# Lecture 15: Distributed Shared Memory

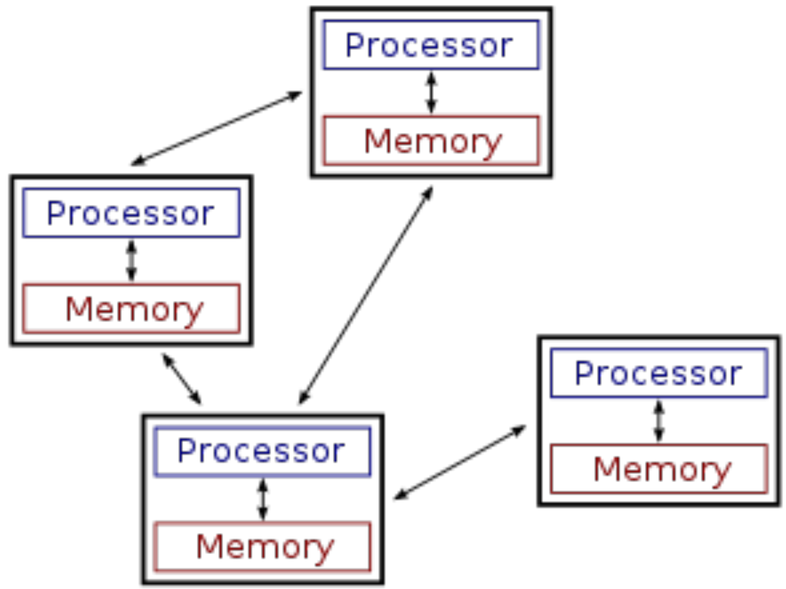
CS 539 / ECE 526

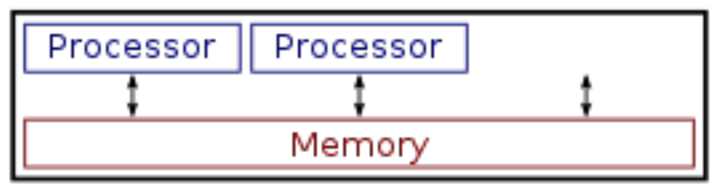
Distributed Algorithms

Some slides are borrowed from Jennifer Welch's slides of CSCE 668 at Texas A&M

# Communication Model

- How do processes communicate?
- Message passing
  - More fundamental
- Shared memory
  - More convenient




https://en.wikipedia.org/wiki/Distributed_computing

# Shared Memory

- Less fundamental, an abstraction/illusion
  - Hard to build a large monolithic memory with many read/write ports
  - Memory is also a component that receive commands and returns responses, via msgs!
- But considered a more convenient (familiar) programming model

# Today

- Distributed Shared Memory (DSM) algo: build shared memory from msg passing
  - What properties do we want?
  - In what model?
  - How?

# Properties of Shared Memory?

- Mimic single-process memory interface
  - $Read_i(X)$ … $Return_i(v)$
  - $Write_i(X, v)$ … $Ack_i()$
  - A read returns the value of the *most recent* write
- What about overlapping operations?
  - Read / write operations are not instantaneous
  - [$Read_1$ … $Return_1$] [$Read_3$ … $Return_3$]
  - [$Write_2$ … $Ack_2$]

# Outline

- Memory consistency model: specify desired behaviors of shared memory
  - Linearizability (atomic consistency)
  - Sequential consistency
- Algorithms for DSM
  - Total-order broadcast (atomic broadcast)
  - ABD (Attiya, Bar-Noy, Dolev)

