ESTR4316 Spring 2024

Consistency Models

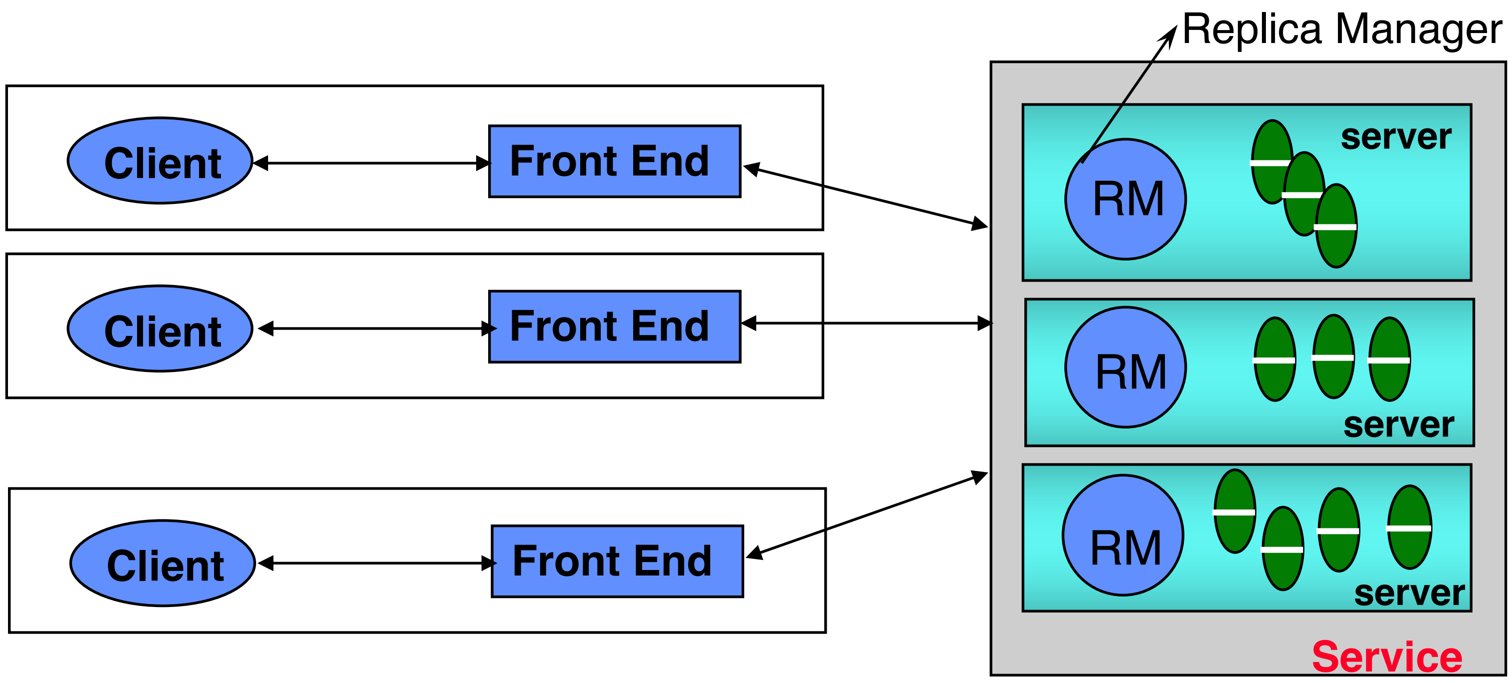
Prof. Wing C. Lau
Department of Information Engineering
wclau@ie.cuhk.edu.hk

Acknowledgements

- The slides used in this chapter are adapted from the following source(s):
  - Prof. Srini Seshan, CMU
  - Roy Campell, “Paxos and ZooKeeper,” Lecture notes of CS498 Cloud Computing, UIUC course, Spring 2014.
  - Shaz Qadeer, “Review: Linearizability”, Microsoft Research, 2011
  - Consistency Models: https://jepsen.io/consistency
  - Martin Kleppmann, Distributed Systems Lecture series, Cambridge University, 2021:
    - https://www.cl.cam.ac.uk/teaching/2122/ConcDisSys/materials.html
    - https://www.youtube.com/playlist?list=PLeKd45zvjcDFUEv_ohr_HdUFe97RItdiB
  - Steve Ko, University of Buffalo (SUNY Buffalo), CSE 486/586, Distributed Systems
  - Indranil Gupta, UIUC, Distributed Systems course

Consistency Models

Data Replication

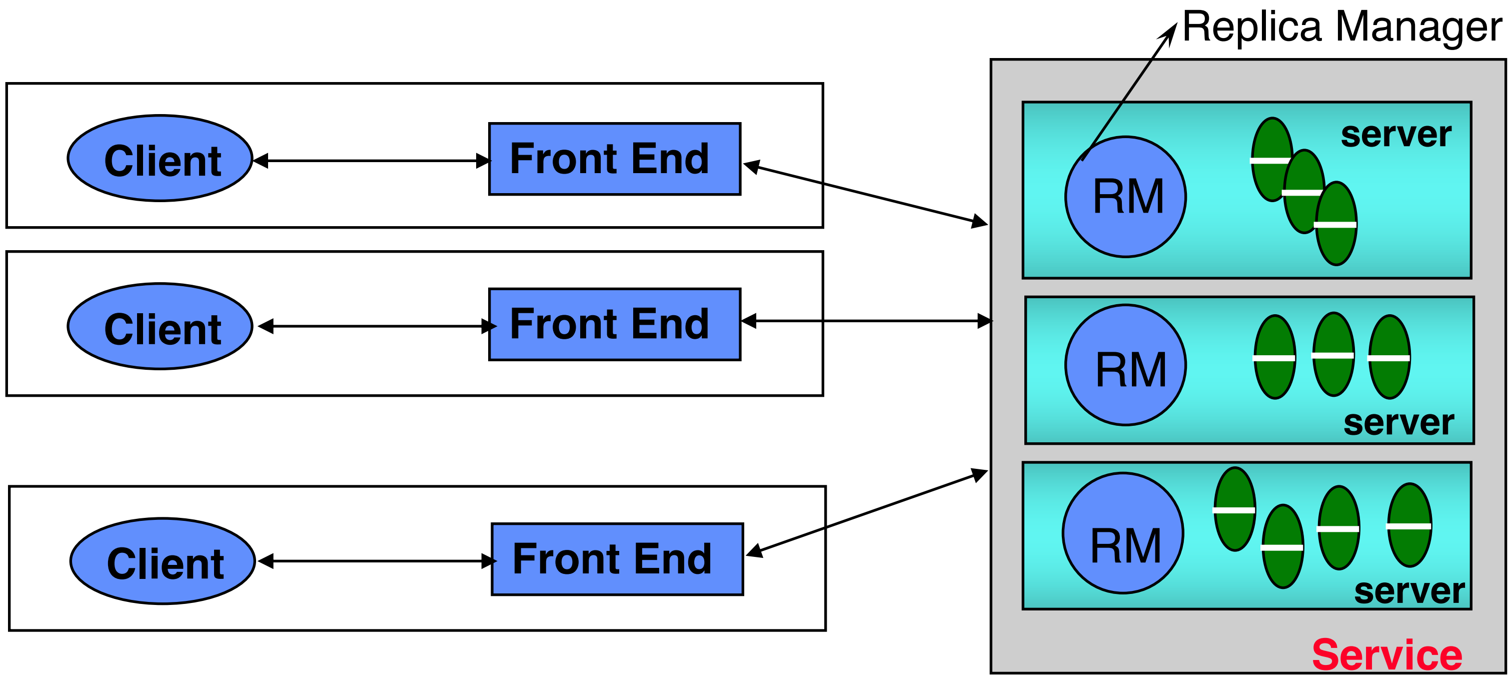


- Consider that this is a distributed storage system that serves read/write requests.
- Multiple copies of a same object stored at different servers

Why do we need Data Replication ?

- Why replicate?
- Increased availability of service. When servers fail or when the network is partitioned.
  - P: probability that one server fails= 1 – P= availability of service. e.g. P = 5% => service is available 95% of the time.
  - P^n: probability that n servers fail= 1 – P^n= availability of service. e.g. P = 5%, n = 3 => service available 99.875% of the time
- Fault tolerance
  - Under the fail-stop model, if up to f of f+1 servers crash, at least one is alive.
- Load balancing
  - One approach: Multiple server IPs can be assigned to the same name in DNS, which returns answers round-robin.

Consistency with Data Replicas



- Consider that this is a distributed storage system that serves read/write requests.
- Multiple copies of a same object stored at different servers
- Question: How to maintain consistency across different data replicas?

Consistency Models

- Consistency Model is a contract between processes and a data store
  - if processes follow certain rules, then store will work “correctly”
- Needed for understanding how concurrent reads and writes behave with respect to shared data
- Relevant for shared memory multiprocessors
  - cache coherence algorithms
- Shared databases, files
  - independent operations
    - » our main focus in the rest of the lecture
  - transactions

Taxonomy of Consistency Models

Data-centric Consistency Models

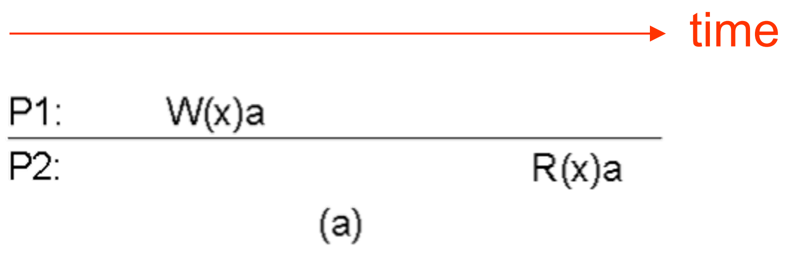
- Strict consistency
- Sequential consistency
- Linearizability
- Causal consistency
- FIFO consistency
- Weak consistency
- Release consistency
- Entry consistency

$\left.\right\}$ use explicit synchronization operations

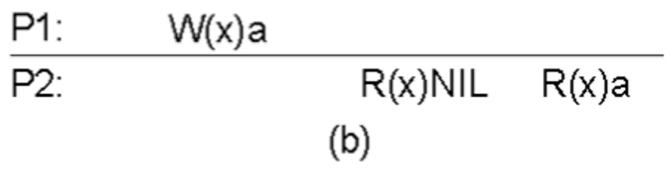
- Notation:
  - $W_i(x)a \rightarrow$ process i writes value a to location x
  - $R_i(x)a \rightarrow$ process i reads value a from location x

Strict Consistency

Any read on a data item x returns a value corresponding to the result of the *most recent* write on x. "All writes are instantaneously visible to all processes"



strictly consistent store



A store that is not strictly consistent.

Behavior of two processes, operating on the same data item.

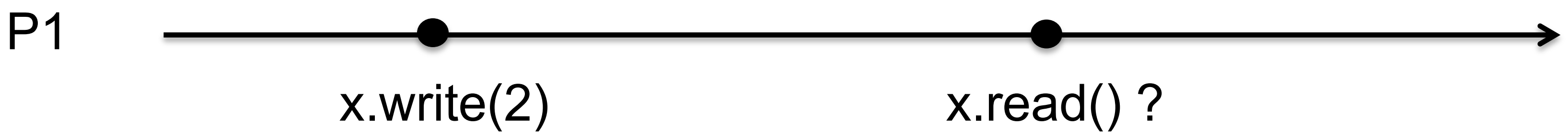
The problem with strict consistency is that it relies on *absolute global time* and is impossible to implement in a distributed system.

Outline

- We will look at different consistency guarantees (models).
- We'll start from the practically strongest guarantee, and gradually relax the guarantees.
  - Linearizability (or sometimes called strong consistency)
  - Sequential consistency
  - Causal consistency
  - FIFO consistency
  - Eventual consistency
- Different applications need different consistency guarantees.
- This is all about client-side perception.
  - When a read occurs, what do you return?
- First
  - Linearizability: we'll look at the concept first, then how to implement it later.

Our Expectation with Data

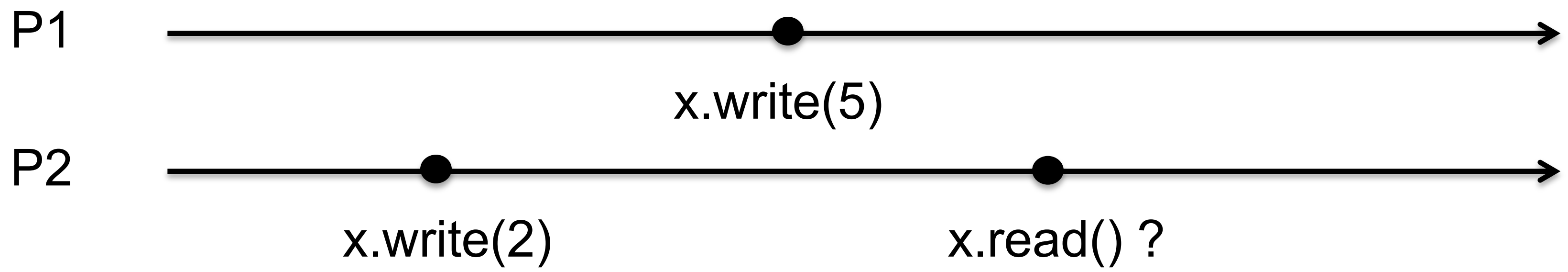
- Consider a single process using a filesystem
- What do you expect to read?



- Our expectation (as a user or a developer)
  - A read operation returns the most recent write.
  - This forms our basic expectation from any file or storage system.
- Linearizability meets this basic expectation.
  - But it extends the expectation to handle multiple processes…
  - …and multiple replicas.
  - The strongest consistency model

Expectation with Multiple Processes

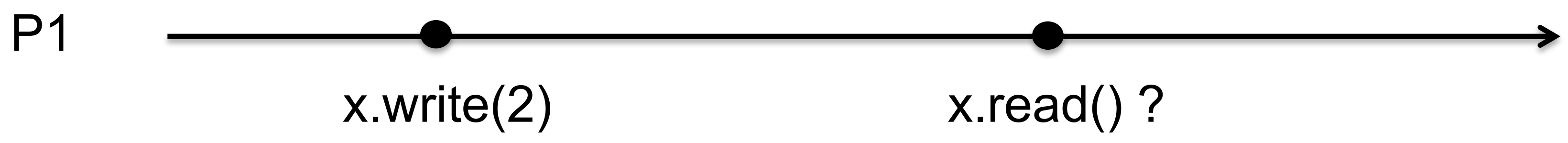
- What do you expect to read?
  - A single filesystem with multiple processes



- Our expectation (as a user or a developer)
- A read operation returns the most recent write, regardless of the clients.
- We expect that a read operation returns the most recent write according to the single actual-time order.
- In other words, read/write should behave as if there were a single (combined) client making all the requests.
- It’s easiest to understand and program for a developer if your storage appears to process one request at a time.

