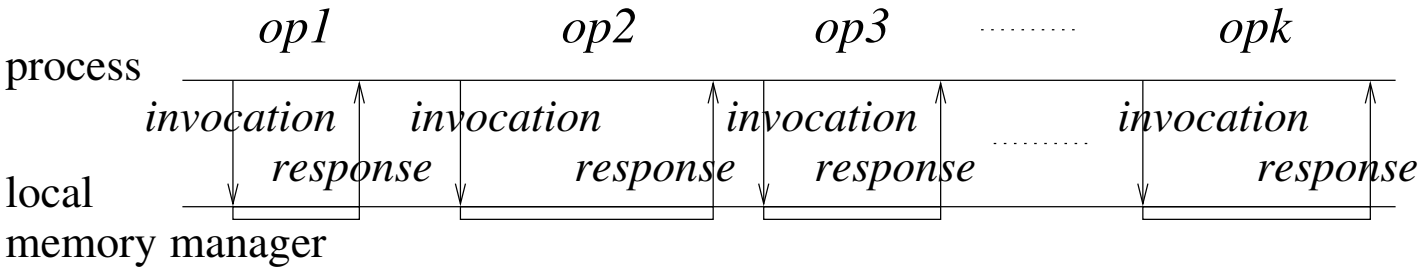
Memory Coherence

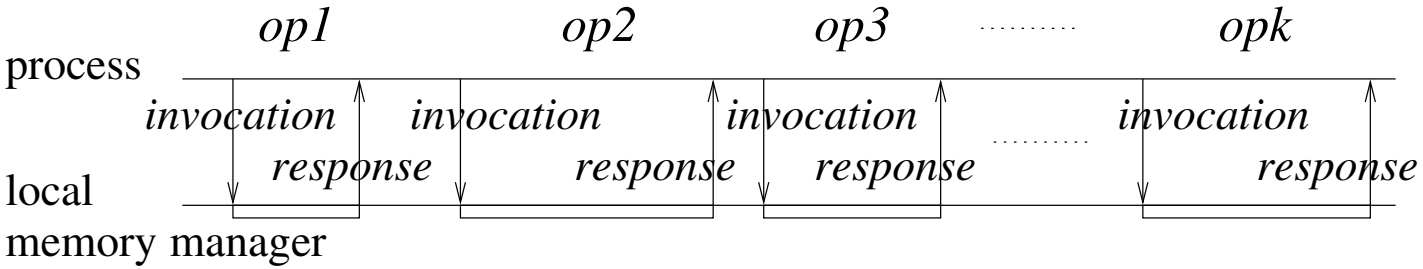
- s_i memory operations by P_i
- $(s_1 + s_2 + \ldots s_n)!/(s_1!s_2! \ldots s_n!)$ possible interleavings
- Memory coherence model defines which interleavings are permitted
- Traditionally, Read returns the value written by the most recent Write
- "Most recent" Write is ambiguous with replicas and concurrent accesses
- DSM consistency model is a *contract* between DSM system and application programmer



Strict Consistency/Linearizability/Atomic Consistency

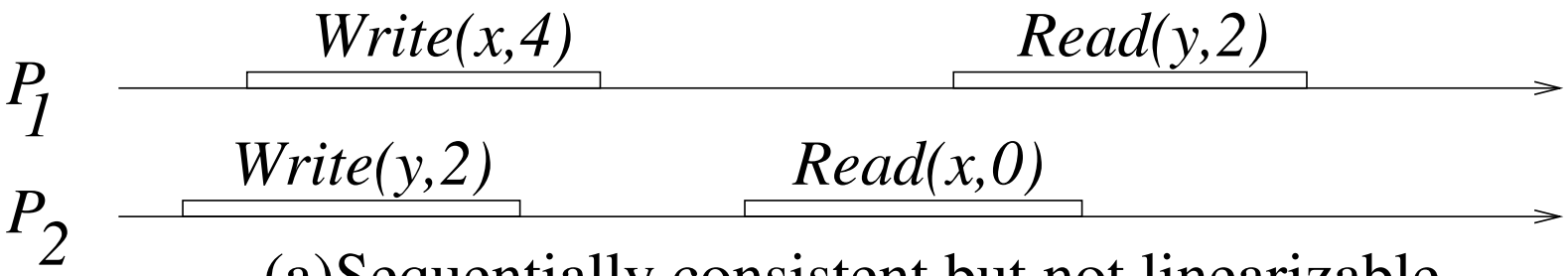
Strict consistency

1. A Read should return the most recent value written, per a global time axis. For operations that overlap per the global time axis, the following must hold.
2. All operations appear to be atomic and sequentially executed.
3. All processors see the same order of events, equivalent to the global time ordering of non-overlapping events.

