
- Define γ_r to be the part ending with φ, δ_r the rest.

Logical Time

[Lamport: Time, clocks,…]

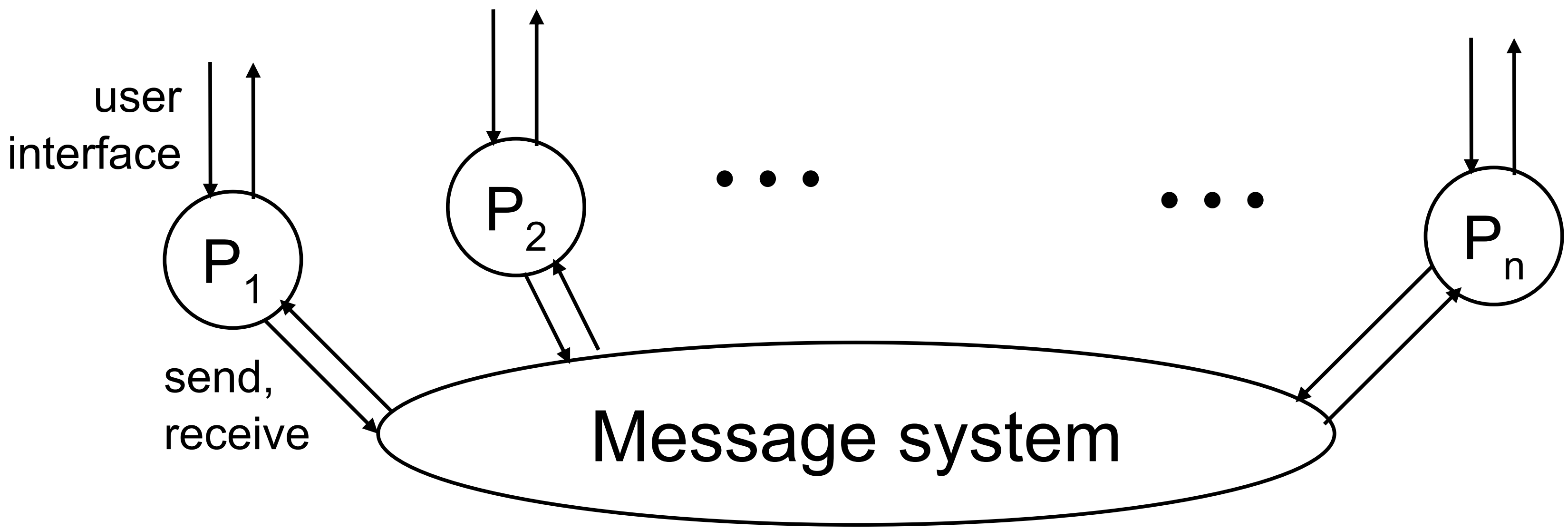
- Winner of first Dijkstra Prize, 2000.

> “Jim Gray once told me that he heard two different opinions of this paper: that's it trivial and that it's brilliant. I can't argue with the former, and I'm disinclined to argue with the latter.” –Lamport

Logical time

- An important abstraction, which simplifies programming for asynchronous networks
- Imposes a single total order on events occurring at all locations.
- Processes know the order.
- Assign logical times (elements of some totally ordered set T, e.g., the real numbers) to all events in an execution of an asynchronous network system, subject to some properties that make the logical times “look like real times”.
- Applications:
  - Global snapshot
  - Replicated state machines, mutual exclusion,…

Logical time



- Consider a send/receive system A with FIFO channels, based on a strongly connected digraph.
- Events of A:
  - User interface events
  - Send and receive events
  - Internal events of process automata
- Q: What conditions should logical times satisfy?

Logical time

- For execution α, function ltime from events in α to totally-ordered set T is a 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. (No "Zeno" behavior.)
- Properties 2 and 3 say that ltimes are consistent with dependencies between events. But we can reorder independent events at different processes.
- Under these conditions, ltime "looks like" real time, to all the processes individually:
- Theorem: For every fair execution α with an ltime function, there is another fair execution α' with events in ltime order such that $\alpha \mid P_i = \alpha' \mid P_i$ for all i.