# Linearizability (informal)

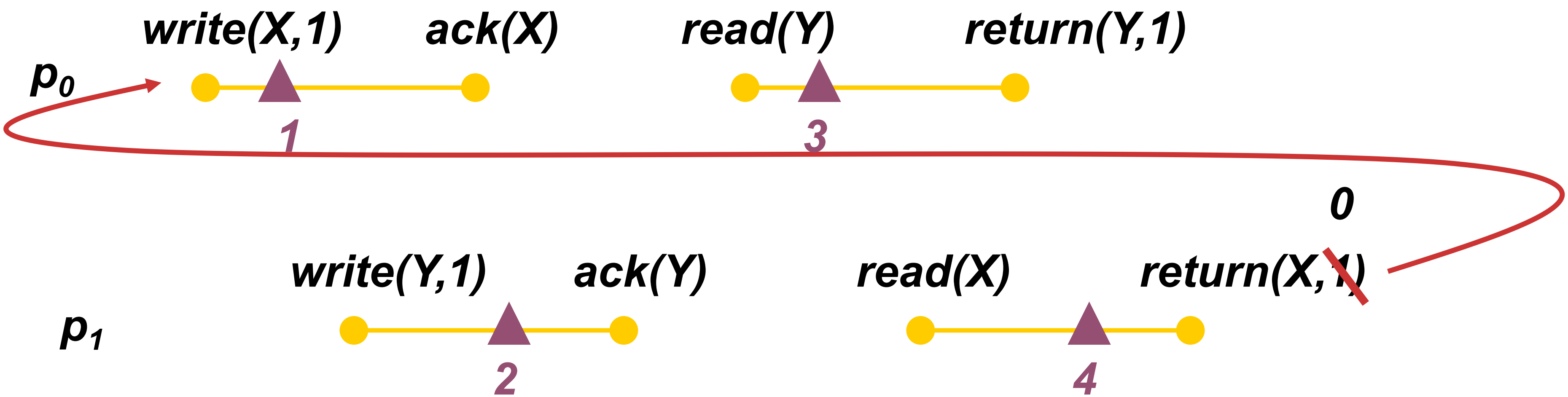
- Illusion that each op is instantaneous
  - Occurs at some point within its start/end
  - Also called *atomic consistency*: ops cannot be further divided
- Respect the real-time ordering of non-overlapping operations

# Linearizability (formal)

- Let S be a sequence of operation invocations and responses. S satisfies linearizability if there exists a permutation S' of S such that
  - Each op is immediately followed by its response
  - Each read of X returns the preceding write to X
  - If op1 ends before op2 starts in S, then op1 occurs before op2 in S'

# Linearizability Examples

Suppose there are two shared variables, $X$ and $Y$, both initially 0




Is this sequence linearizable? Yes - brown triangles.

What if $p_1$'s read returns 0? No - see arrow.

# Sequential Consistency (informal)

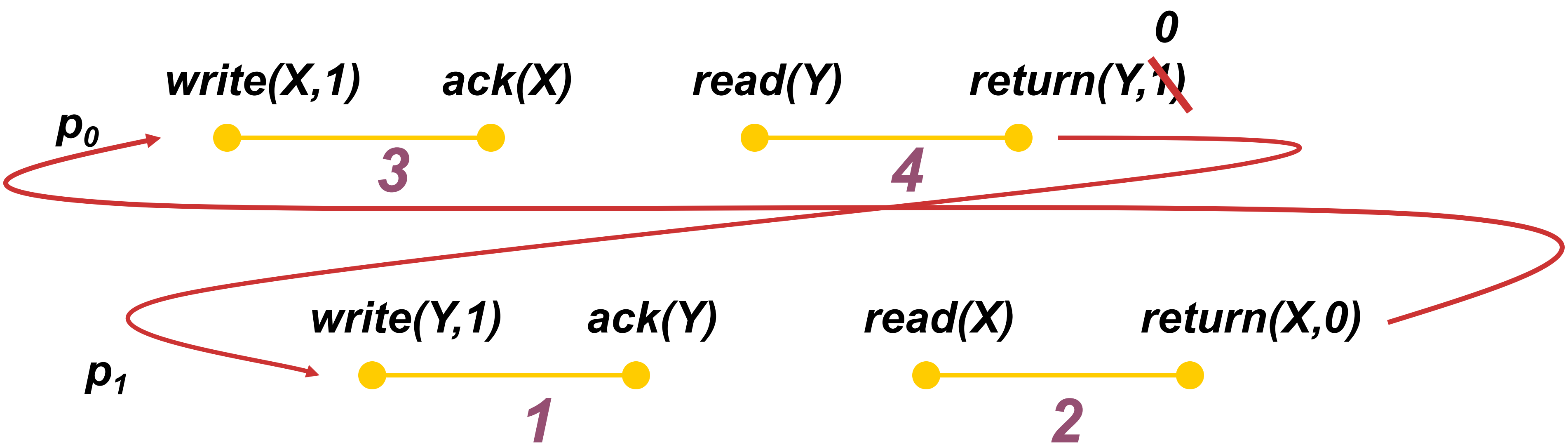
- As if each op is instantaneous
  - Occurs at some point within its start/end
- Respect the real-time ordering of non-overlapping ops at the same process