Expectation with Multiple Copies

- What do you expect to read?
  - A single process with multiple servers with copies



- Our expectation (as a user or a developer)
- A read operation returns the most recent write, regardless of how many copies there are.
- Read/write should behave as if there were a single copy.

Linearizability

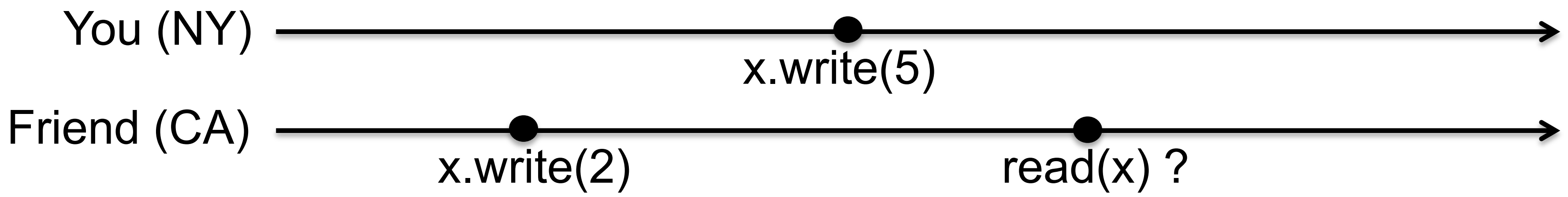
- Three aspects
  - A read operation returns the most recent write,
  - …regardless of the clients,
  - …according to the single actual-time ordering of requests.
- Or, put it differently, read/write should behave as if there were,
  - …a single client making all the (combined) requests in their original actual-time order (i.e., with a single stream of ops),
  - …over a single copy.
- You can say that your storage system guarantees linearizability when it provides single-client, single-copy semantics where a read returns the most recent write.
  - It should *appear* to all clients that there is *a single order (actual-time order) that your storage uses* to process all requests.

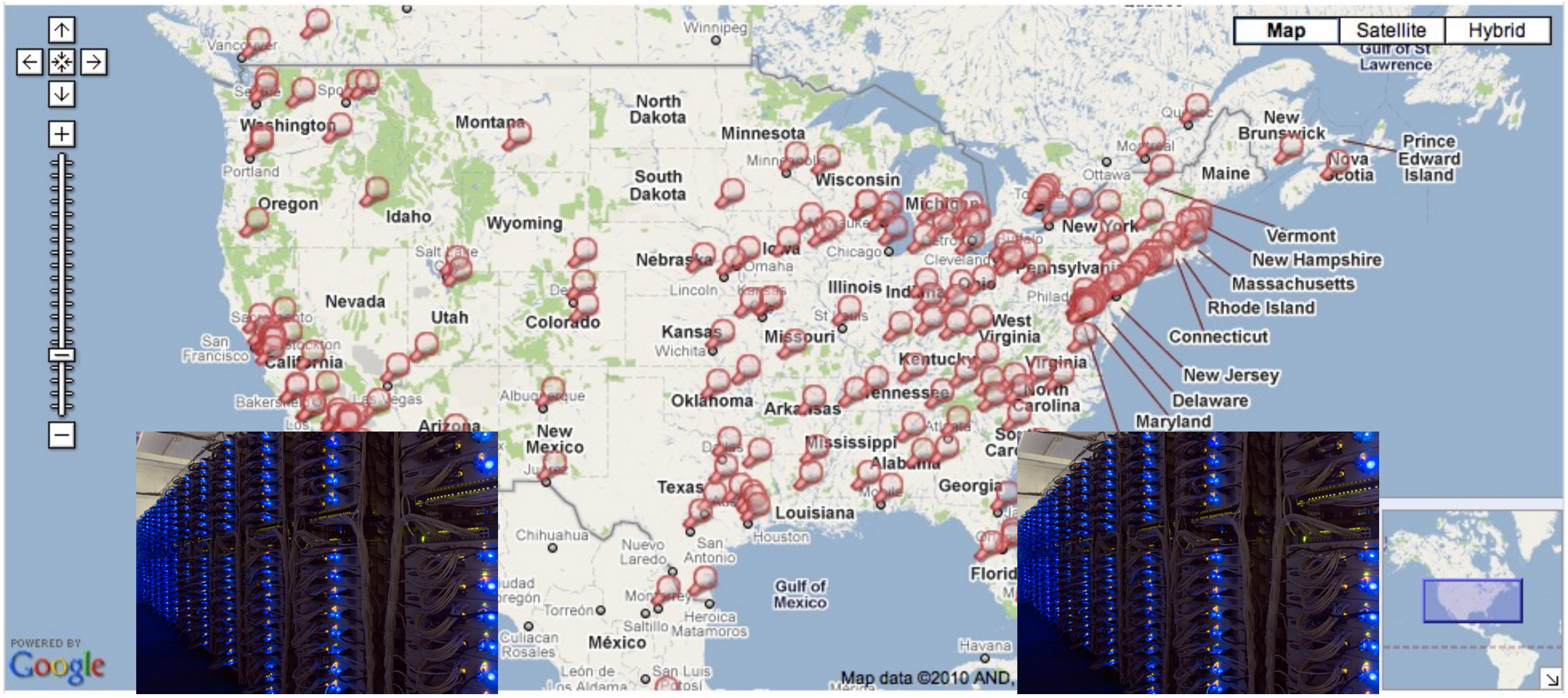
Linearizability Exercise

- Assume that the following happened with object x over a linearizable storage.
  - C1: x.write(A)
  - C2: x.write(B)
  - C3: x.read() → B, x.read() → A
  - C4: x.read() → B, x.read() → A
- What would be an actual-time ordering of the events?
  - One possibility: C2 (write B) -> C3 (read B) -> C4 (read B) -> C1 (write A) -> C3 (read A) -> C4 (read A)
- How about the following?
  - C1: x.write(A)
  - C2: x.write(B)
  - C3: x.read() → B, x.read() → A
  - C4: x.read() → A, x.read() → B

Linearizability Subtleties

- Notice any problem with the representation?

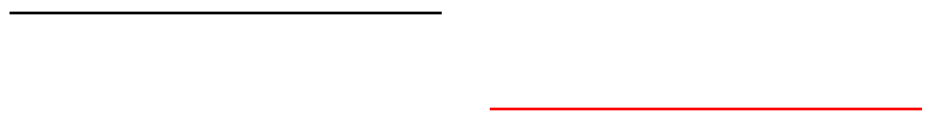




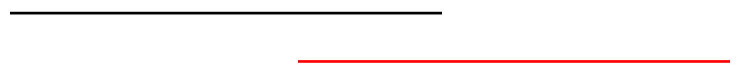
California North Carolina

Linearizability Subtleties

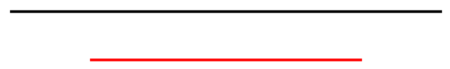
- A read/write operation is never a dot!
  - It takes time. Many things are involved, e.g., network, multiple disks, etc.
  - Read/write latency: the time measured right before the call and right after the call from the client making the call.
- Clear-cut (e.g., black---write & red---read)



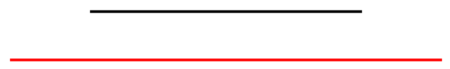
- Not-so-clear-cut (parallel)
  - Case 1:



  - Case 2:



  - Case 3:

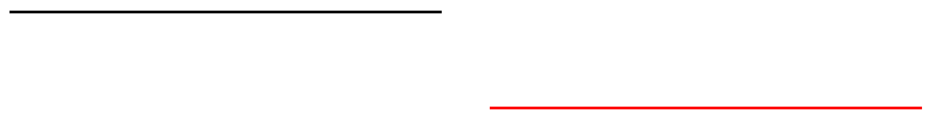


Linearizability Subtleties

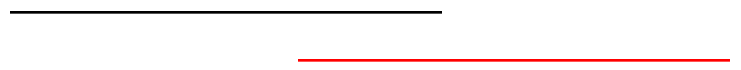
- With a single process and a single copy, can overlaps happen?
  - No, these are cases that do not arise with a single process and a single copy.
  - "Most recent write" becomes unclear when there are overlapping operations.
  - We don't necessarily have any "natural" expectation for this behavior.
- Thus, linearizability defines a reasonable thing:
  - As long as it appears to all clients that there is a single, interleaved ordering for all (overlapping and non-overlapping) operations that your implementation uses to process all requests, it's fine.
  - You can pick an ordering for processing, and there you need to show that you're returning the most recent write.

Linearizability Subtleties

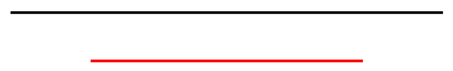
- Definite guarantee



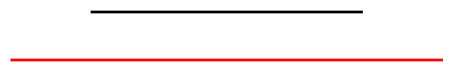
- Relaxed guarantee when overlap
- Case 1



- Case 2

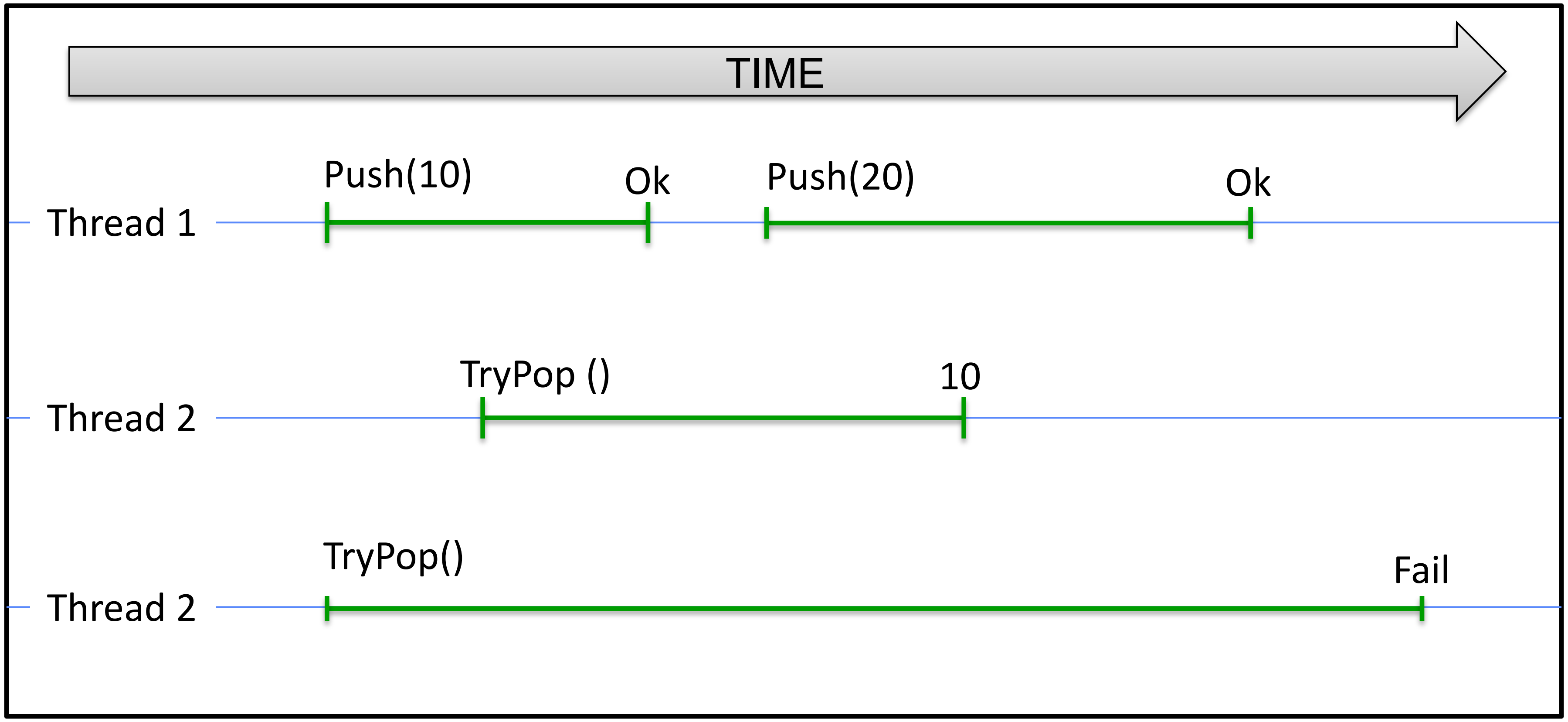


- Case 3



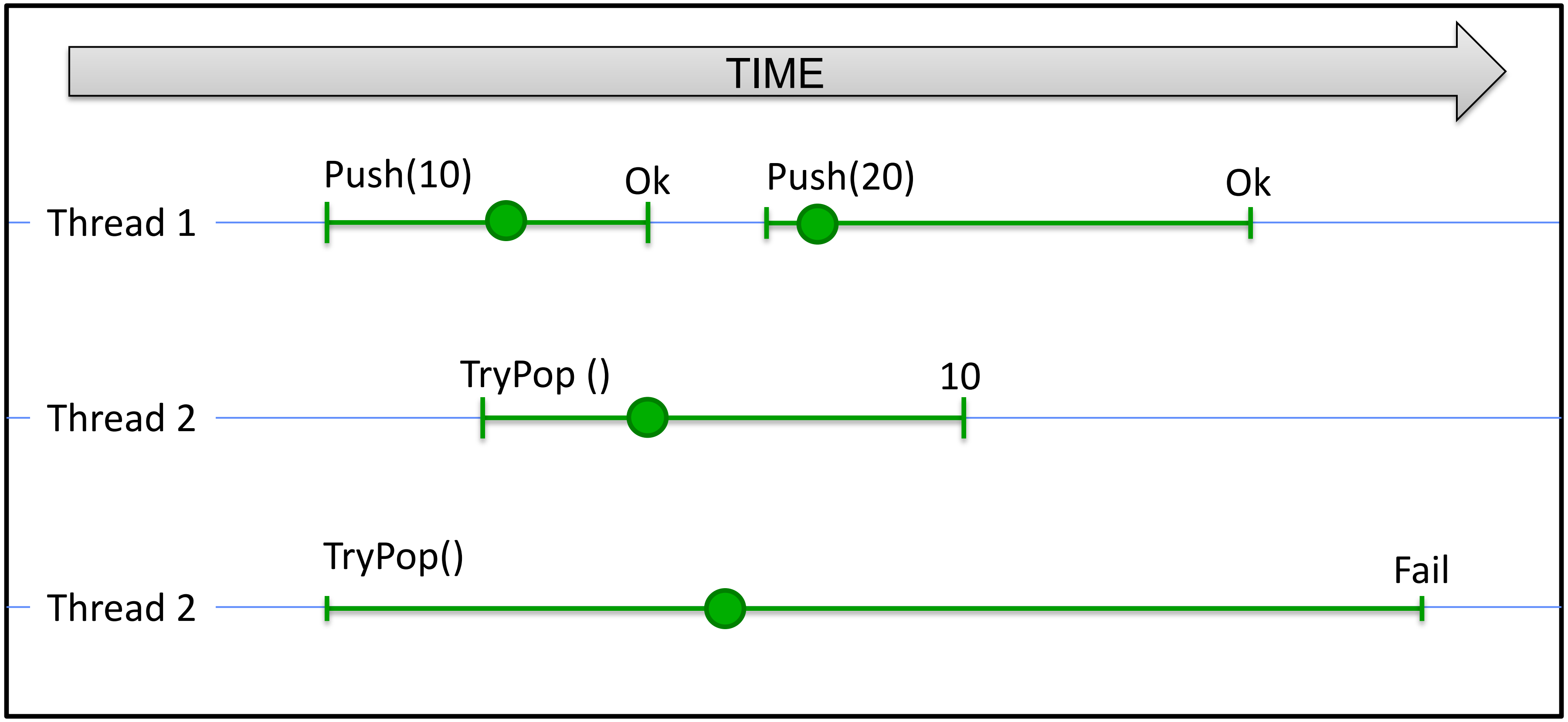
Example 1: Multi-thread access to the System Stack data-structure

- The following history is linearizable.