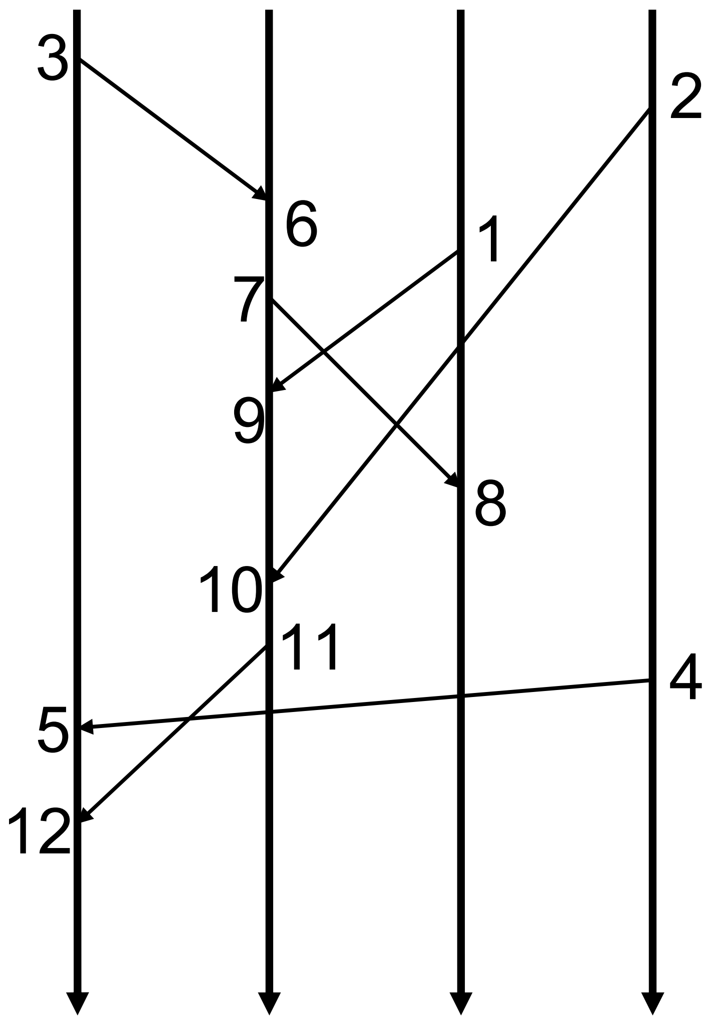
Logical time

- Function ltime from events in α to T is a logical time assignment if:
  1. ltimes are distinct: ltime(e_1) ≠ 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.
- Theorem: For every fair execution α with an ltime function, there is another fair execution α' with events in ltime order such that $\alpha \mid P_i = \alpha' \mid P_i$ for all i.
- Proof:
  - Use properties of ltime.
  - Reorder actions of α in order of ltimes; a unique such sequence exists, by Properties 1 and 4.
  - By Properties 2, and 3, this reordering preserves dependencies, so we can fill in the states to give the needed execution α'.
  - Indistinguishable to each process because we preserve all dependencies.