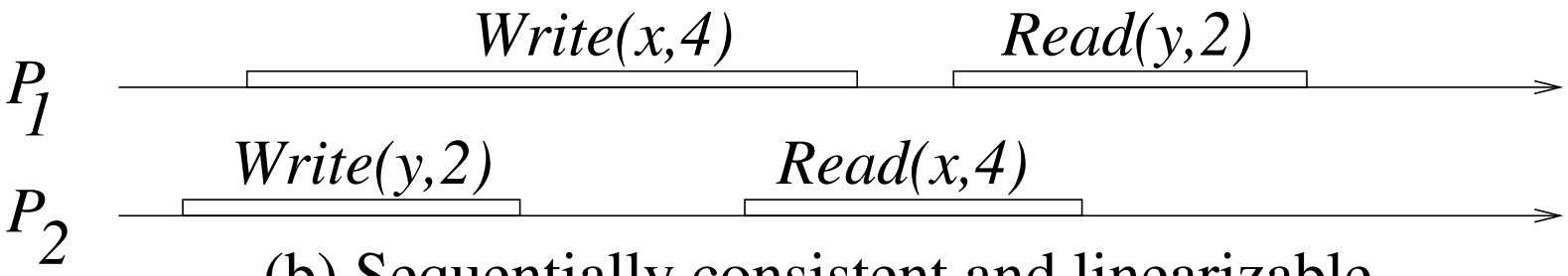


Sequential invocations and responses to each Read or Write operation.

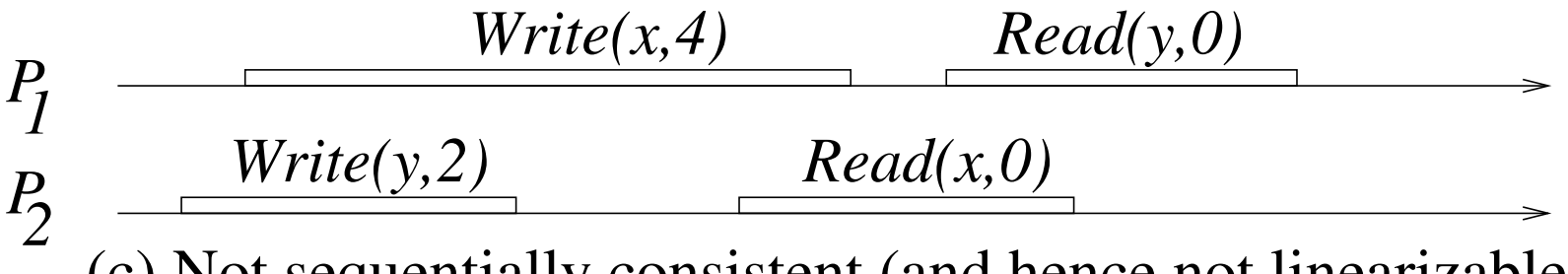
Strict Consistency / Linearizability: Examples



(a)Sequentially consistent but not linearizable



(b) Sequentially consistent and linearizable



(c) Not sequentially consistent (and hence not linearizable

Initial values are zero. (a),(c) not linearizable. (b) is linearizable

Linearlzability: Implementation

- Simulating global time axis is expensive.
- Assume full replication, and total order broadcast support.