Example 1: Stack

- The following history is linearizable.

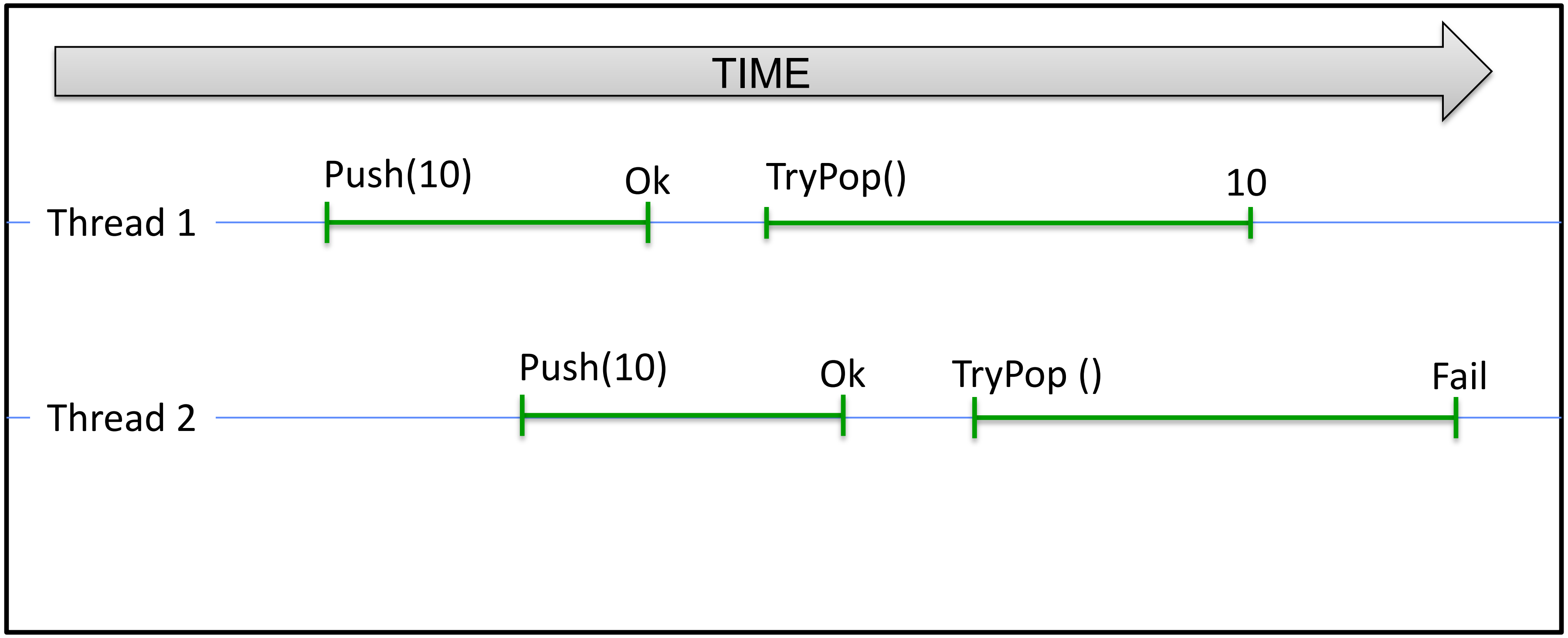


Definition of Linearizability

- Given some component C (say, a class or a data object)
- And some operations O1, O2, .. (say, methods)
- *An operation is linearizable if it always appears to take effect at a single instant of time (called the commit point) which happens sometime after the operation is called and before it returns.*
- Linearizable operations are sometimes called *atomic,* but that term is overused
  - (e.g. Not to confuse this "atomic" with the "All-or-nothing atomic" nature of multiple operations within a Database Transaction !!)

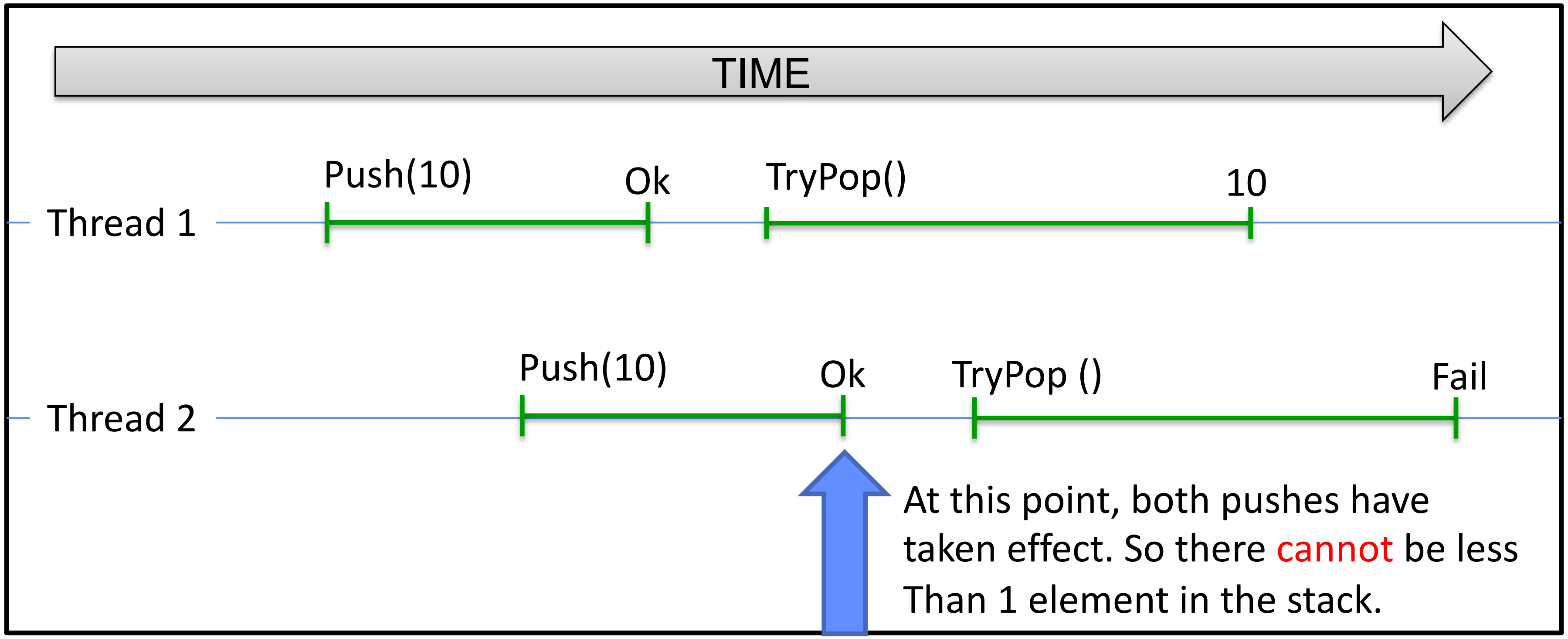
Example 2: Stack

- The following history is not linearizable.



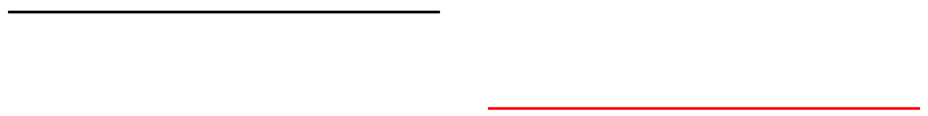
Example 2: Stack

- The following history is not linearizable.

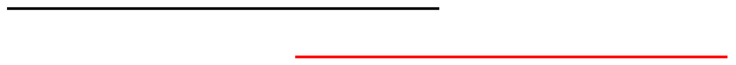

TIME
Push(10)
Ok
TryPop()
10
Thread 1
Push(10)
Ok
TryPop ()
Fail
Thread 2
At this point, both pushes have taken effect. So there cannot be less Than 1 element in the stack.

Linearizability Subtleties

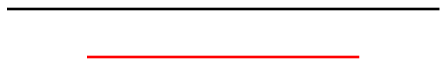
- Definite guarantee



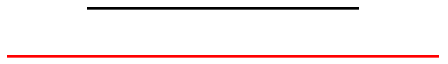
- Relaxed guarantee when overlap
- Case 1



- Case 2

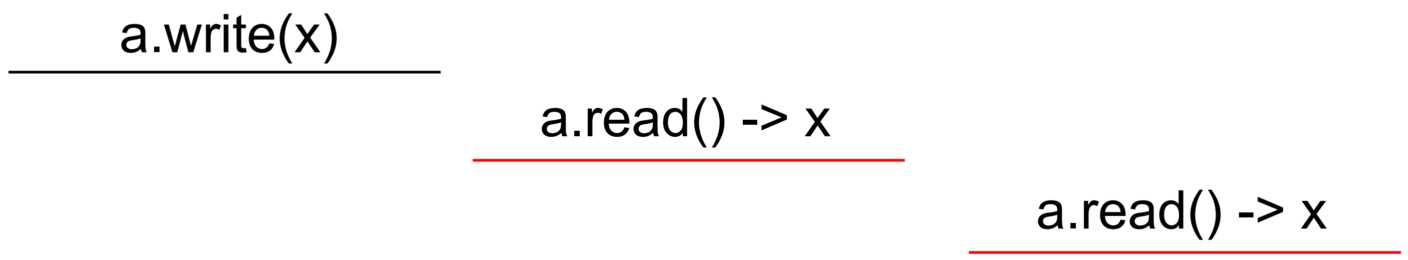


- Case 3

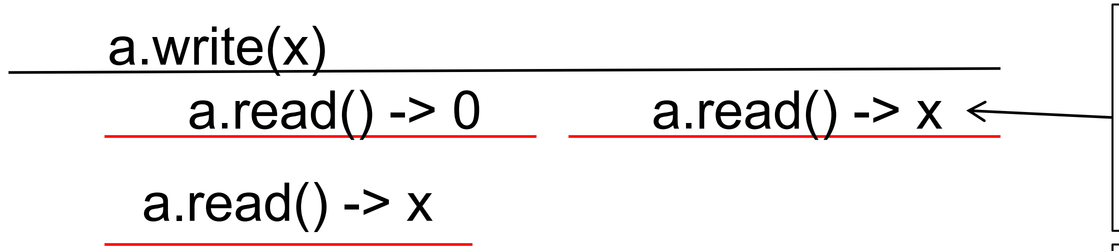


Linearizability Examples

- Example 1: if your system behaves this way…



- Example 2: if your system behaves this way…

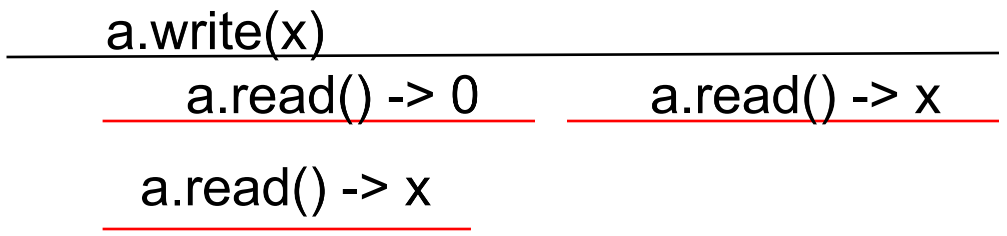


If this were a.read() -> 0, would it support linearizability?

No

Linearizability Examples

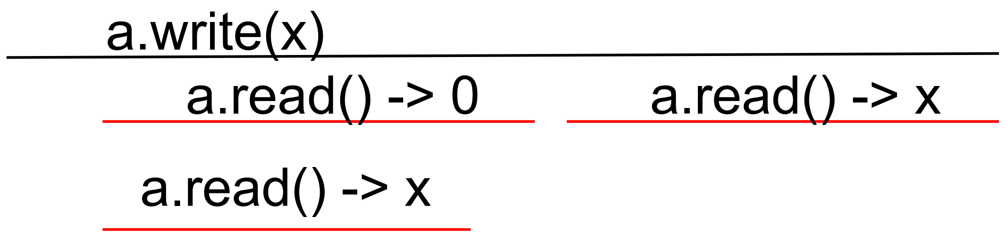
- In example 2, what are the constraints?



- Constraints
  - a.read() → 0 happens before a.read() →x (you need to be able to explain why that happens that way).
  - a.read() → x happens before a.read() →x (you need to be able to explain why that happens that way).
  - The rest are up for grabs.
- Scenario
  - a.write(x) gets propagated to (last client's) a.read() -> x first.
  - a.write(x) gets propagated to (the second process's) a.read() -> x, right after a.read() -> 0 is done.