# Sequential Consistency (formal)

- Let S be a sequence of operation invocations and responses. S satisfies seq. consistency if there exists a permutation S’ of S such that
  - Each op is immediately followed by its response
  - Each read of X returns the preceding write to X
  - If op1 ends before op2 starts in S at the same process, then op1 occurs before op2 in S’

# Sequential Consistency Examples

Suppose there are two shared variables, $X$ and $Y$, both initially 0




Is this sequence sequentially consistent? Yes - brown numbers.

What if $p_0$'s read returns 0? No - see arrows.

# Outline

- Memory consistency model: specify desired behaviors of shared memory
  - Linearizability (atomic consistency)
  - Sequential consistency
- Algorithms for DSM
  - Total-order broadcast (atomic broadcast)
  - ABD (Attiya, Bar-Noy, Dolev)

# Algorithm in Shared Memory

- What timing and fault models?
  - Asynchrony, because processes may get "distracted" for a very long time
    - E.g., interrupts
  - Fault-free or crash faults
    - No need for Byzantine in a multiprocessor

# Algorithm for Linearizable SM

- First idea: use a total-order broadcast!
  - Each process replicates the full memory
  - Op invoked: send a new request
  - Op finishes when request is decided

  - All processes see the same sequence of ops (consensus), so same correct read responses

# Proof of Linearizability

- First idea: use a total-order broadcast!
  - Each process replicates the full memory
  - Op invoked: send a new request
  - Op finishes when request is decided
- Proof: just need to construct a permutation S'
  - Each op is immediately followed by its response
  - Each read of X returns the preceding write to X
  - Respect real-time order of non-overlapping ops