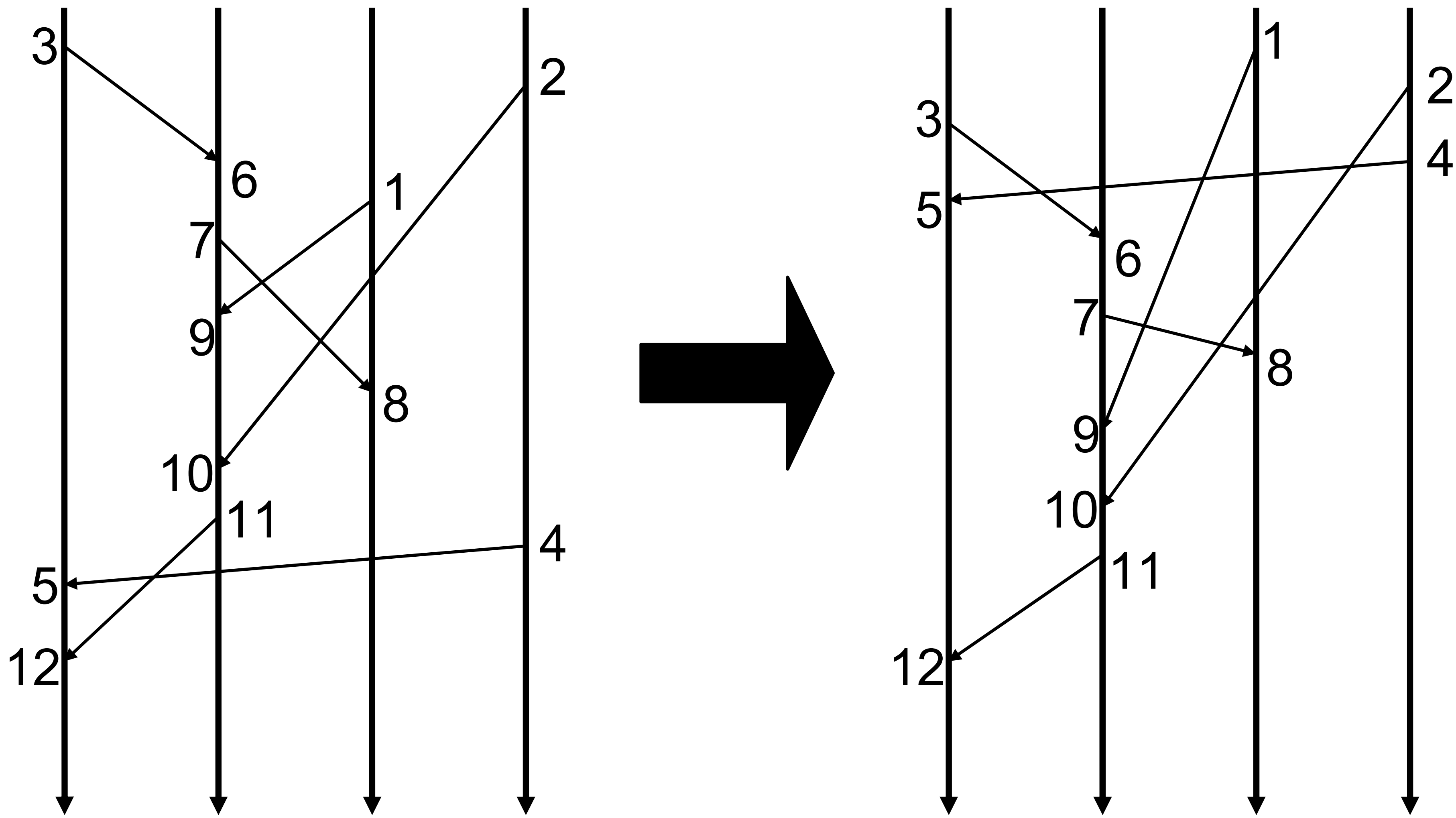
Logical time

- For execution α, function ltime from events in α to T is a logical time assignment if:
  1. ltimes are distinct: ltime(e_1) ≠ 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.
- Combination of dependencies described in Properties 2 and 3 often called causality, or Lamport causality.
- Common way to represent dependencies: Causality Diagram:

Logical time



Logical time



Lamport's algorithm for generating logical times

- Based on timestamping algorithm by Johnson and Thomas.
- Each process maintains a local nonnegative integer clock variable, used to count steps.
- clock is initially 0.
- Every event of the process (send, receive, internal, or user interface) increases clock:
  - When process does an internal or user interface step, increments clock.
  - When process sends, first increments clock, then piggybacks the new value c on the message, as a timestamp.
  - When process receives a message with timestamp c, increases clock to be max(clock, c) + 1.

- Using the clocks to generate logical time for events:
  - ltime of an event is (c,i), where
    - c = clock value immediately after the event
    - i = process index, to break ties
  - Order the (c,i) pairs lexicographically.

Lamport's algorithm generates logical times

1. Events' ltimes are unique.
   - Because clock at each process is increased at every step and we use process indices as tiebreakers.
2. Events of each individual process have strictly increasing ltimes.
   - The rules ensure this.
3. ltime(send) < ltime(receive) for same message.
   - By the way the receiver determines the clock after the receive event.
4. Non-Zeno.
   - Because every event increases the local clock by at least 1 and there are only finitely many processes.

Welch's algorithm

- What if we already have clocks?
  - Monotonically non-decreasing, unbounded.
  - Can't change the clock (e.g., maintained by a separate algorithm, or arrive from some external time source).
- Welch's algorithm:
  - Idea: Instead of advancing the clock in response to received timestamps, simply delay the receipt of "early" messages.
  - Messages carry clock value from sender.
  - Receiver puts incoming messages in a FIFO buffer.
  - At each locally-controlled step, first remove from buffer all messages whose timestamp < current clock, and process them, in same order in which they appear in the buffer.
  - Logical time of event is (c,i,k), order lexicographically.
    - c = local clock value when event "occurs"
      - receive event is said to "occur" when message is removed from buffer, not when it first arrives.
    - i = process index, first-order tiebreaker
    - k = sequence number, second-order tiebreaker

Logical time in broadcast systems

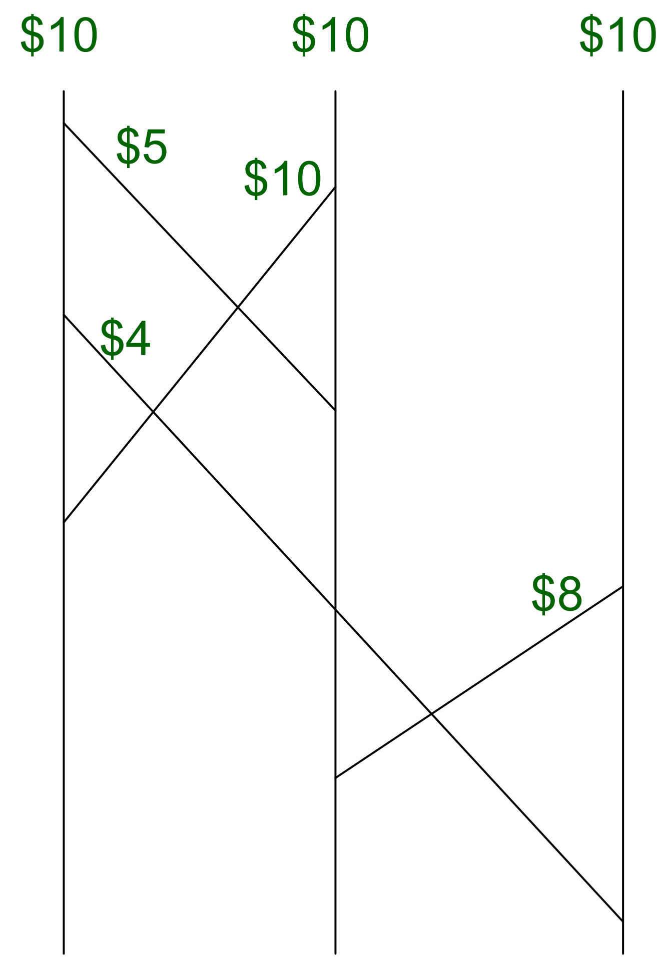
- Analogous definition and theorem:

- For execution α, function ltime from events in α to T is a logical time assignment if:
  1. ltimes are distinct: ltime(e_1) ≠ ltime(e_2) if $e_1 \neq e_2$.
  2. ltimes of events at each process are monotonically increasing.
  3. ltime(bcast) < ltime(receive) for same message.
  4. For any t, the number of events e with ltime(e) < t is finite.
- Theorem: For every fair execution α with an ltime function, there is another fair execution α' with events in ltime order such that $\alpha \mid P_i = \alpha' \mid P_i$ for all i.

Applications of Logical Time

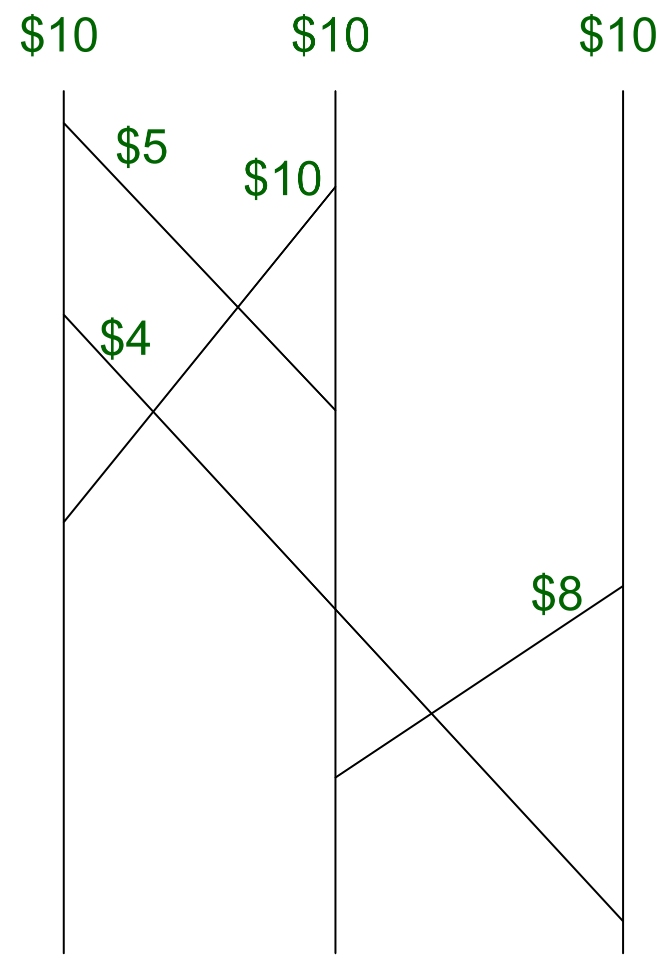
Applications of logical time: Banking system

- Distributed banking system with transfers (no external deposits or withdrawals).
- Assume:
  - Asynchronous send/receive system.
  - Each process has an account with money ≥ 0.
  - Processes can send money at any time to anyone.
    - Send message with value, subtract value from account.
    - Add value received in message to account.
  - Add "dummy" $0 transfers (heartbeat messages).



Banking system

- Algorithm triggered by input signal to one or more processes; processes awaken upon receiving either such a signal or a message from another process.
- Require:
  - Each process should output local balance, so that the total of the balances = correct amount of money in the system.
    - Well-defined because there are no deposits/withdrawals.
  - Don't "interfere" with underlying money transfer, just "observe" it.



Banking system algorithm

- Assume logical-time algorithm, which assigns logical times to all banking system events.
- Algorithm assumes agreed-upon logical time value t.
  - Each process determines value of its money at logical time t.
    - Specifically, after all events with ltime ≤ t and before all events with ltime > t.
  - Each process determines, for each incoming channel, the amount of money in transit at time t.
    - Specifically, in messages sent at ltime ≤ t and received at ltime > t.
    - Start counting from when local clock > t, stop when message timestamp > t.
- Q: What if local clock > t when node wakes up?
  - Keep logs just in case, or
  - Retry with different values of t.

