System timeline; equivalent representation

Edge B -> C implies t(B) < t(C)

Move vertices so that higher number vertices appear higher.
All the vertex numbers go upward (forwards in time);

Linear representation:
System computation
P0; P1; Q2; P3; R4; R5; P6;…

System computation represented by an interleaving of process computations.
Theorem 1

Any total order consistent with the partial order, i.e., any numbering of integers 0, 1, 2, 3, … in which all numbers are from lower-numbered to higher-numbered vertices, is a valid system computation.
Theorem 1

Given the log of each process, i.e., given the process computation of each process:
The events (state transitions and messages) **could have** occurred in this total order.
Proof Outline

1. Prove process computations in new ordering are valid computations.
2. Prove channel computations in new ordering are valid computations.
Proof Outline: Part 1, Process Computations

- **Process Computations are Valid**: Each state transition (old state, message received, new state, messages sent) in a process computation depends only on:
  1. the state before the transition and
  2. The message (if any) that arrives at that transition.

- The state before the transition and the message (if any) received at the transition remain unchanged by the new (total) ordering.
Proof Outline: Part 1. Pictorial Explanation

Move vertices so that higher number vertices appear higher.

This doesn’t change each vertical line.
Proof Outline, Part 2: Channel Computations

- Channel Computations are Valid:
- A channel computation is valid if and only if the state of the channel is a queue of messages, where the only transitions are:
  1. **New message sent on the channel:** Append a message to the tail of the queue.
  2. **Message from the channel delivered to destination:** A message at the head of the queue is deleted.
Proof Outline: Channel Computations

- A channel computation is valid if and only if:
  1. A message is delivered only after it is sent.
  2. Messages are delivered in the order in which they were sent:
     i.e., the sequence of messages delivered is a prefix of the sequence of messages sent.
The reordering ensures that for each channel:

1. A message is delivered only after it is sent.
2. Messages are delivered in the order in which they were sent.
The rescheduling of events:
1. doesn’t change the crossing of message lines for any channel.
2. Doesn’t go backwards in time
Corollaries to the Theorem

- These corollaries will help us develop important algorithms.
Local Times on Process Computations

Processes take local snapshots at different times.

What is the criterion that guarantees that these points on local process computations are consistent?
Consistency of Local Snapshots

There exists a system computation in which all events before snapshot occur before all events after snapshot.
Pictorial Explanation: Consistency means the line can be straightened
Theorem

- The local snapshots are consistent if and only if for all messages $m$ received before a local snapshot, the message $m$ was sent before the local snapshot.

- Proof: Follows immediately from basic theorem given earlier.
Local Times of Global Snapshots

Processes take local snapshots at different times that satisfy the criterion.

Events BEFORE snapshot

Events AFTER snapshot
Key Property of Distributed Snapshots

- All edges between “AFTER” snapshot events and “BEFORE” snapshot events are directed from BEFORE snapshot event to AFTER snapshot event.
Pictorial Explanation

Events
AFTER snapshot

All lines crossing barrier go from “before” to “after”

Events
BEFORE snapshot
Pictorial Explanation

Events AFTER snapshot

No line crosses barrier from “after” to “before”

Events BEFORE snapshot
Theorem

- Let \( c_{\text{numberSent}} \) be the number of messages sent by a process \( p \) along an outgoing channel \( c \) before \( p \) takes its (local) snapshot.
- Let \( c_{\text{numberReceived}} \) be the number of messages received by a process \( q \) along an incoming channel \( c \) before \( q \) takes its (local) snapshot.
- The local snapshots are consistent if and only if:
  For all channels \( c \): \( c_{\text{numberSent}} \geq c_{\text{numberReceived}} \).

- Proof: Follows immediately from basic theorem given earlier.
“Client” distributed system: “user processes”
OS processes use same channels: They can “see” but not interfere with user processes.
OS processes use same channels:
They can send OS messages that are trapped by receiving OS processes but not sent on to user processes.
Distributed Operating System: The Operational Model as a Protocol Stack

User 1

OS 1

User 2

OS 2

OS process 2 responsible for user process 2

Location 2
Question

- Develop an algorithm by which a distributed operating system can pick points in each process’ (local) computation, so that the set of points is consistent.
Question: Develop Algorithm that Ensures all Edges are Directed Outward
Incorrect Global Snapshot

Events AFTER snapshot

Events BEFORE snapshot

No line crosses barrier from “after” to “before”
Global Snapshot Algorithm

- Key question: How can we avoid the situation shown in the last slide?
- In this bad situation, messages sent by a process after it took its snapshot was received by another process before it took its snapshot.
Global Snapshot Algorithm

- Avoid this bad situation, messages sent by a process after it took its snapshot was received by another process before it took its snapshot.

- SOLUTION:
  1. When a process takes its snapshot, it sends a special message (let’s call it a marker message) along each of its outgoing channels.
  2. When a process receives a marker message, it takes its snapshot if it has not done so.
Theorem

- Every message sent by a process P after P took its snapshot is received by a process Q after Q took its snapshot.
Proof of Correctness

- Consider a message $M$ sent by a process $P$ along one of its outgoing channels $c$ to a process $Q$, where $M$ was sent by $P$ after $P$ took its snapshot.
- From rule 1 of the algorithm, $P$ sent a marker message before sending $M$ along channel $c$.
- Since messages along a channel are delivered in the order sent, the marker message sent along $c$ was received by $Q$ before $M$ was received by $Q$. 
Proof of Correctness

- From rule 2 of the algorithm, process Q took its snapshot when it received the *marker* along channel c, if Q had not taken its snapshot already.
- Hence Q took its snapshot before receiving M.
Termination of Global Snapshot Algorithm

- One of more processes initiates the algorithm.
- If the graph representation of the system is connected then every process receives a marker eventually, and takes its snapshot, and exactly one marker is sent along every channel.
- Prove by induction on the number of hops from any process that initiates the computation.
Channel States

- Each process snapshots its own state.
- How can processes collaborate in determining channel states?
Channel States

- Each process snapshots its own state.
- How can processes collaborate in determining channel states?
- The state of a channel c from a process P to a process Q is recorded as the sequence of messages received by Q after Q took its snapshot and before it receives a marker along channel c.
- Prove this.
Logical Clocks

- Design an algorithm where each process maintains a local clock such that if each process takes a snapshot at the same time on the local clock then the collection of local process and channel snapshots is a correct global snapshot.

- Prove correctness
Next Class

- Systematic ways of designing distributed algorithms.
- Designing around invariant and progress properties.
- Introduction to diffusing computations
- Discussion of the midterm Quiz questions.