Linearizability Examples

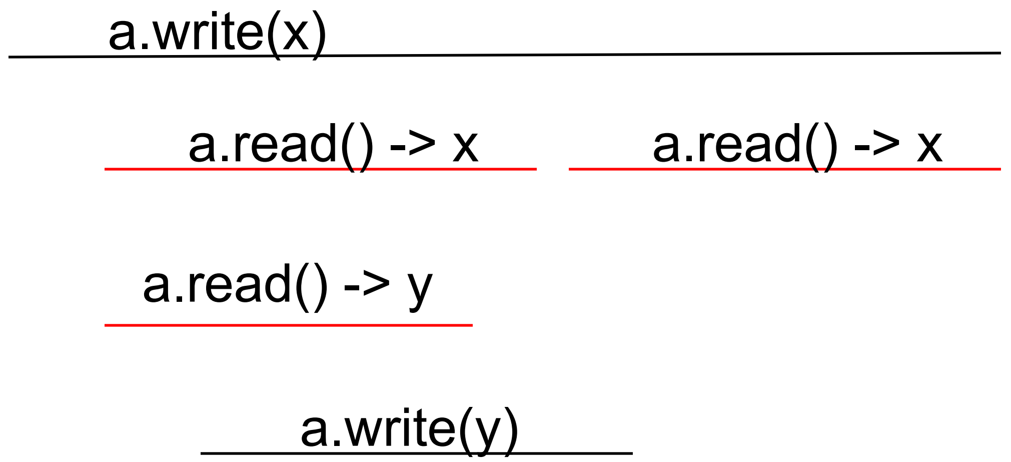
- In example 2, why would a.read() return 0 and x when they're overlapping?



- This assumes that there's a particular storage system that shows this behavior.
- At some point between a read/write request sent and returned, the result becomes visible.
  - E.g., you read a value from physical storage, *prepare it for return (e.g., putting it in a return packet, i.e., making it visible)*, and actually return it.
  - Or you *actually write a value to a physical disk, making it visible* (out of multiple disks, which might actually write at different points).

Linearizability Examples

- Example 3



- Constraints
  - a.read() → x and a.read() → x: we cannot change these.
  - a.read() → y and a.read() → x: we cannot change these.
  - The rest is up for grabs.

Linearizability (Textbook Definition)

- Let the sequence of read and update operations that client i performs in some execution be oi1, oi2,….
  - "Program order" for the client
- A replicated shared object service is linearizable if for any execution (real), there is some interleaving of operations (virtual) issued by all clients that:
  - meets the specification of a single correct copy of objects
  - is consistent with the actual times at which each operation occurred during the execution
- Main goal: any client will see (at any point of time) a copy of the object that is correct and consistent
- The strongest form of consistency

Summary

- Linearizability
  - Single-client, Single-copy semantics
- A read operation returns *the most recent* write, regardless of the clients, according to their actual-time ordering.

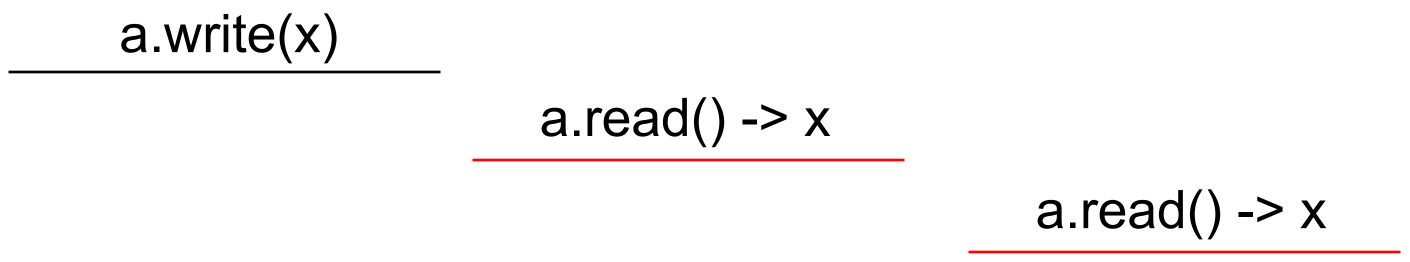
Consistency Models
Part II

Recap: Linearizability

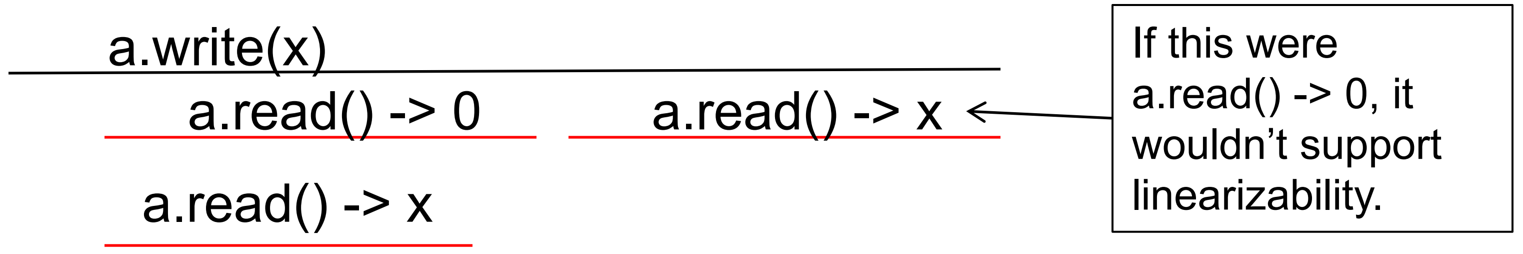
- Linearizability
  - Should provide the behavior of a single client and a single copy
  - A read operation returns the most recent write, regardless of the clients according to their original actual-time order.
- Complication
  - In the presence of concurrency, read/write operations overlap.
  - There, you should be able to show that you’re using some ordering of requests, where you return the most recent write (every time there’s a read).

Linearizability Examples

- Example 1



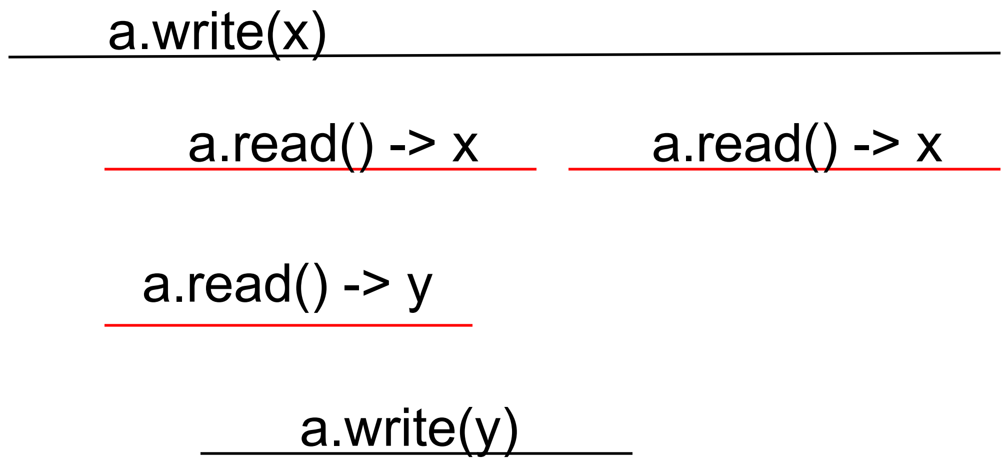
- Example 2



If this were a.read() -> 0, it wouldn't support linearizability.

Linearizability Examples

- Example 3

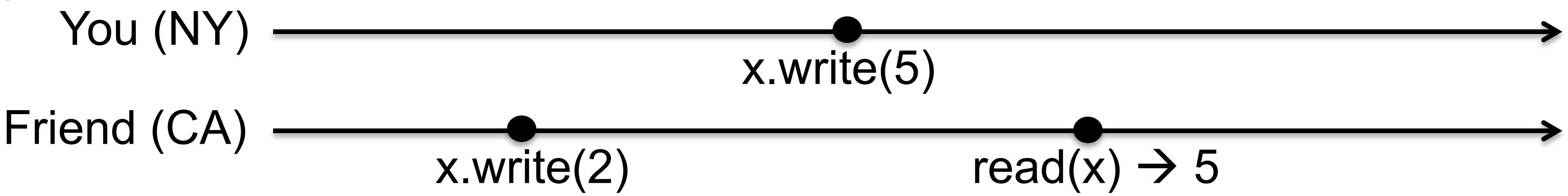


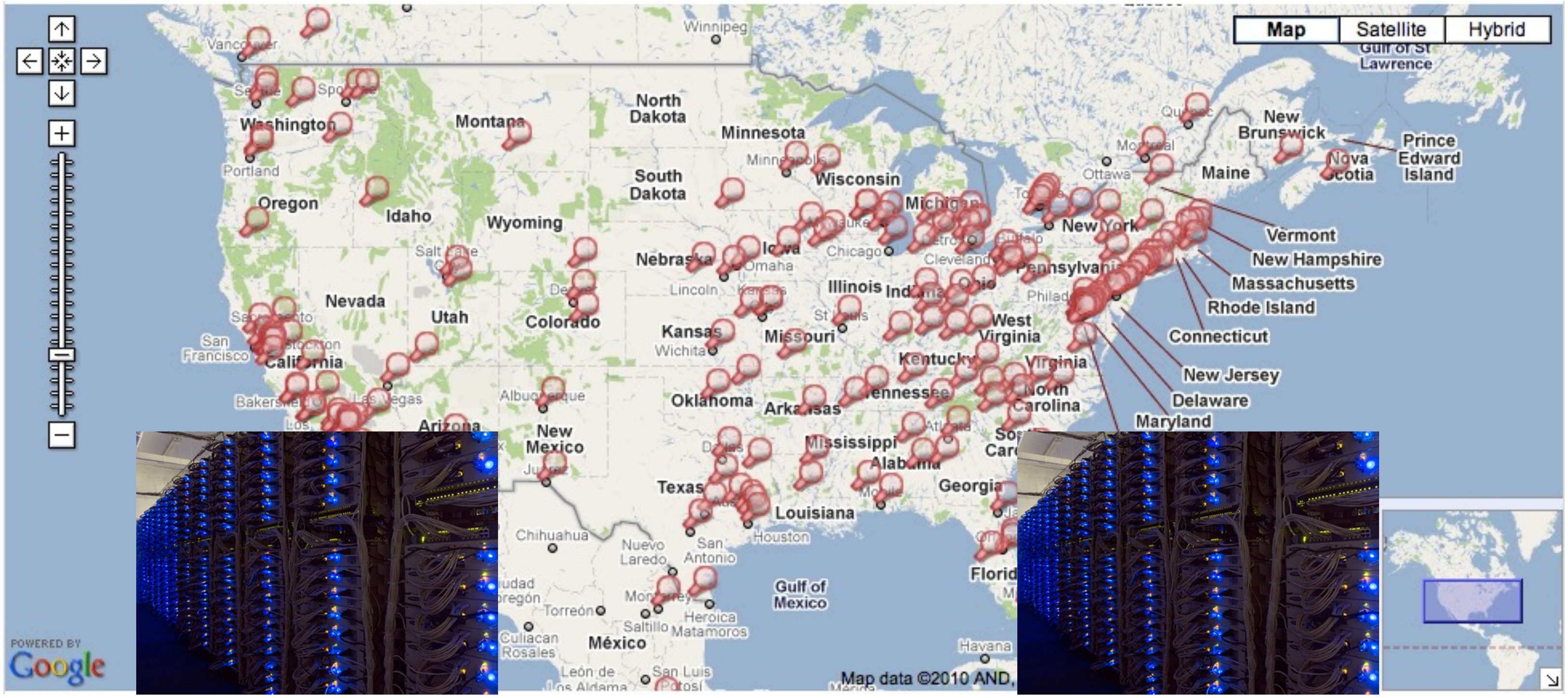
Linearizability

- Linearizability is all about client-side perception.
  - The same goes for all consistency models for that matter.
- If you write a program that works with a linearizable storage, *it works as you expect it to work*.
- There’s no surprise.

Implementing Linearizability

- Will this be difficult to implement? Any strategy?



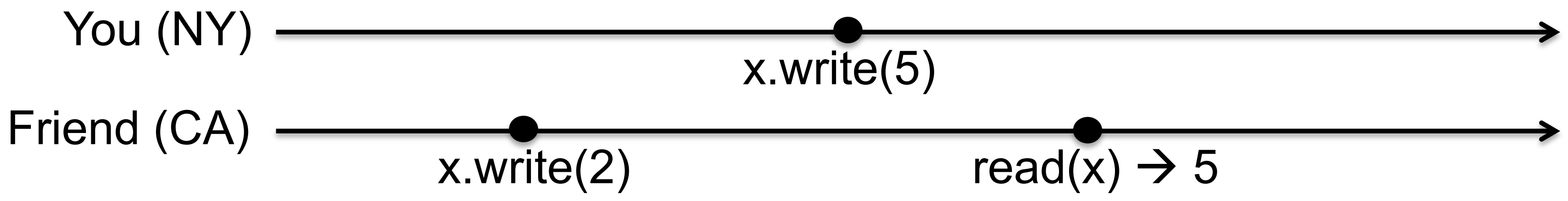


California

North Carolina

Implementing Linearizability

- Will this be difficult to implement?
  - It depends on what you want to provide.

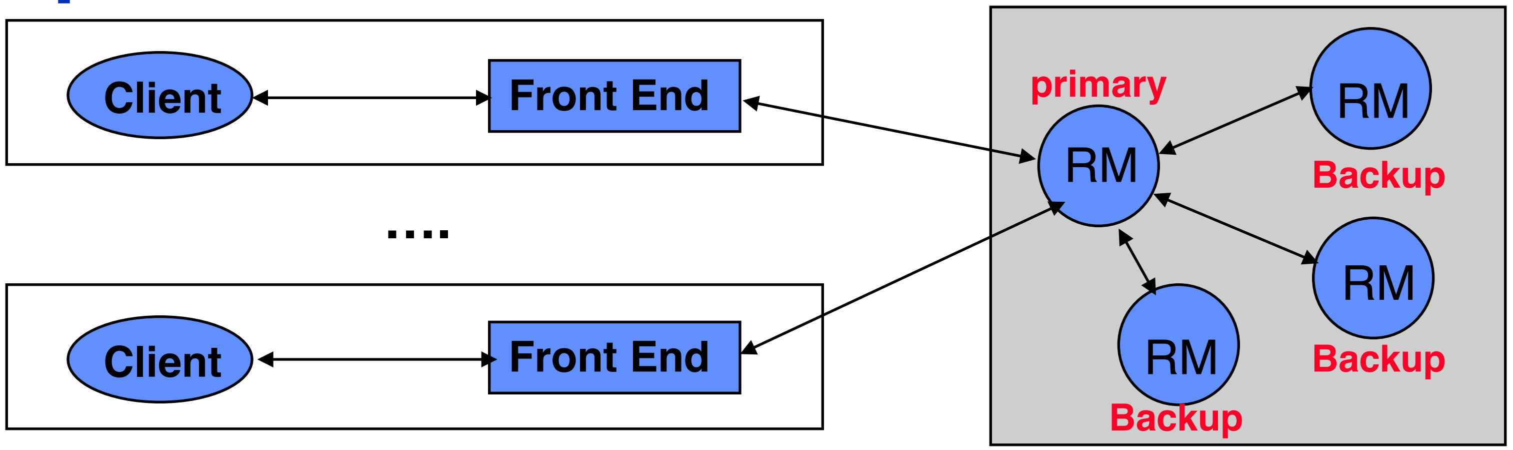


- How about:
  - All clients send all read/write to CA datacenter.
  - CA datacenter propagates to NC datacenter.
  - A request never returns until all propagation is done.
  - Correctness (linearizability)? yes
  - Performance? No

Implementing Linearizability

- Importance of latency
  - Amazon: every 100ms of latency costs them 1% in sales.
  - Google: an extra .5 seconds in search page generation time dropped traffic by 20%.
- Linearizability typically requires *complete* synchronization of multiple copies before a write operation returns.
  - So that any read over any copy can return the most recent write.
  - No room for asynchronous writes (i.e., a write operation returns before all updates are propagated.)
- It makes less sense in a global setting.
  - Inter-datecenter latency: ~10s ms to ~100s ms
- It might still makes sense in a local setting (e.g., within a single data center).