Applications of logical time: Global snapshot

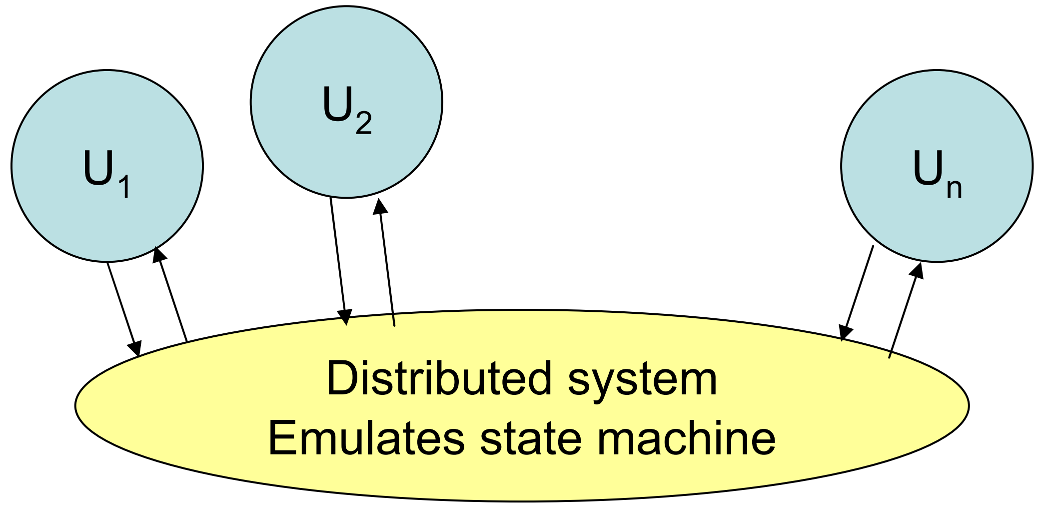
- Generalizes banking system.
- Assume:
  - Arbitrary asynchronous send/receive system A that sends infinitely many messages on each channel.
- Require:
  - Global snapshot of system state (nodes and channels) at some point after a triggering input.
  - Should not interfere with the system's operation.
- Useful for debugging, system backups, detecting termination.
- Use same strategy as for bank audit:
  - Select logical time, all snap at that time (nodes and channels).
  - Combining all these results give global snapshot of an "equivalent" execution.

Applications of logical time: Replicated state machines (RSMs)

- Important use of logical time.
- A focal point of Lamport's paper.
- Allows a distributed system to simulate a single centralized state machine.
- Centralized state machine:
  - V: Set of possible states
  - v_0: Initial state
  - invs: Set of possible invocations
  - resps: Set of possible responses
  - trans: invs $\times$ V $\rightarrow$ resps $\times$ V: Transition function
- Same formal definition as shared variable, defined in Chapter 9 (see next week).

Replicated State Machines

- Users of distributed system submit invocations, get responses in well-formed manner (blocking invocations).



- Want system to look like "atomic" version of the centralized state machine (defined in Chapter 13).
- Allows possible delays before and after actually operating on the state machine).
- Could weaken requirement to "sequential consistency", same idea but allows reordering of events at different nodes.

RSM algorithm

- Assume broadcast network.
- First attempt:
  - Originator of an invocation broadcasts the invocation to all processes (including itself).
  - All processes (including the originator) perform the transition on their copies when they receive the messages.
  - When originator performs the transition, determines response to pass to the user.

- Not quite right---all processes should peform the transitions in the same order.
- So, use logical time to order the invocations.

RSM algorithm

- Assume logical times.
- Originator of an invocation bcasts the invocation to all processes, including itself; attaches the logical time of the bcast event.
- Each process maintains state variables:
  - X: Copy of machine state.
  - inv-buffer: Invocations it has heard about and their timestamps
    - Timestamp = logical time of bcast event.
  - known-time: Vector of largest logical times for each process
    - For itself: Logical time of last local event.
    - For each other node j: Timestamp of last message received from j.
- Process may perform invocation π from its inv-buffer, on its copy X of the machine state, when π has the smallest timestamp of any invocation in inv-buffer, and known-time(j) $\geq$ timestamp(π) for all j.
- After performing π, remove it from inv-buffer.
- If π originated locally, then also respond to the user.

Correctness

- Liveness: Termination for each operation
  - LTTR. Depends on logical times growing unboundedly and all nodes sending infinitely many messages.
- Safety: Atomicity (each operation "appears to be performed" at a point in its interval, as in a centralized machine):
  - Each process applies operations in the same (logical time) order.
    - FIFO channels ensure that no invocations are "late".
  - Each operation "appears to be performed" at a point in its interval:
    - Define a serialization point for each operation π---a point in π's interval where we can "pretend" π occurred.
    - Namely, serialization point for π is the earliest point when all processes have reached the logical time t of π's bcast event.
    - Claim this point is within π's interval:
      - It's not before the invocation, because the originating process doesn't reach time t until after the invocation arrives.
      - It's not after the response, because the originator waits for all known-times to reach t before applying the operation and responding to the user.