(shared var)
int: x;

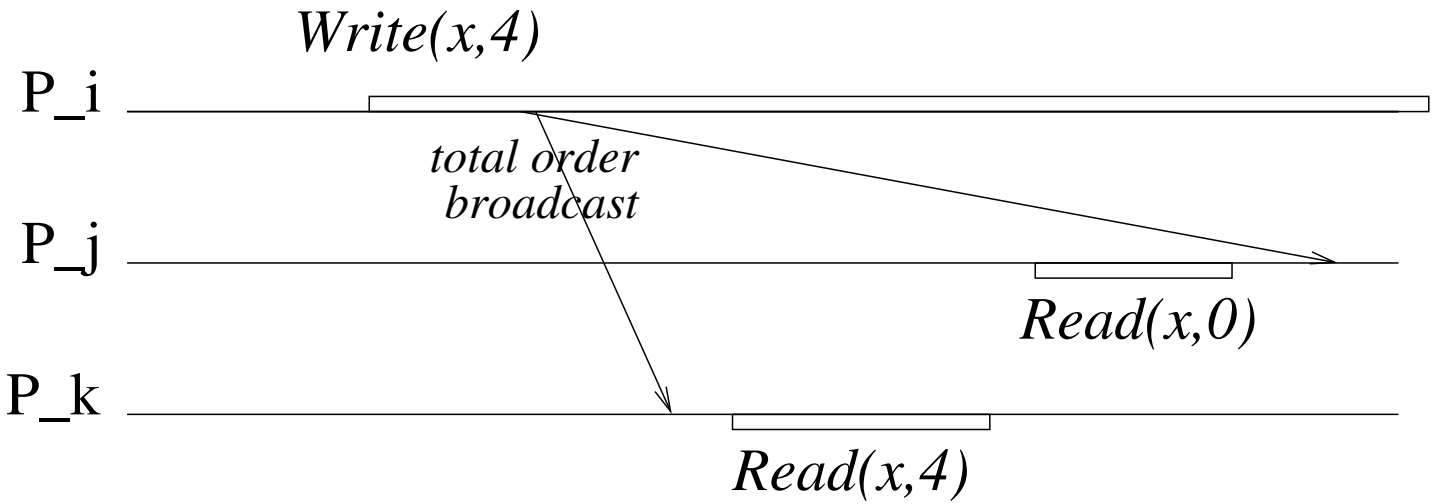
(1) When the Memory Manager receives a *Read* or *Write* from application:
(1a) **total_order_broadcast** the *Read* or *Write* request to all processors;
(1b) **await** own request that was broadcast;
(1c) **perform** pending response to the application as follows
(1d) **case** *Read*: return value from local replica;
(1e) **case** *Write*: write to local replica and return ack to application.

(2) When the Memory Manager receives a **total_order_broadcast**(*Write*, x, *val*) from network:
(2a) **write** *val* to local replica of x.

(3) When the Memory Manager receives a **total_order_broadcast**(*Read*, x) from network:
(3a) **no operation**.

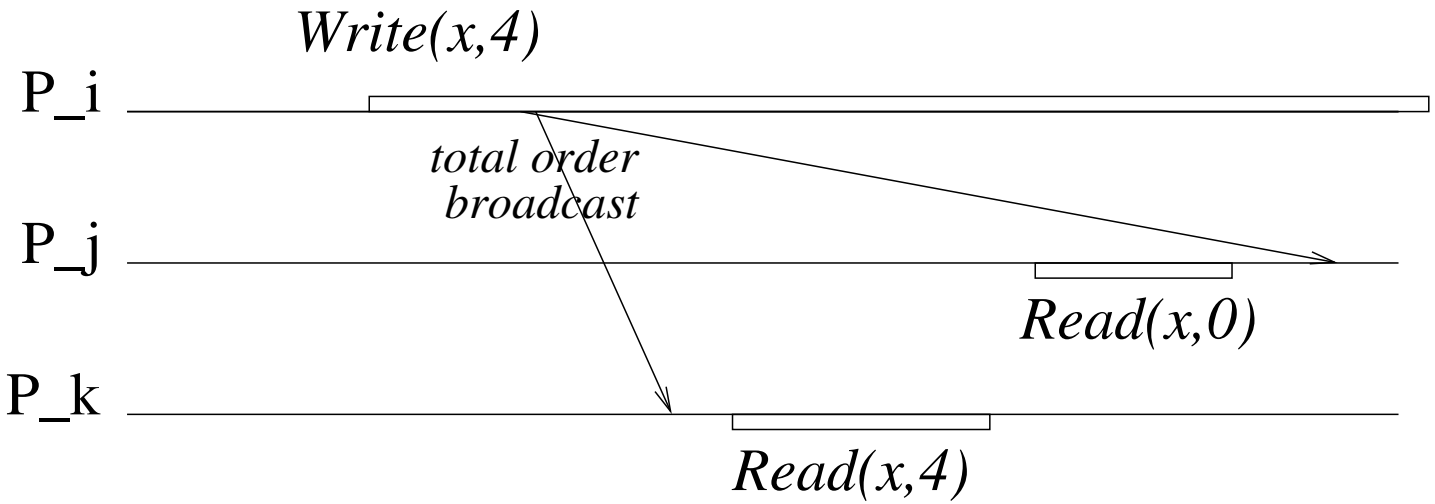
Linearizability: Implementation (2)