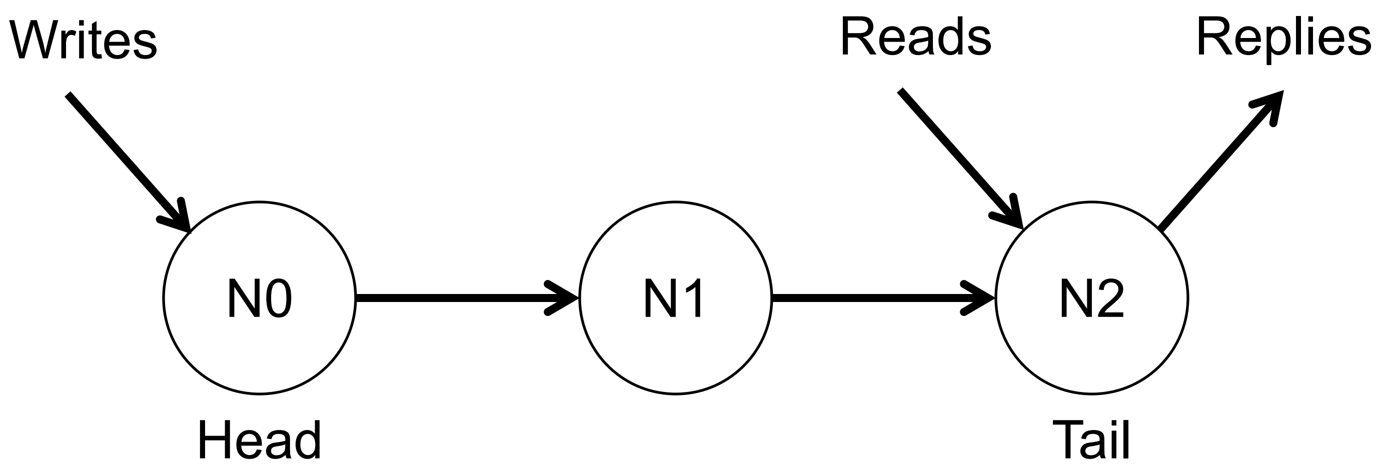
Passive (Primary-Backup) Replication



- Request Communication: the request is issued to the primary RM and carries a unique request id.
- Coordination: Primary takes requests atomically, in order, checks id (resends response if not new id.)
- Execution: Primary executes & stores the response
- Agreement: If update, primary sends updated state/result, req-id and response to all backup RMs (1-phase commit enough).
- Response: primary sends result to the front end

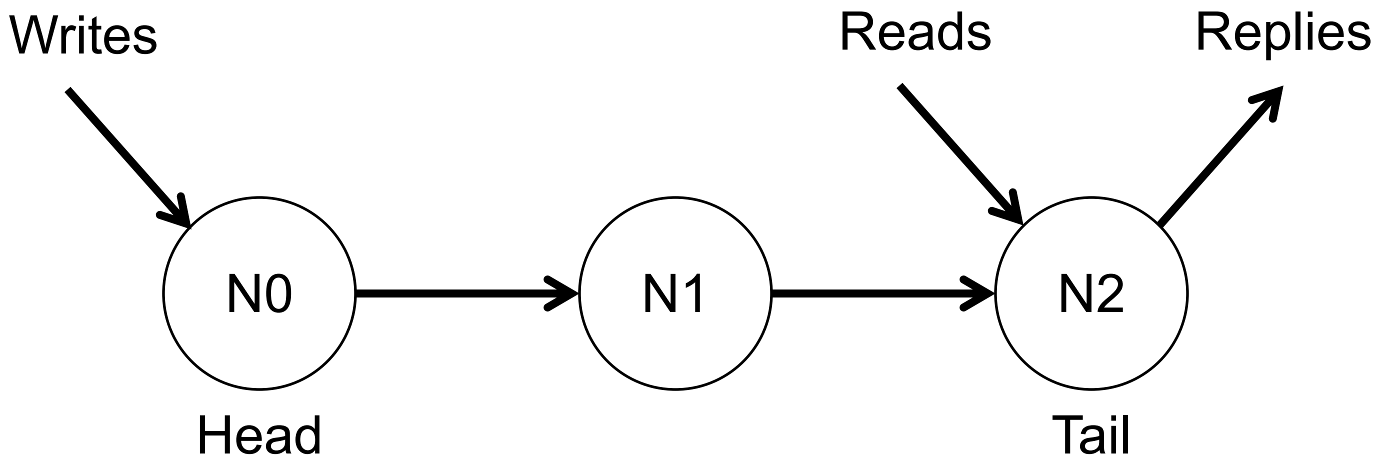
Chain Replication

- One technique to provide linearizability with better performance
  - All writes go to the head.
  - All reads go to the tail.



- Linearizability?
  - Clear-cut cases: straightforward
  - Overlapping ops?

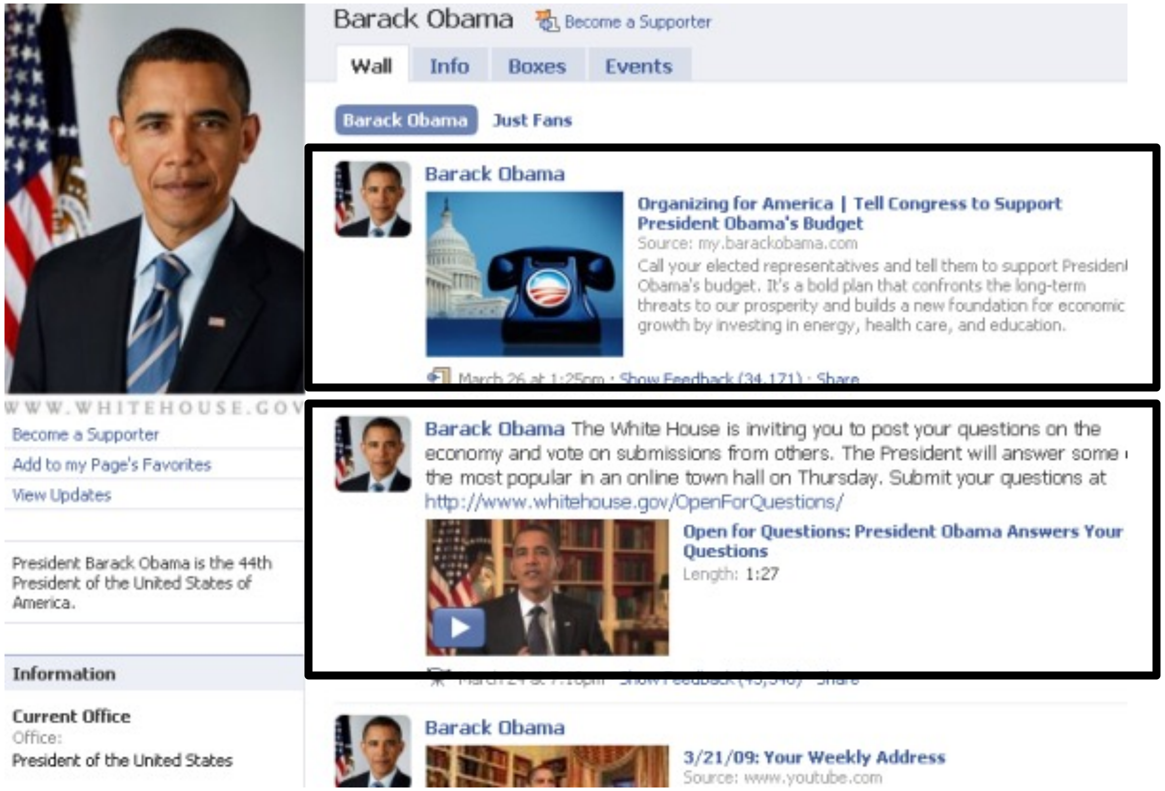
Chain Replication



- What ordering does this have for overlapping ops?
  - We have freedom to impose an order.
  - Case 1: A write is at either N0 or N1, and a read is at N2. The ordering we're imposing is read then write.
  - Case 2: A write is at N2 and a read is also at N2. The ordering we're imposing is write then read.
- Linearizability
  - Once a write becomes visible (at the tail), all following reads get the write result.

Relaxing the Guarantees

- Do we need linearizability?



Barack Obama Become a Supporter

Wall | Info | Boxes | Events

Barack Obama | Just Fans

Barack Obama

Organizing for America | Tell Congress to Support President Obama's Budget
Source: my.barackobama.com
Call your elected representatives and tell them to support President Obama's budget. It's a bold plan that confronts the long-term threats to our prosperity and builds a new foundation for economic growth by investing in energy, health care, and education.

March 26 at 1:25pm · Show Feedback (34,171) · Share

Barack Obama The White House is inviting you to post your questions on the economy and vote on submissions from others. The President will answer some of the most popular in an online town hall on Thursday. Submit your questions at http://www.whitehouse.gov/OpenForQuestions/

Open for Questions: President Obama Answers Your Questions
Length: 1:27

Barack Obama

3/21/09: Your Weekly Address
Source: www.youtube.com

WWW.WHITEHOUSE.GOV

Become a Supporter
Add to my Page's Favorites
View Updates

President Barack Obama is the 44th President of the United States of America.

Information

Current Office
Office:
President of the United States

- Does it matter if I see some posts some time later?
- Does everyone need to see these in this particular order?

Relaxing the Guarantees

- Linearizability advantages
  - It behaves as expected.
  - There’s really no surprise.
  - Application developers do not need any additional logic.
- Linearizability disadvantages
  - It’s difficult to provide high-performance (low latency).
  - It might be more than what is necessary.
- Relaxed consistency guarantees
  - Sequential consistency
  - Causal consistency
  - Eventual consistency
- It is still all about client-side perception.
  - When a read occurs, what do you return?

Sequential Consistency

Definition of Sequential consistency:

The result of any execution is the same as if the read and write operations by all processes were executed *in some (but the same) sequential order* and the operations of each individual process appear in this sequence in the order specified by its program [Lamport, 1979].

i.e. any valid interleaving for instructions from different processes is legal but all processes must see the same interleaving.

Examples of Sequential Consistency

Process P1	**Process P2**	**Process P3**
x = 1; print (y, z);	y = 1; print (x, z);	z = 1; print (x, y);

(a)

```
x = 1;
print (y, z);
y = 1;
print (x, z);
z = 1;
print (x, y);
```

Prints: 001011

(b)

```
x = 1;
y = 1;
print (x,z);
print(y, z);
z = 1;
print (x, y);
```

Prints: 101011

(c)

```
y = 1;
z = 1;
print (x, y);
print (x, z);
x = 1;
print (y, z);
```

Prints: 010111

(d)

```
y = 1;
x = 1;
z = 1;
print (x, z);
print (y, z);
print (x, y);
```

Prints: 111111

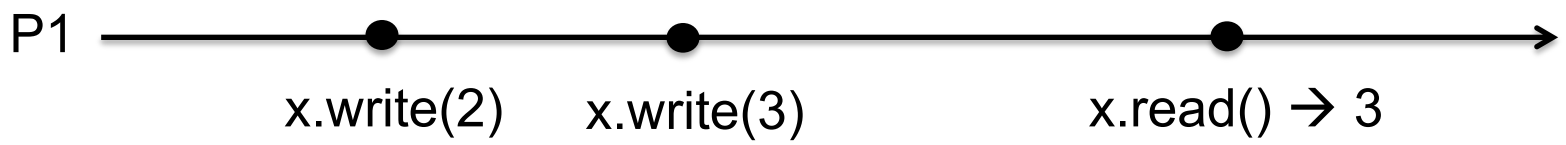
(a)-(d) are all legal interleavings.

Sequential Consistency

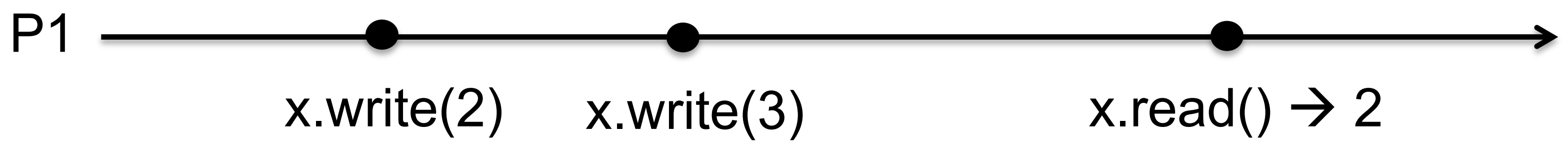
- A little weaker than linearizability, but still quite strong
  - Essentially linearizability, except that it allows writes from other processes to show up later.
- It still captures some reasonable expectation, but not the most natural one (which is captured by linearizability).
- For the remaining discussion,
  - Let's assume that there are multiple processes.
  - Let's also assume that each write has a unique value (just for the same of illustration).

Sequential Consistency

- Scenario 1: does this meet our natural expectation?



- Scenario 2: does this meet our natural expectation?

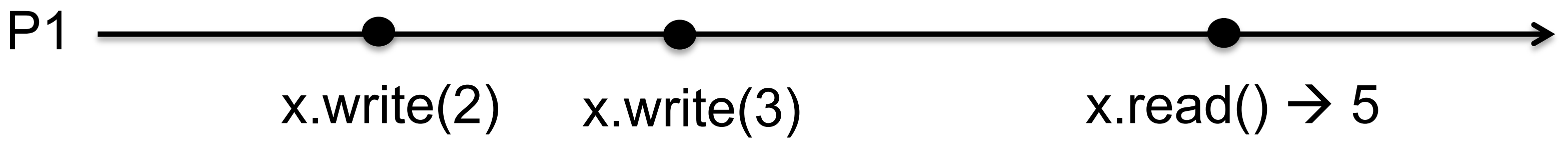


  – No. Why? Not the most recent write.
  – Another way to put it: we expect that a program order for a process is preserved.