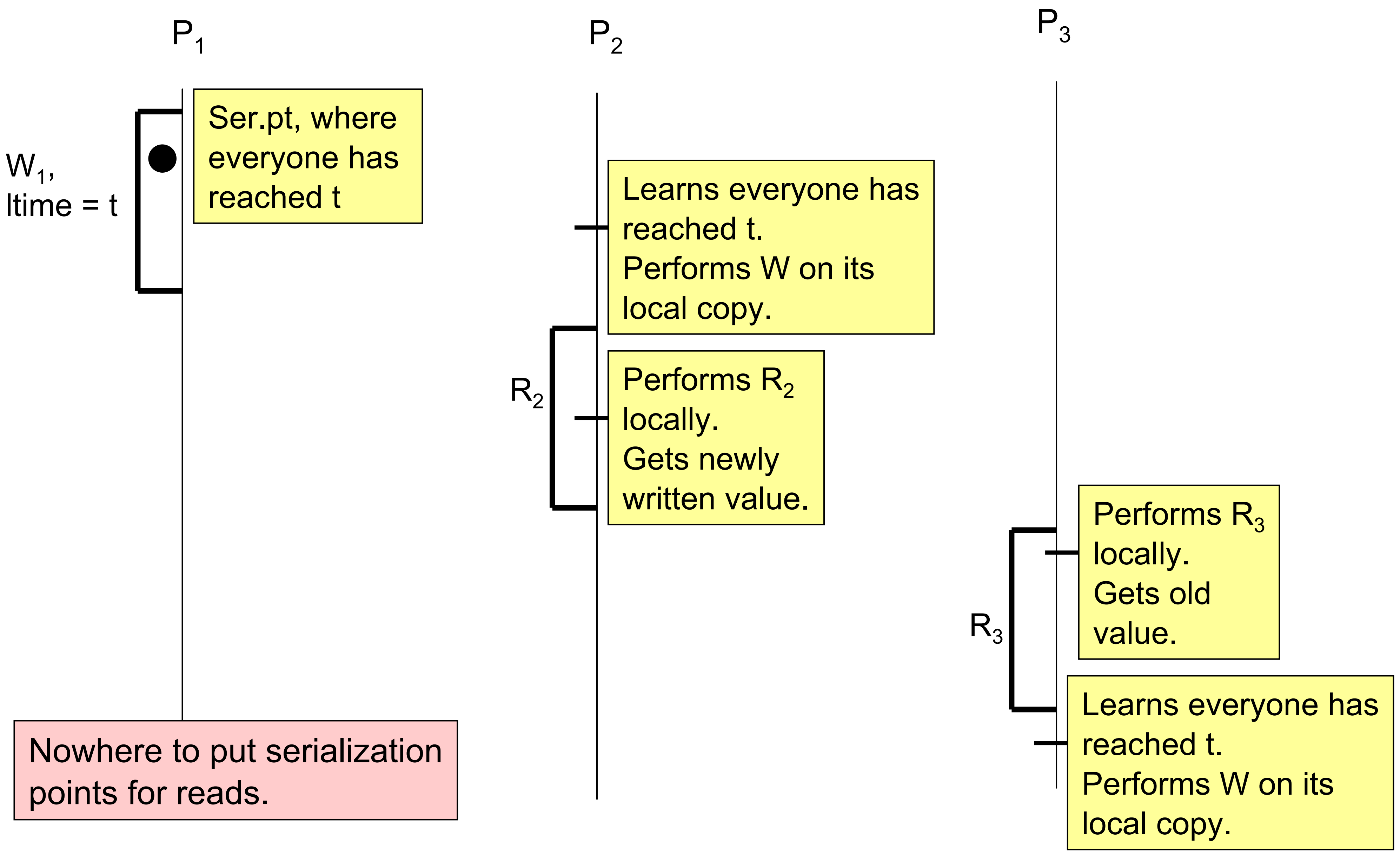
Safety, cont'd

- Safety: Atomicity (each operation "appears to be performed" at a point in its interval, as in a centralized machine):
  - Each process applies operations in the same (logical time) order.
  - Define serialization point for each operation π to be the earliest point when all processes have reached the logical time t of π's bcast event.
  - This point is within π's interval.
  - The order of the serialization points is the same as the logical time order, which is the same as the order in which the operations are performed on all copies.
  - So, responses are consistent with the order of serialization points.
  - That is, it looks to all the users as if the operations occurred at their serialization points---as in a centralized machine.

Special handling of reads

- Don't bcast---just perform them locally.
- Now, doesn’t satisfy atomicity.
- Satisfies weaker property, sequential consistency.