# Proof of Linearizability

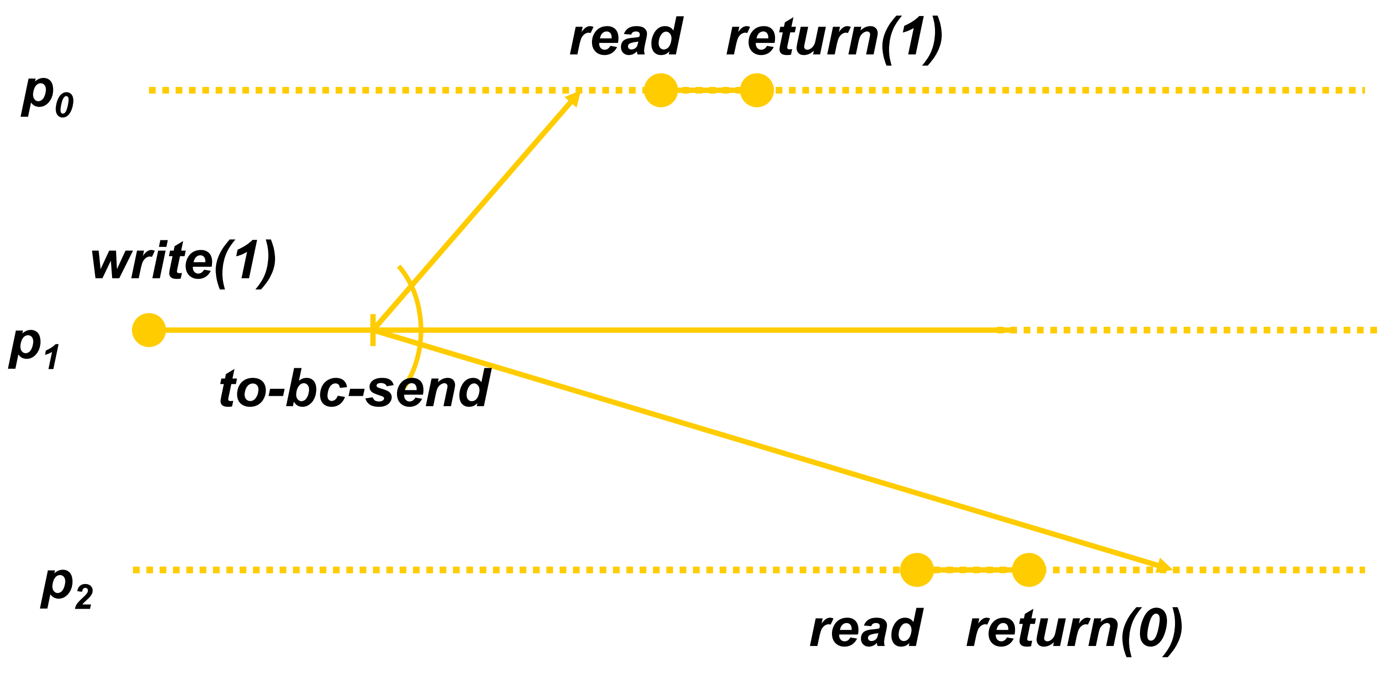
- Let S' be the total-order broadcast order
  - Each op is immediately followed by its response
    - Easily guaranteed by construction
  - Each read of X returns the preceding write to X
    - Easily guaranteed at each process given consensus
  - Respect real-time order of non-overlapping ops
    - Op1 ended = decided in TO-bcast
    - Op2 starts = appears in TO-bcast only later
    - Op2 is after Op1 in TO-bcast and hence in S'

# Why are Reads Broadcasted?

- Writes need to inform all processes to update all their local replicas
- But why do reads also need to be broadcasted to all processes?

# Why are Reads Broadcasted?

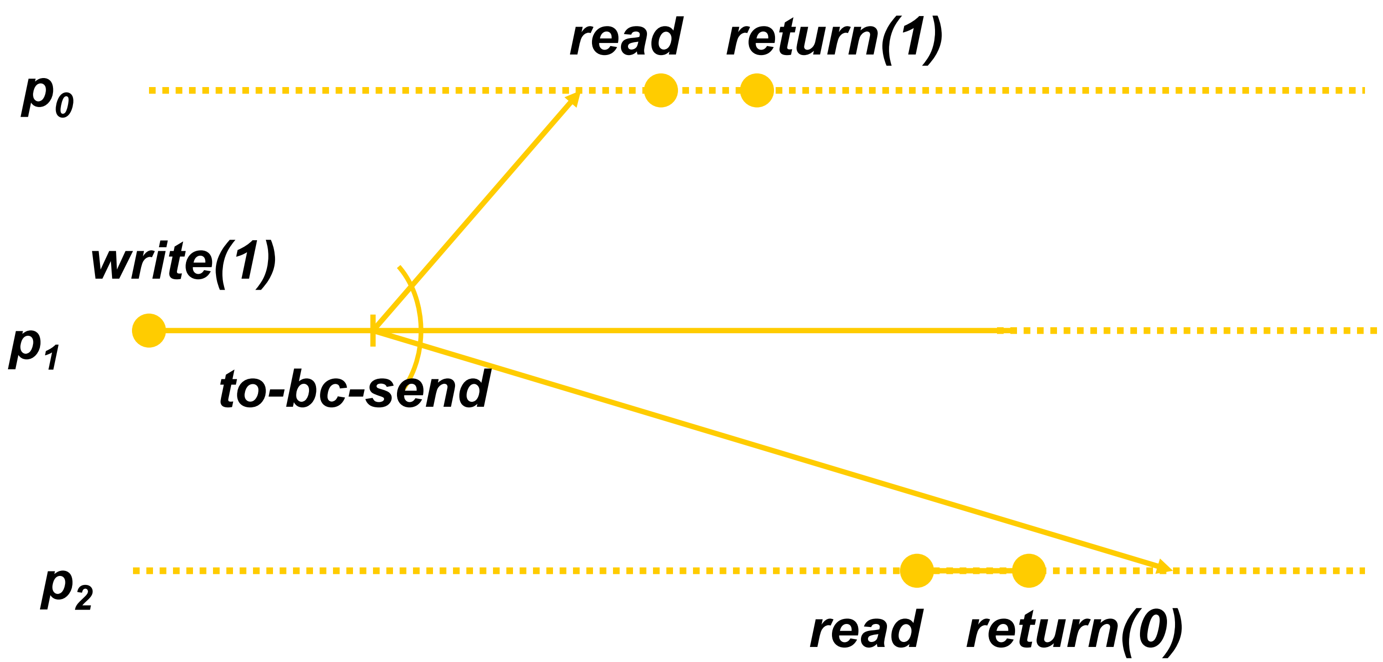
- If not, scenario below violates linearizability
  - Different processes in async TO-bcast may decide at (very) different times