- Sequential consistency at least preserves this expectation (each process's program order).

Sequential Consistency

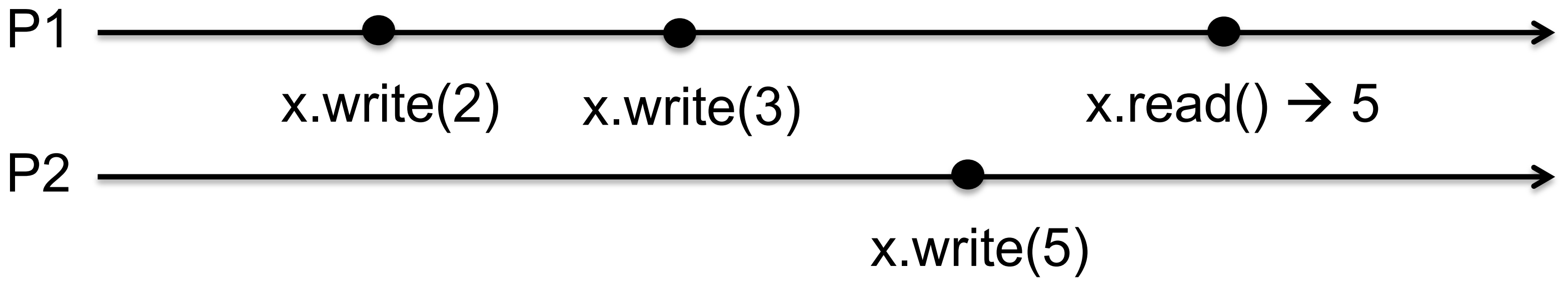
- Scenario 3: what if this happens (remember, there are multiple processes)?



  – We'll think that there must be a write after the last write.

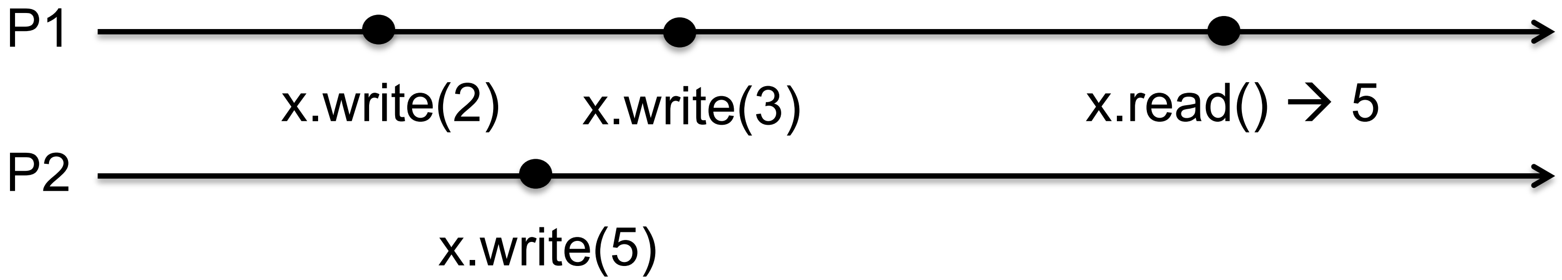
- Would we care which of these were true?

Case 1:



Or

Case 2:

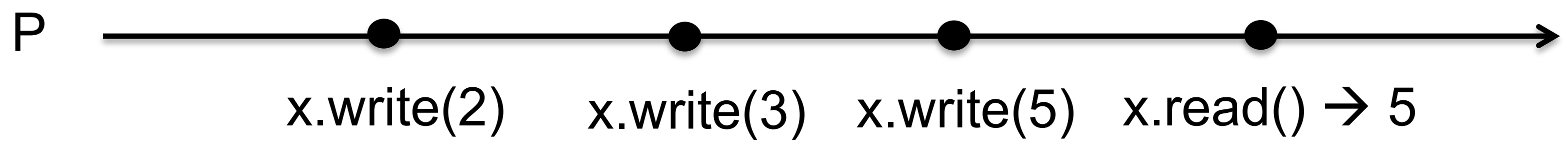


The gist of sequential consistency is that it allows Case 2. Linearizability doesn't.

The reason is that if P1 does not know anything about P2, then it is fine.
(e.g., P1 is a web browser and P2 is another web browser used by two different people),
P1 will just think that something must have happened between its write and read. Doesn't really matter when exactly that happened.

Sequential Consistency

- In both cases, the logical ordering is this:



- Sequential consistency: Your storage should *appear* to process all requests in a single interleaved ordering, where…
  - …each and every process's program order is preserved,
  - …and each process's program order is only *logically preserved w.r.t. other processes' program orders*, i.e., it doesn't need to preserve its physical-time ordering.
- It works as if all clients are reading out of a single copy.
  - This meets the expectation from a (isolated) client, working with a single copy.

N.B. Linearizabilty is based on physical time ordering. Sequential consistency is based on logical time ordering.
In both models, as long as we can come up with one ordering, where we show that we're definitely returning the most recent write, we're fine.
In linearizability, that's physical time ordering. In sequential consistency, that's logical ordering.

Sequential Consistency Examples

- Example 1: Can a sequentially consistent storage show this behavior? (I.e., can you come up with an interleaving that behaves like a single copy?)

```
– P1: a.write(A)
– P2:              a.write(B)
– P3:                           a.read()->B     a.read()->A
– P4:                                   a.read()->B     a.read()->A
```

- Example 2

```
– P1: a.write(A)
– P2:              a.write(B)
– P3:                           a.read()->B     a.read()->A
– P4:                                   a.read()->A     a.read()->B
```

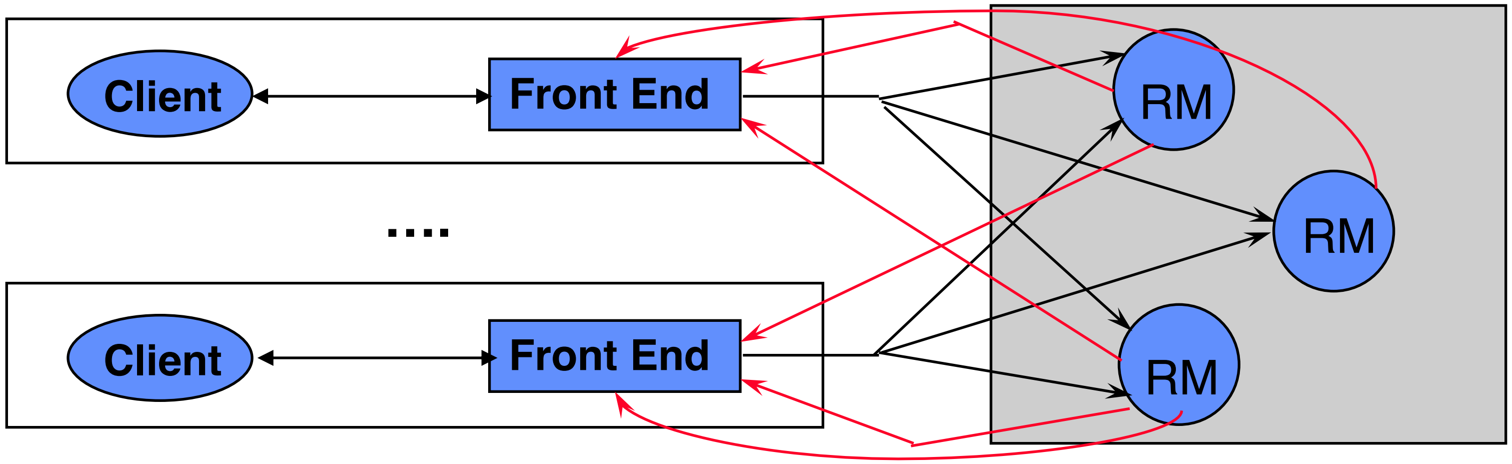
Implementing Sequential Consistency

- In what implementation would the following happen?
  - P1: a.write(A)
  - P2:   a.write(B)
  - P3:   a.read()->B   a.read()->A
  - P4:   a.read()->A   a.read()->B
- Possibility
  - P3 and P4 use different copies.
  - In P3’s copy, P2’s write arrives first and gets applied.
  - In P4’s copy, P1’s write arrives first and gets applied.
  - Writes are applied in different orders across copies.
  - This doesn’t provide sequential consistency.

Implementing Sequential Consistency

- Typical implementation
  - You’re not obligated to make the most recent write (according to actual time) visible (i.e., applied to all copies) right away.
  - But you are obligated to apply all writes in the same order for all copies. This order should be FIFO-total.

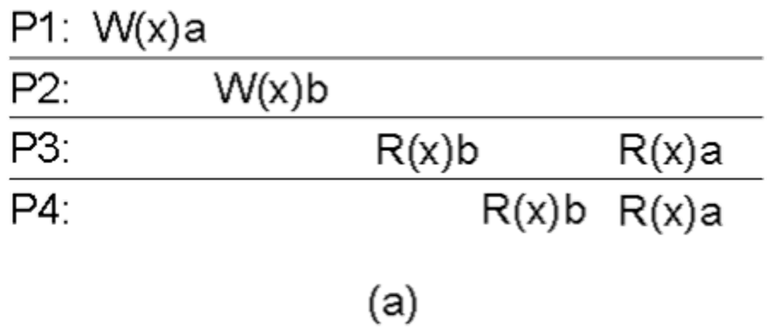
Active Replication

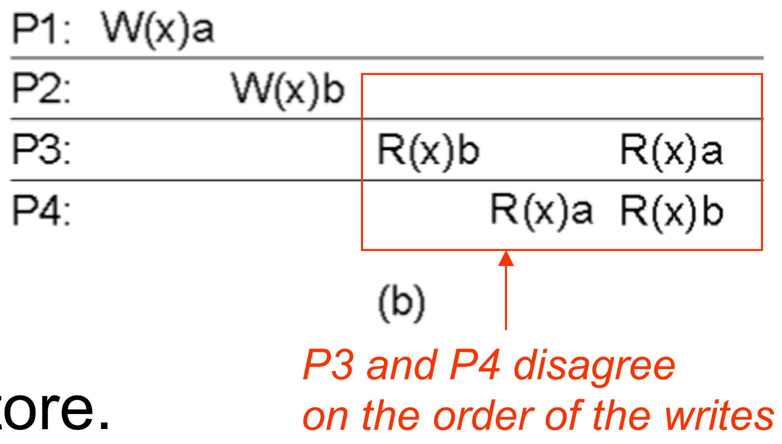


- A front end FIFO-orders all reads and writes.
- A read can be done completely with any single replica.
- Writes are totally-ordered and asynchronous (after at least one write completes, it returns).
  - Total ordering doesn't guarantee when to deliver events, i.e., writes can happen at different times at different replicas.
- Sequential consistency, not linearizability
  - Read/write ops from the same client will be ordered at the front end (program order preservation).
  - Writes are applied in the same order by total ordering (single copy).
  - No guarantee that a read will read the most recent write based on actual time.

Summary: Sequential Consistency

Sequential consistency: the result of any execution is the same as if the read and write operations by all processes were executed *in some (but the same) sequential order* and the operations of each individual process appear in this sequence in the order specified by its program [Lamport, 1979].
Note: Any valid interleaving is legal but all processes must see the same interleaving.





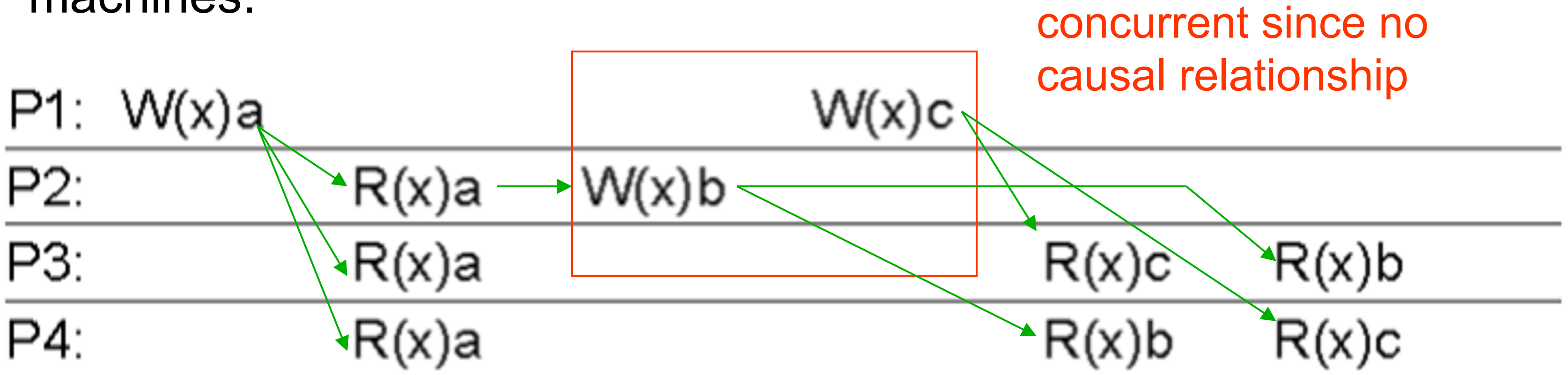
a) A sequentially consistent data store.
b) A data store that is not sequentially consistent.

Two More Consistency Models

- Even more relaxed
  - We don't even care about providing an illusion of a single copy.
- Causal consistency
  - We care about ordering causally related write operations correctly.
- FIFO consistency
  - Writes done by a single process are seen by all other processes in the order in which they were issued ;
  - BUT Writes from different processes may be seen in a different order by different processes.
- Eventual consistency
  - As long as we can say all replicas converge to the same copy eventually, we're fine.

Causal Consistency

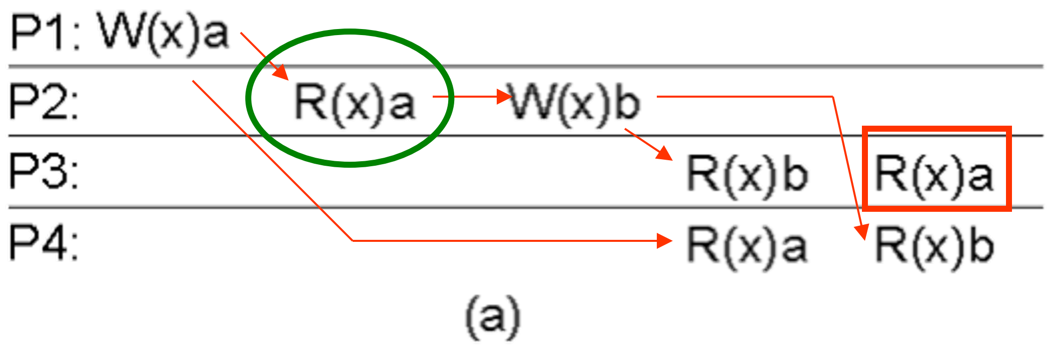
- Necessary condition: For “Writes” that are potentially causally related, they (and thus their results) must be seen by all processes in the same order.
- Concurrent writes may be seen in a different order on different machines.