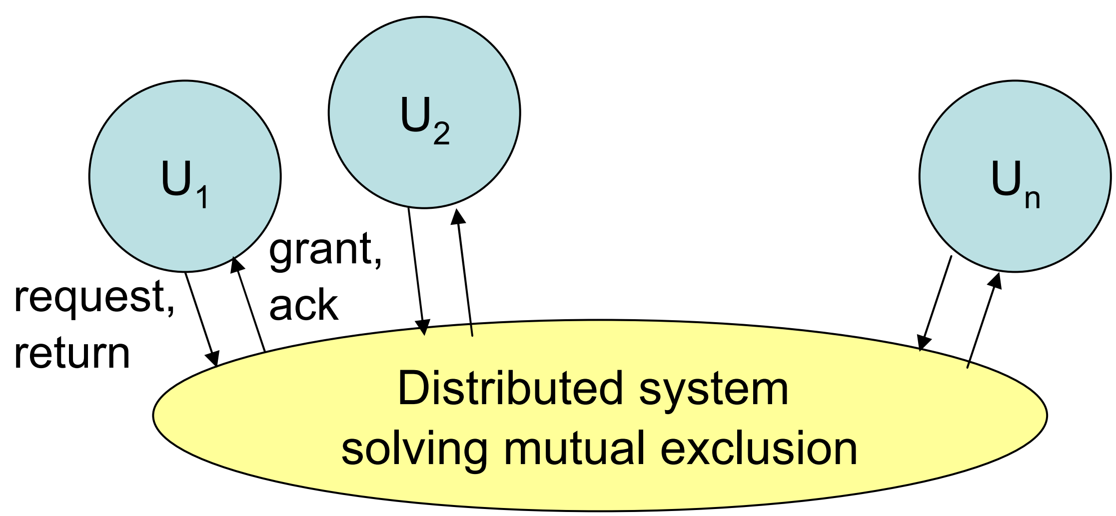
No serialization points…



Application of RSM: Distributed mutual exclusion

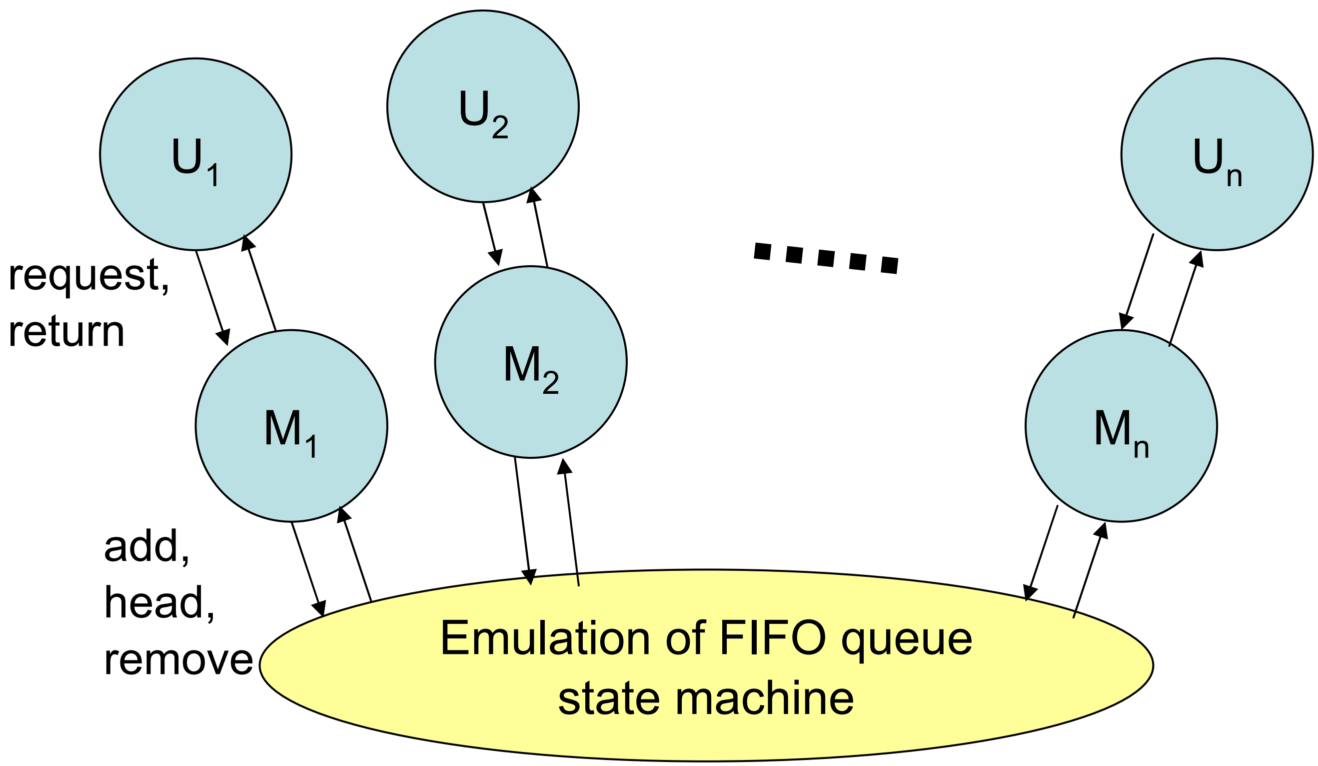
- Distributed mutual exclusion problem:
  - Users at different locations submit requests for a resource from time to time.
  - System grants requests, so that:
    - No two users get the resource at the same time, and
    - Every request is eventually granted.
  - Users must return the resource.