- When a Read in simulated at other processes, there is a no-op.
- Why do Reads participate in total order broadcasts?
- Reads need to be serialized w.r.t. other Reads and all Write operations. See counter-example where Reads do not participate in total order broadcast.



Linearizability: Implementation (2)

- When a Read in simulated at other processes, there is a no-op.
- Why do Reads participate in total order broadcasts?
- Reads need to be serialized w.r.t. other Reads and all Write operations. See counter-example where Reads do not participate in total order broadcast.



Sequential Consistency

Sequential Consistency.

- The result of any execution is the same as if all operations of the processors were executed in *some* sequential order.
- The operations of each individual processor appear in this sequence in the local program order.

Any interleaving of the operations from the different processors is possible. But all processors must see *the same* interleaving. Even if two operations from different processors (on the same or different variables) do not overlap in a global time scale, they may appear in reverse order in the *common* sequential order seen by all. See examples used for linearizability.

Sequential Consistency

Only Writes participate in total order BCs. Reads do not because:

- all consecutive operations by the same processor are ordered in that same order (no pipelining), and
- *Read* operations by different processors are independent of each other; to be ordered only with respect to the *Write* operations.

- Direct simplification of the LIN algorithm.
- Reads executed atomically. Not so for Writes.
- Suitable for Read-intensive programs.

Sequential Consistency using Local Reads

(shared var)
int: x;

(1) When the Memory Manager at P_i receives a *Read* or *Write* from application:
(1a) **case** *Read*: **return** value from local replica;
(1b) **case** *Write(x,val)*: **total_order_broadcast**$_i$(*Write(x,val)*) to all processors including itself.

(2) When the Memory Manager at P_i receives a **total_order_broadcast**$_j$(*Write*, x, *val*) from networl
(2a) **write** *val* to local replica of x;
(2b) **if** $i = j$ **then return** ack to application.

Sequential Consistency using Local Writes

(shared var)
int: x;

(1) When the Memory Manager at P_i receives a $Read(x)$ from application:
(1a) **if** $counter = 0$ **then**
(1b) **return** x
(1c) **else** Keep the *Read* pending.

(2) When the Memory Manager at P_i receives a *Write(x,val)* from application:
(2a) $counter \longleftarrow counter + 1$;
(2b) **total_order_broadcast**$_i$ the $Write(x, val)$;
(2c) **return** ack to the application.

(3) When the Memory Manager at P_i receives a **total_order_broadcast**$_j$(*Write*, x, val) from networ
(3a) **write** val to local replica of x.
(3b) **if** $i = j$ **then**
(3c) $counter \longleftarrow counter - 1$;
(3d) **if** ($counter = 0$ and any *Read*s are pending) **then**
(3e) **perform** pending responses for the *Read*s to the application.

Locally issued Writes get acked immediately. Local Reads are delayed until the locally preceding Writes have been acked. All locally issued Writes are pipelined.

Causal Consistency

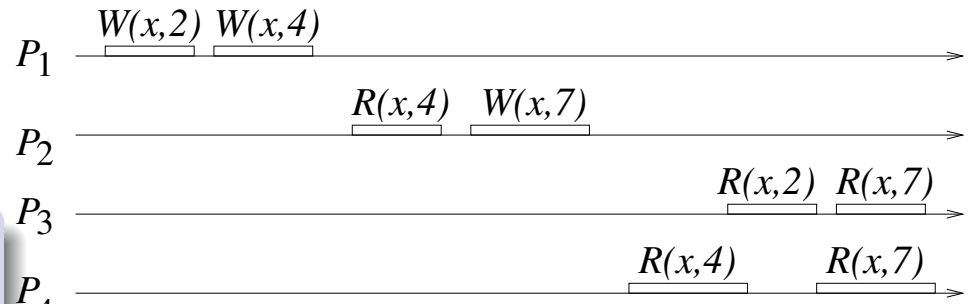
In SC, all Write ops should be seen in common order.