# Algo for Seq. Consistency SM

- Sequential consistency is weaker, OK to have

  p2 read(0) — p1 write(1) — p0 read(1)



# Algo for Seq. Consistency SM

- First idea: use a total-order broadcast
- But only on writes!
  - Each process replicates the full memory
  - Write op invoked: send a new request
  - Write op finishes when request is decided
  - Read returns local replica right away

# Proof of Seq. Consistency

- Proof: just need to construct a permutation S'
  - Each op is immediately followed by its response
  - Each read of X returns the preceding write to X
  - Respect real-time order of ops **at same process**

# Proof of Seq. Consistency

- Naturally, put writes in TO-bcast order
- Put each read after latter of: (1) preceding op at that process and (2) the write op it reads from
  - Each op is immediately followed by its response
    - By construction
  - Each read of X returns the preceding write to X
    - Need to prove this
  - Respect real-time order of ops **at same process**
    - By construction of rule (1)

# Proof of Seq. Consistency

- Put each read after latter of: (1) preceding op at that process and (2) the write op it reads from
  - Each read of X returns the preceding write to X
    - Just need to show another write (W') to X does not fall between this read (R) and preceding write (W)
    - W(X, a) ~~W'(X, b)~~ R(X)=a
    - If W' is by the same process as R, R sees W'

# Proof of Seq. Consistency

- Put each read after latter of: (1) preceding op at that process and (2) the write op it reads from
  - Each read of X returns the preceding write to X
    - W(X, a) ~~W'(X, b)~~ O R(X)=a
    - If W' by another proc, exist O by same proc of R
    - If O is a write, W' before O by TO-bcast, R sees W'
    - If O is a read, consider earliest such read
    - Put there because O reads from W'
    - O sees W', so does R