- Solve distributed mutual exclusion using a distributed simulation of a centralized state machine.
- See book, p. 609-610.



Distributed mutual exclusion

- Use one emulated FIFO queue state machine:
  - State contains a FIFO queue of process indices.
  - Operations:
    - add(i), i a process index: Adds i to end of queue.
    - head: Returns head of queue, or “empty”.
    - remove(i): Removes all occurrences of i from the queue.



Distributed mutual exclusion

- Given (emulated) shared queue, mutex processes cooperate to implement mutual exclusion.
- Process i operates as follows:
  - To request the resource:
    - Invoke add(i), adding i to the end of the queue.
    - Repeatedly invoke head, until the response yields index i.
    - Then grant the resource to its user.
  - To return the resource:
    - Invoke remove(i).
    - Return ack to user.
- Complete distributed mutual exclusion algorithm:
  - Use Lamport's logical time algorithm to give logical times.
  - Use RSM algorithm, based on logical time, to emulate the shared queue.
  - Use mutex algorithm above, based on shared queue.

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 them values from a partially-ordered set P.

- Weak logical time:
  - Properties 1-4 same as before---the only difference is that the ltimes don't need to be totally ordered.
- 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 $\leq$ 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.
- That is, 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′.

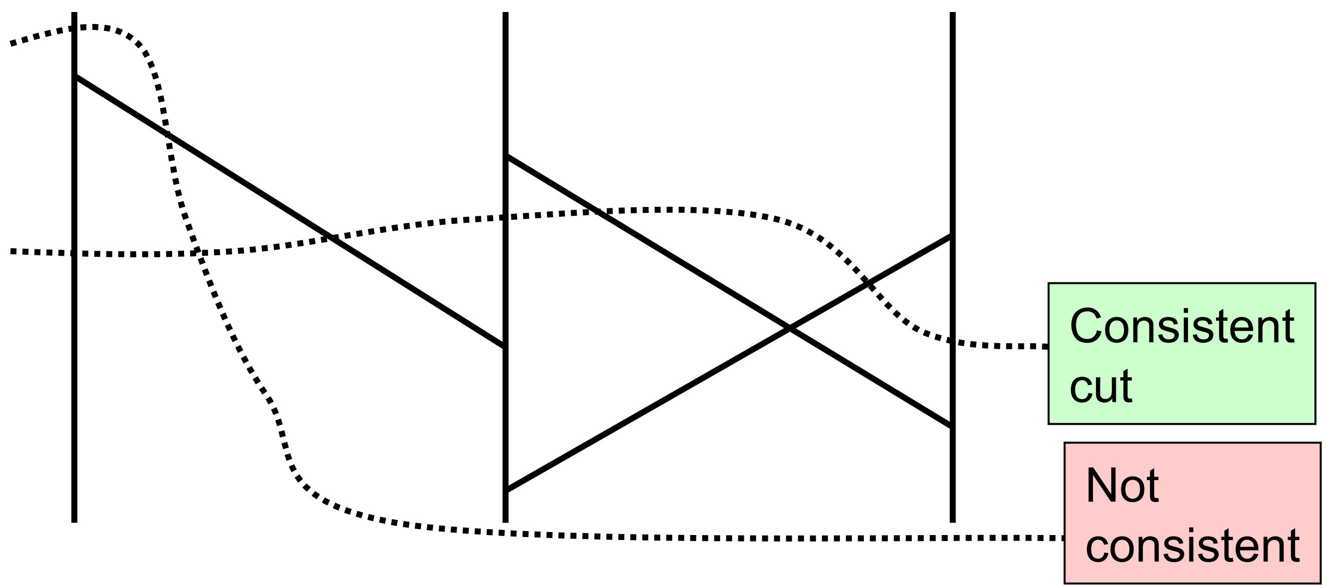
Proof of Theorem 2

- 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 $\leq V'$.

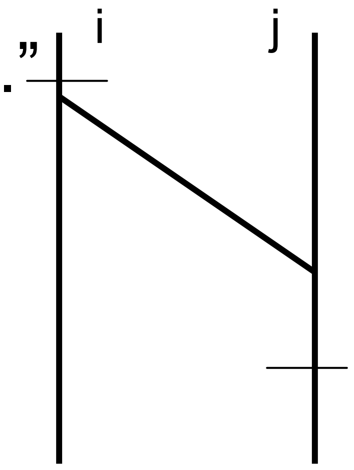
Another theorem about vector timestamps [Mattern]

- 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 exactly at i's cut-point.
- Then $V = \max(V_1, V_2,\ldots,V_n)$ gives the maximum information obtainable by combining everyone knowledge at the 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; in that case, j would have more information 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 time).
  - Use this to check correctness properties for the execution, e.g., invariants.

Next time

- Consistent global snapshots
- Stable property detection
- Reading: Chapter 19

6.852J / 18.437J Distributed Algorithms
Fall 2009