For *causal consistency*, only causally related Writes should be seen in common order.

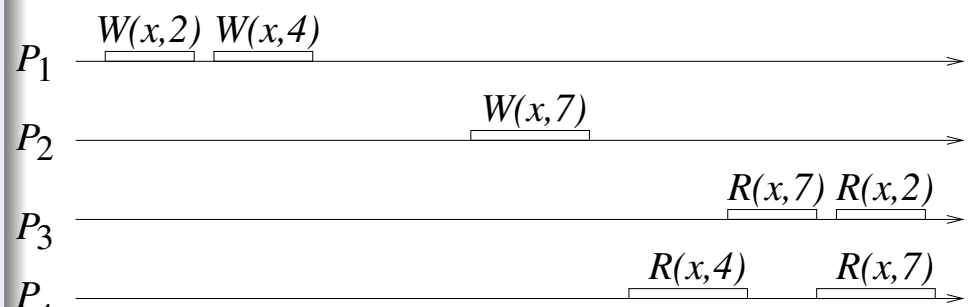
Causal relation for shared memory systems

- At a processor, local order of events is the causal order
- A Write causally precedes Read issued by another processor if the Read returns the value written by the Write.
- The transitive closure of the above two orders is the causal order

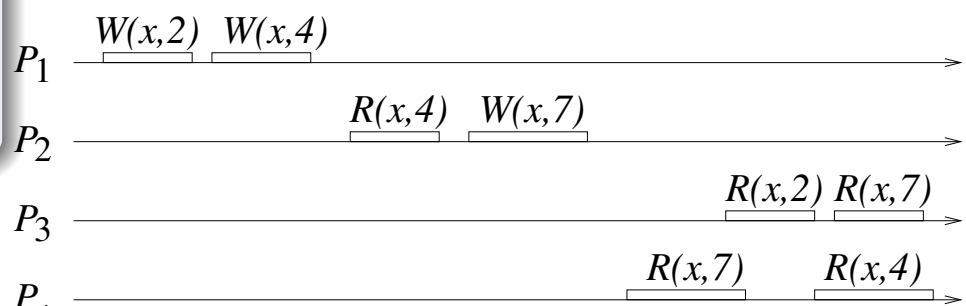
Total order broadcasts (for SC) also provide causal order in shared memory



(a)Sequentially consistent and causally consistent



(b) Causally consistent but not sequentially consistent



(c) Not causally consistent but PRAM consistent

Pipelined RAM or Processor Consistency

PRAM memory

Only Write ops issued by the same processor are seen by others in the order they were issued, but Writes from different processors may be seen by other processors in different orders.

PRAM can be implemented by FIFO broadcast? PRAM memory can exhibit counter-intuitive behavior, see below.

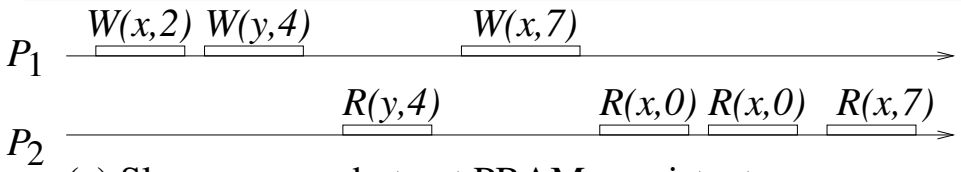
(shared variables)
int: x, y;

Process 1	Process 2
...	...
(1a) $x \longleftarrow 4$;	(2a) $y \longleftarrow 6$;
(1b) **if** $y = 0$ **then kill**(P_2).	(2b) **if** $x = 0$ **then kill**(P_1).

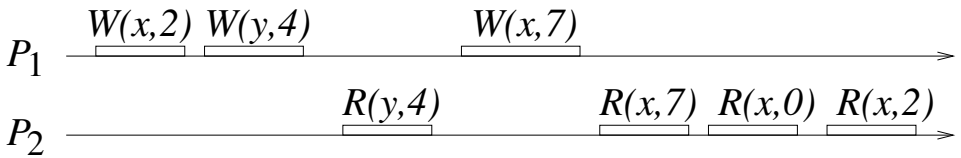
Slow Memory

Slow Memory

Only Write operations issued by the same processor and to the same memory location must be seen by others in that order.