# Outline

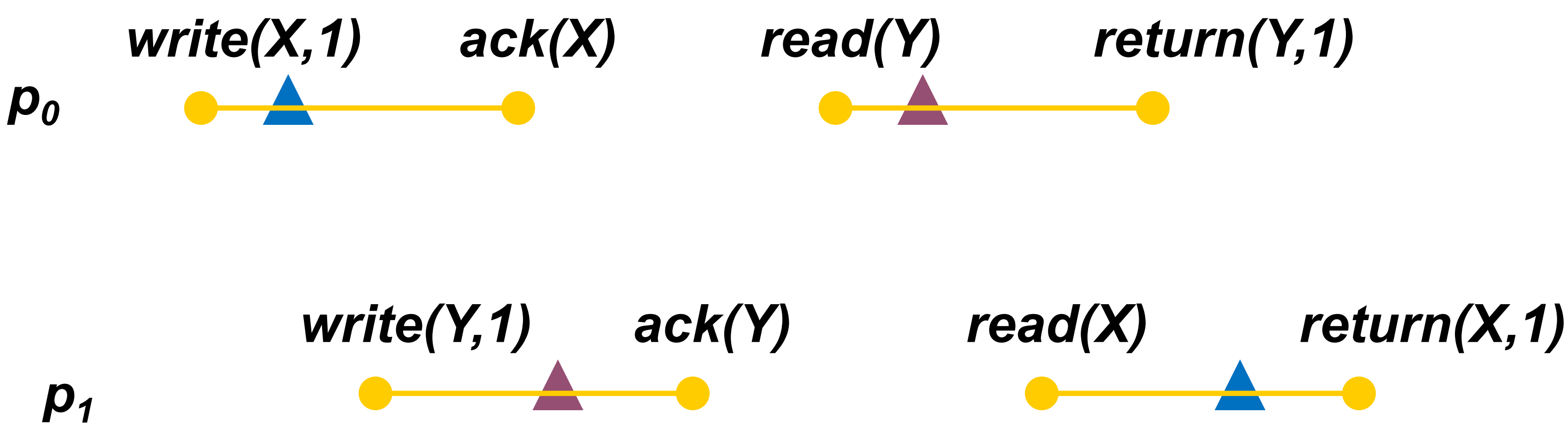
- Memory consistency model: specify desired behaviors of shared memory
  - Linearizability (atomic consistency)
  - Sequential consistency
- Algorithms for DSM
  - Total-order broadcast (atomic broadcast)
  - ABD (Attiya, Bar-Noy, Dolev)

# Algorithm for Linearizable SM

- Total-order broadcast: randomization needed in async to tolerate crashes
- Deterministic, async, tolerate $f < n/2$ crashes [Attiya, Bar-Noy, Dolev, 1995]
  - Contradict FLP?
  - No, just means linearizable SM is easier than consensus