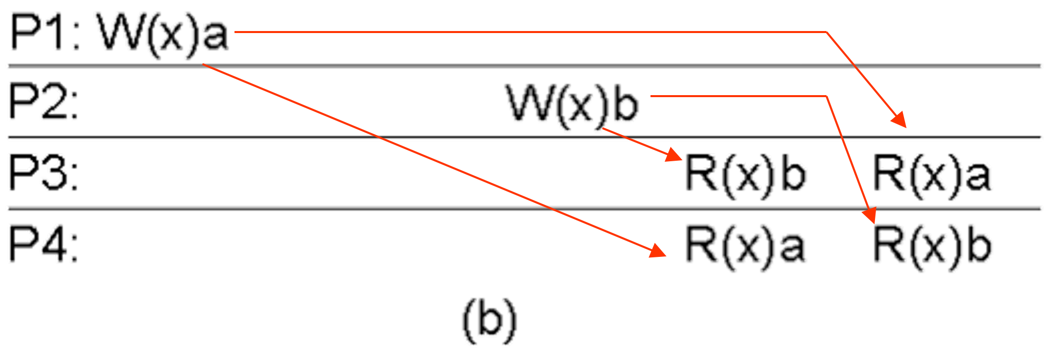


This sequence is allowed with a causally-consistent store, but not with sequentially or strictly consistent store.

Can be implemented with vector clocks.

Causal Consistency (cont’d)



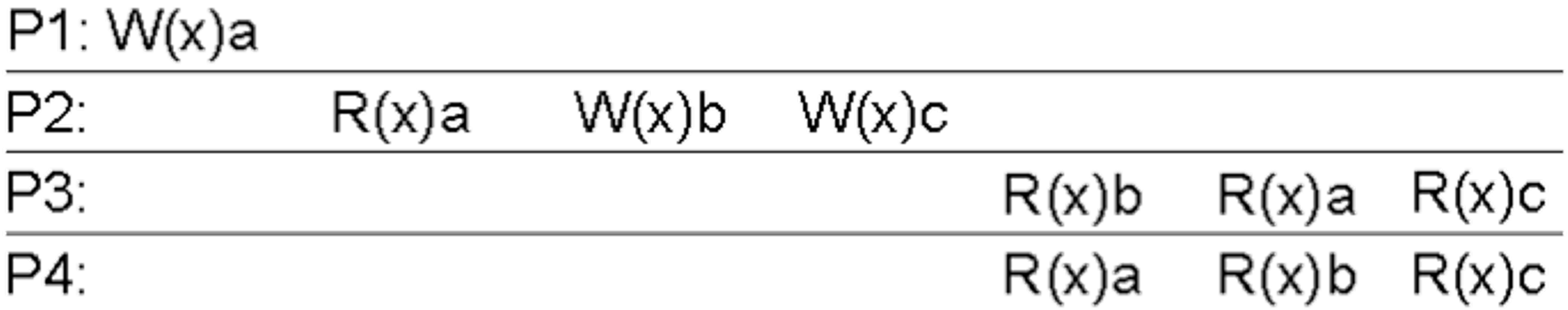


a) A violation of a causally-consistent store. The two writes are NOT concurrent because of the $R_2(x)a$.

b) A correct sequence of events in a causally-consistent store ($W_1(x)a$ and $W_2(x)b$ are concurrent).

FIFO Consistency

Necessary Condition: Writes done by a single process are seen by all other processes in the order in which they were issued, but Writes from different processes may be seen in a different order by different processes.



A valid sequence of events of FIFO consistency. Only requirement in this example is that P2's writes are seen in the correct order. FIFO consistency is easy to implement.

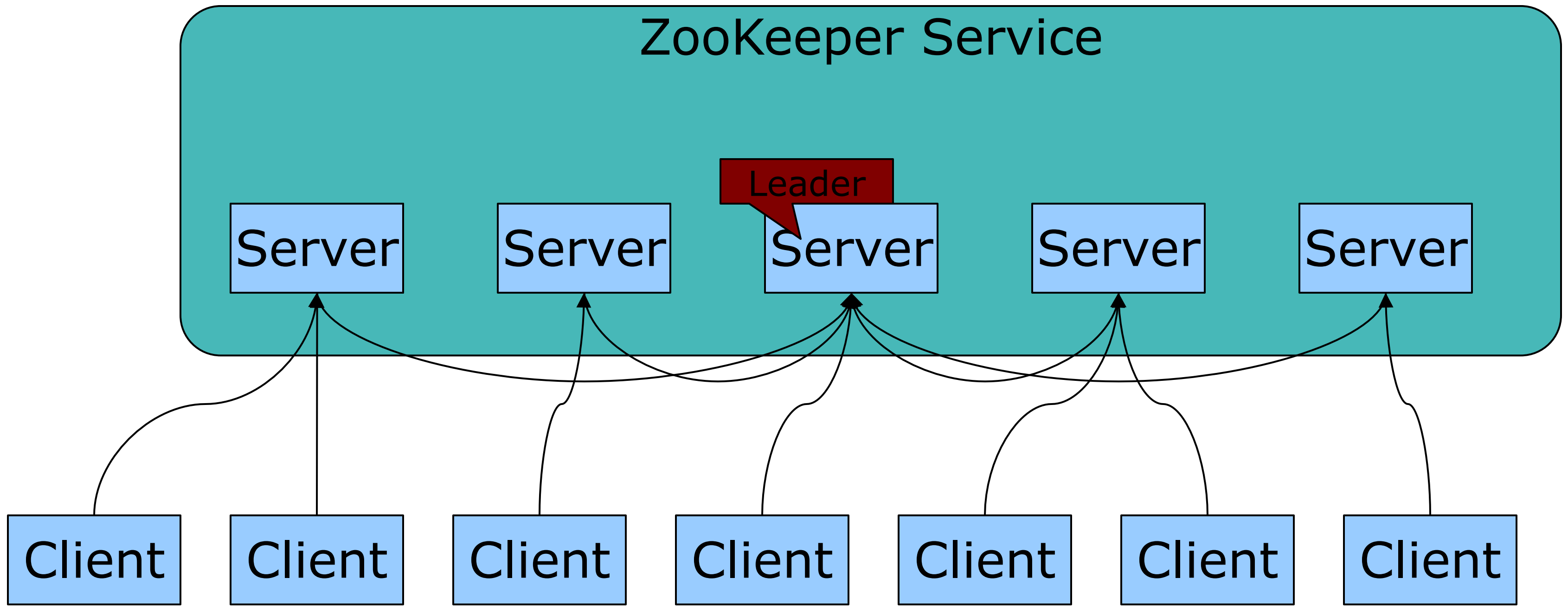
Summary

- Linearizability
  - The ordering of operations is determined by time.
  - Primary-backup can provide linearizability.
  - Chain replication can also provide linearizability.
- Sequential consistency
  - The ordering of operations preserves the program order of each client.
  - Active replication can provide sequential consistency.

Summary of Some Common Consistency Models

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (loosely synchronized, non-unique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order

Recap: ZooKeeper Service



- All servers store a copy of the data, logs, snapshots on disk and use an in memory database
- A leader is elected at startup
- Followers service clients
- Update responses are sent when a majority of servers have persisted the change

Properties Guaranteed by ZooKeeper

- FIFO per Client Order
  - All requests of the same client will be applied/ processed in the order they were sent.
  - Ordering of notifications and state changes to a Client are also guaranteed
- Linearizability for all requests that change Zookeeper states
  - i.e. Linearizability for Writes (only): All Clients will observe “parallel” writes issued by different Clients in the same order ; the exact order is determined by time-stamps of a global, loosely synchronized clock.
  - Linearized writes is realized using the Zookeeper Atomic Broadcast (ZAB) protocol/ algorithm (http://research.yahoo.com/files/ladis08.pdf)
  - ZAB is inspired by, but different from, the **Paxos** algorithm
  - **Reads issued by different clients are not linearized !**
- Atomicity - updates either Succeed or Fail, no partial results
  - File API without partial reads/writes
  - Simple **wait free** data objects organized hierarchically as in filesystem.
  - “Multi-hop” construct to support atomic (i.e. all or nothing) execution of a block of multiple requests

Properties Guaranteed by ZooKeeper

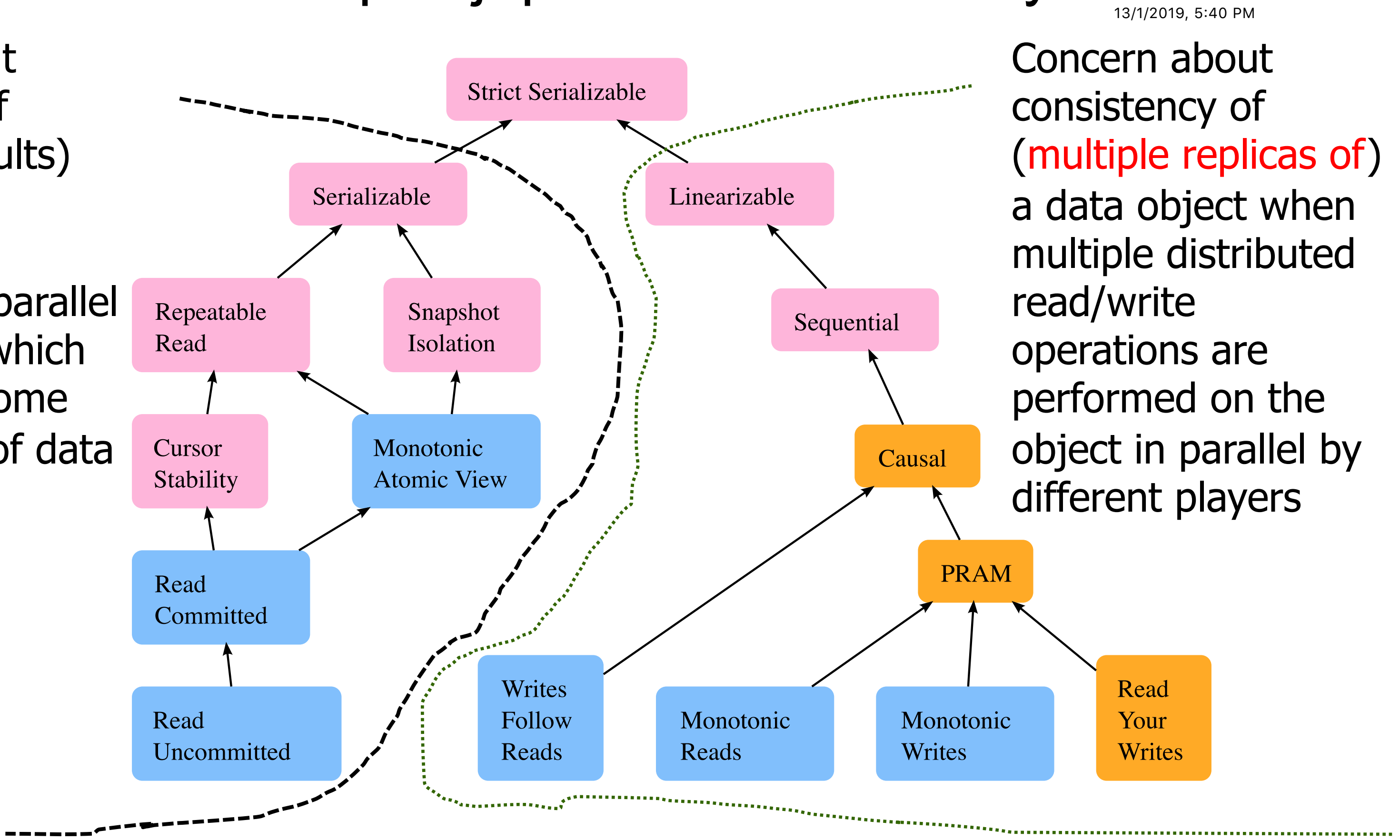
- Single System Image
  - The SAME client will see the same view of the service no matter which server it connects to ; (Different Clients may see a different (delayed) version of the view though ;
- Durability - once an update has been applied, it will persist from that time forward until a client overwrites the update.
- High Availability - 2F+1 servers can tolerate F crash failures
- Timeliness – The client's view of the system is guaranteed to be up-to-date within a certain bound delay.
  - By setting the "watch" flag in a Read request, a client will get notified of a change to data it is watching within a bounded period of time.
  - Either system changes will be seen by a client within this bound or the client will detect a service outage
  - There are corner cases that intermediate state-changes can be missed by a particular client due to the "one-time-trigger" notion of watch (see p.g. 70 of the ZK book)

Recap: Taxonomy of Consistency Models

https://jepsen.io/consistency

Concern about consistency of outcome (results) produced by multiple interleaving/ parallel transactions which may access some common set of data objects

Concern about consistency of (multiple replicas of) a data object when multiple distributed read/write operations are performed on the object in parallel by different players



Legend

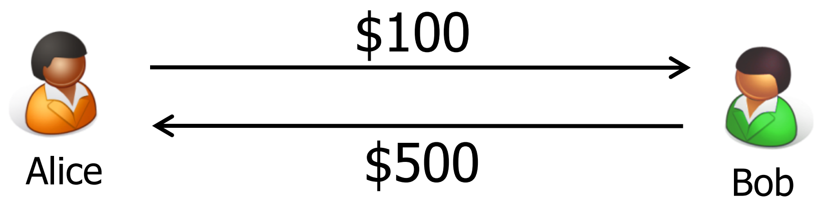
Unavailable	Not available during some types of network failures. Some or all nodes must pause operations in order to ensure safety.
Sticky Available	Available on every non-faulty node, so long as clients only talk to the same servers, instead of switching to new ones.
Total Available	Available on every non-faulty node, even when the network is completely down.