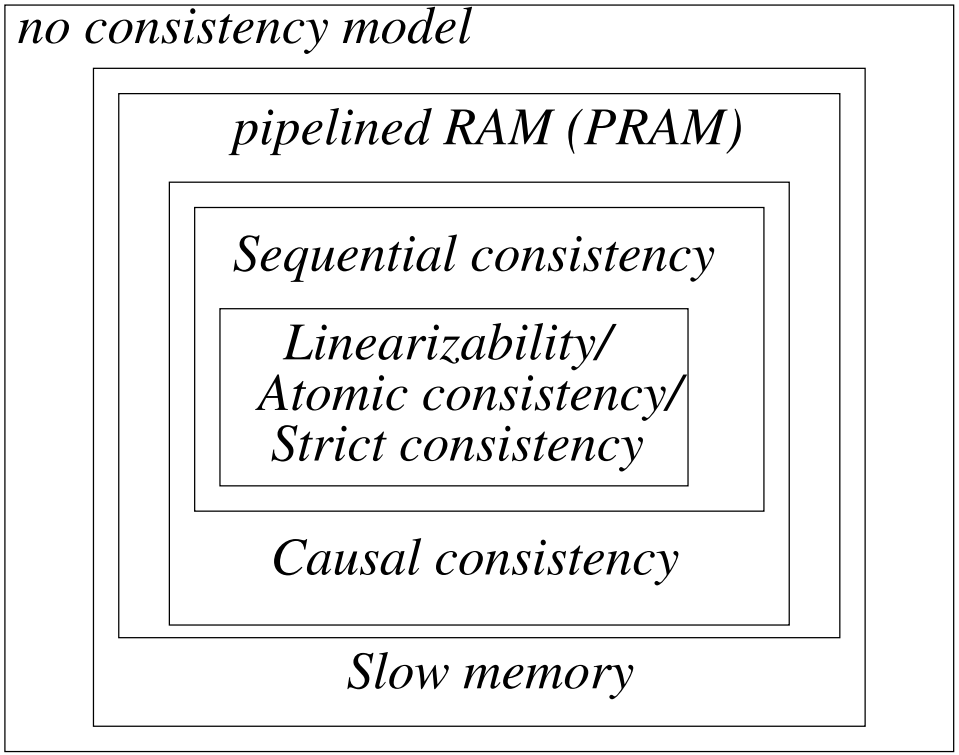


(a) Slow memory but not PRAM consistent



(b) Violation of slow memory consistency

Hierarchy of Consistency Models



Synchronization-based Consistency Models: Weak Consistency

- Consistency conditions apply only to special "synchronization" instructions, e.g., barrier synchronization
- Non-sync statements may be executed in any order by various processors.
- E.g.,weak consistency, release consistency, entry consistency

Weak consistency:

All Writes are propagated to other processes, and all Writes done elsewhere are brought locally, at a sync instruction.

- Accesses to sync variables are sequentially consistent
- Access to sync variable is not permitted unless all Writes elsewhere have completed
- No data access is allowed until all previous synchronization variable accesses have been performed

Drawback: cannot tell whether beginning access to shared variables (enter CS), or finished access to shared variables (exit CS).

Synchronization based Consistency Models: Release Consistency and Entry Consistency

Two types of synchronization Variables: *Acquire* and *Release*

Release Consistency

- *Acquire* indicates CS is to be entered. Hence all *Writes* from other processors should be locally reflected at this instruction
- *Release* indicates access to CS is being completed. Hence, all Updates made locally should be propagated to the replicas at other processors.
- *Acquire* and *Release* can be defined on a subset of the variables.
- If no CS semantics are used, then Acquire and Release act as barrier synchronization variables.
- Lazy release consistency: propagate updates on-demand, not the PRAM way.

Entry Consistency

- Each ordinary shared variable is associated with a synchronization variable (e.g., lock, barrier)
- For Acquire /Release on a synchronization variable, access to only those ordinary variables guarded by the synchronization variables is performed.