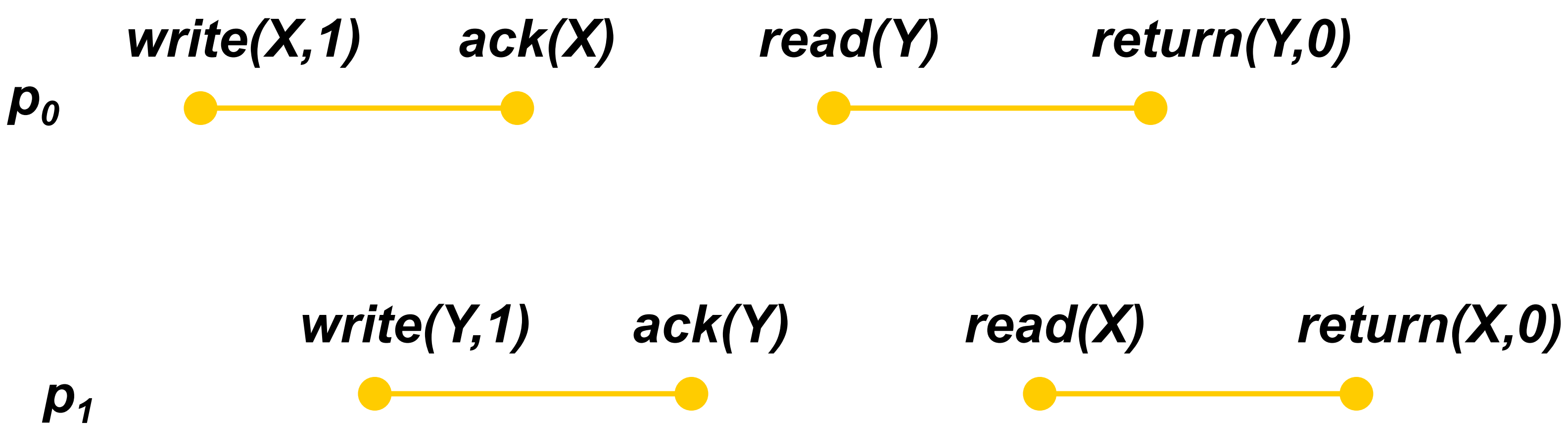
# Linearizability is Composable

- If each memory cell is linearizable, the entire memory is also linearizable
  - Rigorous proof omitted, intuition clear: merge sub-sequences, all conditions hold
- In fact, composable for any objects



# Sequential Consistency is NOT Composable

- Subsequences are sequentially consistent
  - Need not respect real-time ordering at different procs
- Put together, not sequentially consistent



# Back to ABD: Simplifications

- Linearizability is composable
- So we can focus on one memory cell (also called a register)

- For now, we assume only a single proc can write to the register (single writer)
  - Will extend to *n* writers later

# ABD Algorithm

- Augment with timestamp: reg = (val, ts=0)
- Replicate reg, one per process

- Upon write(v) operation:

```
t = ts = ts + 1
send "update, v, t" to all
                          update(v, t)   val = v if t > ts
                                         send back "ack, t"
wait until receiving majority acks
return      // write completes
```

# ABD Algorithm

- Augment with timestamp: reg = (val, ts=0)
- Replicate reg, one per process

- Upon write(v) operation:

```
increment ts
update(v, ts)
return      // write completes
```

# ABD Algorithm

- Augment with timestamp: reg = (val, ts=0)
- Replicate reg, one per process

- Upon read() operation:

  request local copy from all processes

  wait to collect majority copies

  (val, ts) = ($v_j$, $t_j$) with largest $t_j$

  **update(val, ts)**

  return val to reader

- Exercise: what goes wrong without update?

# Proof of Linearizability

- Need to find S' of all ops and responses s.t.
  - Each op immediately followed by its response
  - Each read returns preceding write
  - Real-time order of non-overlapping ops respected

- Naturally, order all operations by ts
  - Each write is associated with unique ts
    - Note again we have a single writing proc for now
  - Each read return value is associated with a ts
  - Ops with same ts: write before reads, earlier read before later read, remaining ties broken arbitrarily

# Proof of Linearizability

- S': order all ops by ts and attach responses
  - Write before reads, earlier read before later read, remaining ties broken arbitrarily

- By construction, each op followed by its response, and read returns preceding write

- It remains to show S' respects real-time order of non-overlapping ops

# Proof of Linearizability

- S': order all ops by ts and attach responses
  - Write before reads, earlier read before later read, remaining ties broken arbitrarily
- Respect real-time order of non-overlapping ops
  - R ends before W begins → ts of R < ts of W because W increments ts → R occurs before W in S'
  - W ends before R begins → ts of W ≤ ts of R because one process relays W's ts to R (quorum intersection) → R occurs after W in S' (S' puts W before R)
  - $R_1$ ends before $R_2$ begins → ts of $R_1$ ≤ ts of $R_2$ for the same reason since $R_1$ calls update() just like W → $R_1$ occurs before $R_2$ in S' (S' puts earlier reads first)

# Linearizable SM Fault Tolerance

- Can we tolerate $f \geq n/2$ crashes?
- No, standard proof technique for disjoint quorums in asynchrony
  - Network partitioned

    [ Write X ]

    [ Read ]
  - Both ops finish eventually for fault tolerance but reader is unaware of writer due to async
- The proof/impossibility do not apply to sequential consistency

# Summary

- Atomic and sequential consistency are two most basic consistency models for shared memory systems
  - They are nice to have but expensive to achieve (atomic broadcast or ABD)

- Real-world processors opt for much weaker consistency models
  - Learn more in CS 598 Storage Systems