Why do we need Serializability ?

```
void transferMoney(customer A, customer B, int amount)
{
  showMessage("Transferring "+amount+" to "+B);
  int balanceA = getBalance(A);
  int balanceB = getBalance(B);
  setBalance(B, balanceB + amount);
  setBalance(A, balanceA - amount);
  showMessage("Your new balance: "+(balanceA-amount));
}
```

- Simple example: Accounting system in a bank
  - Maintains the current balance of each customer's account
  - Customers can transfer money to other customers

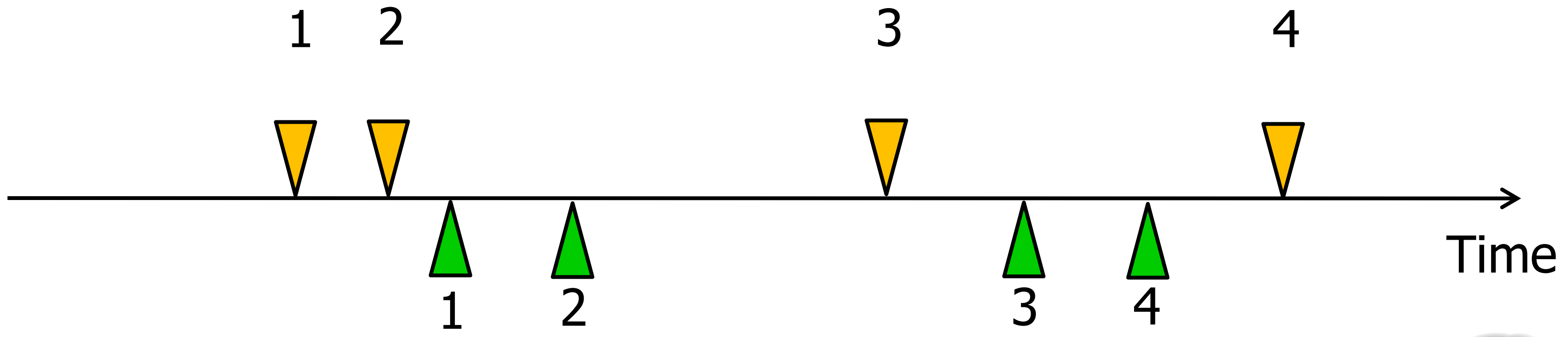
Why do we need Serializability ?



```
1) B=Balance(Bob)
2) A=Balance(Alice)
3) SetBalance(Bob,B+100)
4) SetBalance(Alice,A-100)
```

```
1) A=Balance(Alice)
2) B=Balance(Bob)
3) SetBalance(Alice,A+500)
4) SetBalance(Bob,B-500)
```

- What can happen if these 2 programs run concurrently according to the following interlaced pattern (which is a valid outcome under sequential consistency) ?



Alice's balance:	$200	$200	$700	$700	$100
Bob's balance:	$800	$900	$900	$300	$300

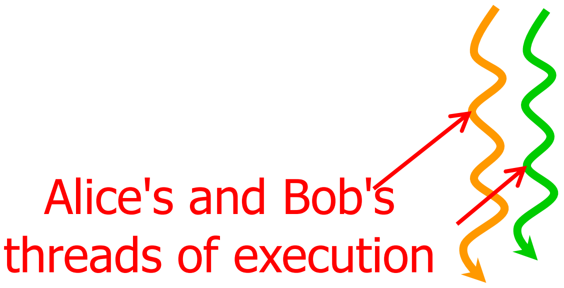
Recall: Sequential Consistency example

Process P1	Process P2	Process P3
x = 1;	y = 1;	z = 1;
print (y, z);	print (x, z);	print (x, y);

`x = 1;`	`x = 1;`	`y = 1;`	`y = 1;`
`print (y, z);`	`y = 1;`	`z = 1;`	`x = 1;`
`y = 1;`	`print (x,z);`	`print (x, y);`	`z = 1;`
`print (x, z);`	`print(y, z);`	`print (x, z);`	`print (x, z);`
`z = 1;`	`z = 1;`	`x = 1;`	`print (y, z);`
`print (x, y);`	`print (x, y);`	`print (y, z);`	`print (x, y);`
Prints: 001011	Prints: 101011	Prints: 010111	Prints: 111111
(a)	(b)	(c)	(d)

(a)-(d) are all legal interleavings.

Problem: Race Condition



```
void transferMoney(customer A, customer B, int amount)
{
  showMessage("Transferring "+amount+" to "+B);
  int balanceA = getBalance(A);
  int balanceB = getBalance(B);
  setBalance(B, balanceB + amount);
  setBalance(A, balanceA - amount);
  showMessage("Your new balance: "+(balanceA-amount));
}
```

- What happened?
  - Race condition: Result of the computation depends on the exact timing of the two threads of execution, i.e., the order in which the instructions are executed
  - Reason: Concurrent <u>updates</u> to the same state
    - Can you get a race condition when all the threads are reading the data, and none of them are updating it ?

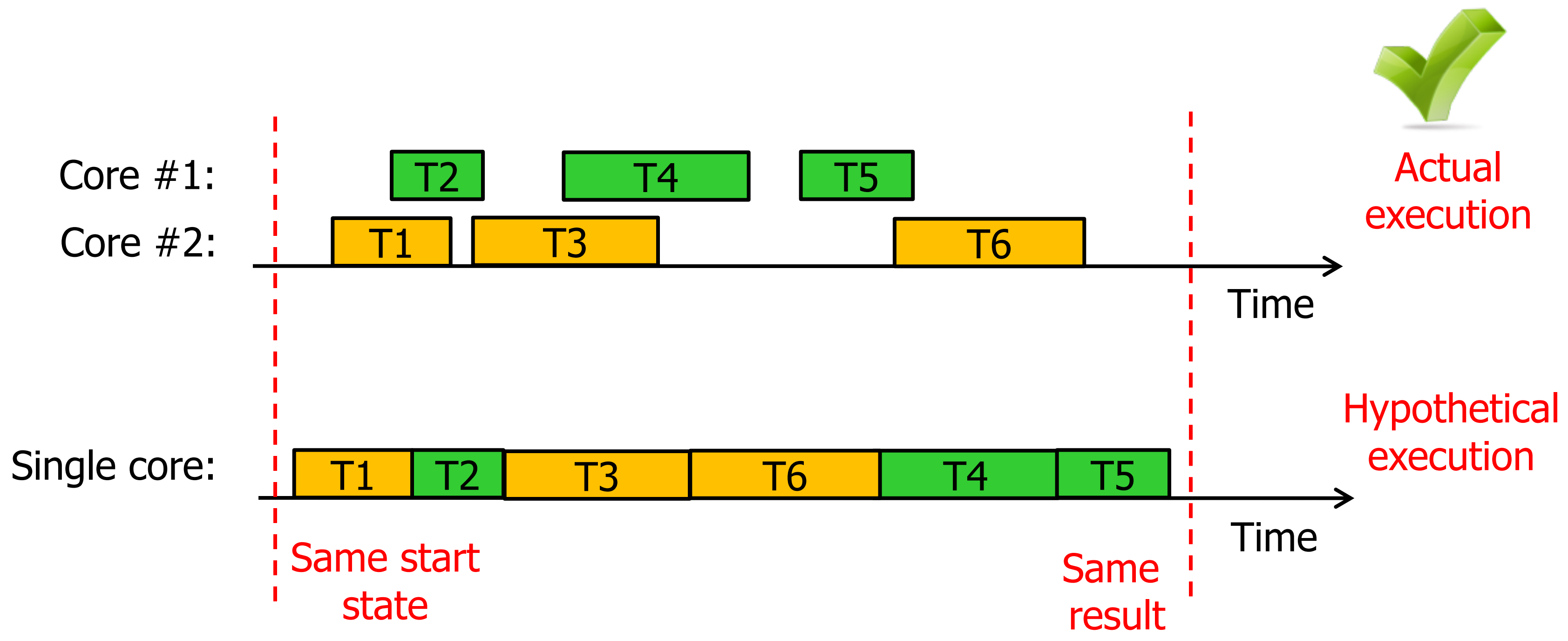
Goal: To Get a “Consistent” outcome

- What should have happened?
  - Intuition: It shouldn't make a difference whether the requests are executed concurrently or not
- How can we formalize this?
  - Need a consistency model that specifies how the system should behave in the presence of concurrency

A Common approach: Serializable executions of Transactions

- A “serial” execution is one in which there is at most one transaction running at a time, and it always completes via commit or abort before another starts
- “Serializability” is the “illusion” of a serial execution
  - Transactions execute concurrently and their operations interleave at the level of the database files
  - Yet database is designed to guarantee an outcome identical to some serial executions of the transactions: it masks concurrency
  - In past they used locking; these days “snapshot isolation”

Serialize Concurrent Transactions



Serialize Concurrent Transactions via Locking/ Mutual exclusion

Critical section

```
void transferMoney(customer A, customer B, int amount)
{
  showMessage("Transferring "+amount+" to "+B);
  int balanceA = getBalance(A);
  int balanceB = getBalance(B);
  setBalance(B, balanceB + amount);
  setBalance(A, balanceA - amount);
  showMessage("Your new balance: "+(balanceA-amount));
}
```

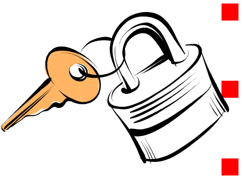
- How can we achieve better consistency?
  - Key insight: Code has a critical section where accesses from other codes (transactions) to the same resources will cause problems
- Approach: Mutual exclusion
  - Enforce restriction that only one core (or machine) can execute the critical section at any given time
  - What does this mean for scalability?

Locking

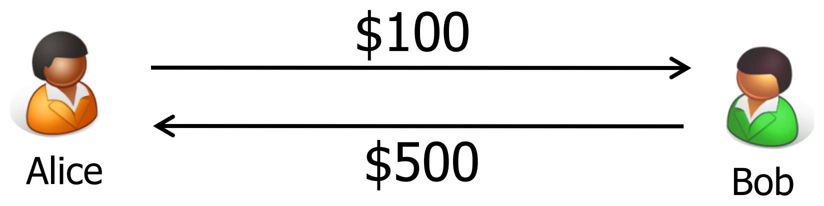
Critical section

```
void transferMoney(customer A, customer B, int amount)
{
  showMessage("Transferring "+amount+" to "+B);
  int balanceA = getBalance(A);
  int balanceB = getBalance(B);
  setBalance(B, balanceB + amount);
  setBalance(A, balanceA - amount);
  showMessage("Your new balance: "+(balanceA-amount));
}
```

- Idea: Implement locks
  - If LOCK(X) is called and X is not locked, lock X and continue
  - If LOCK(X) is called and X is locked, wait until X is unlocked
  - If UNLOCK(X) is called and X is locked, unlock X
- How many locks, and where do we put them?
  - Option #1: One lock around the critical section
  - Option #2: One lock per variable (A's and B's balance)
  - Pros and cons? Other options?

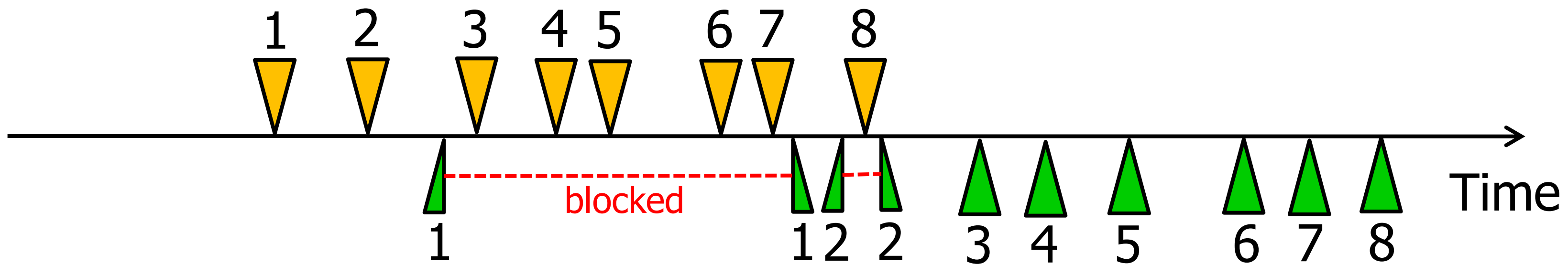


Locking helps!



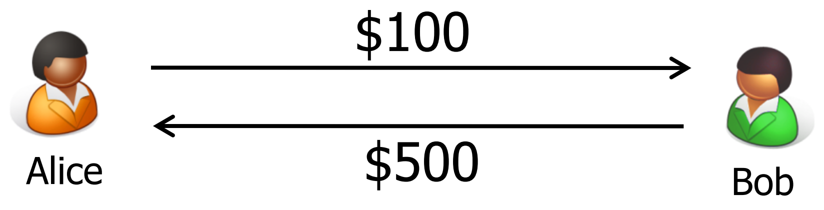
```
1) LOCK(Bob)
2) LOCK(Alice)
3) B=Balance(Bob)
4) A=Balance(Alice)
5) SetBalance(Bob,B+100)
6) SetBalance(Alice,A-100)
7) UNLOCK(Alice)
8) UNLOCK(Bob)
```

```
1) LOCK(Alice)
2) LOCK(Bob)
3) A=Balance(Alice)
4) B=Balance(Bob)
5) SetBalance(Alice,A+500)
6) SetBalance(Bob,B-500)
7) UNLOCK(Bob)
8) UNLOCK(Alice)
```



Alice's balance:	$200	$200	$100	$600	$600
Bob's balance:	$800	$900	$900	$900	$400

Problem of Locking: Deadlock

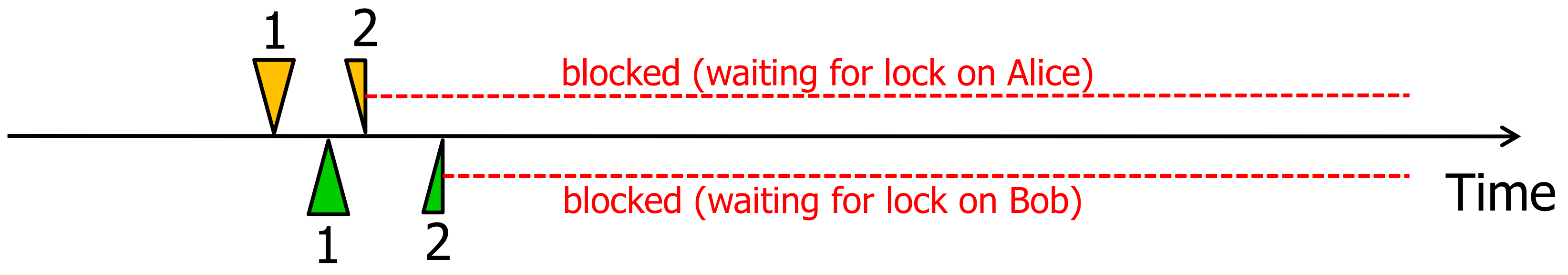


Alice:

```
1) LOCK(Bob)
2) LOCK(Alice)
3) B=Balance(Bob)
4) A=Balance(Alice)
5) SetBalance(Bob,B+100)
6) SetBalance(Alice,A-100)
7) UNLOCK(Alice)
8) UNLOCK(Bob)
```

Bob:

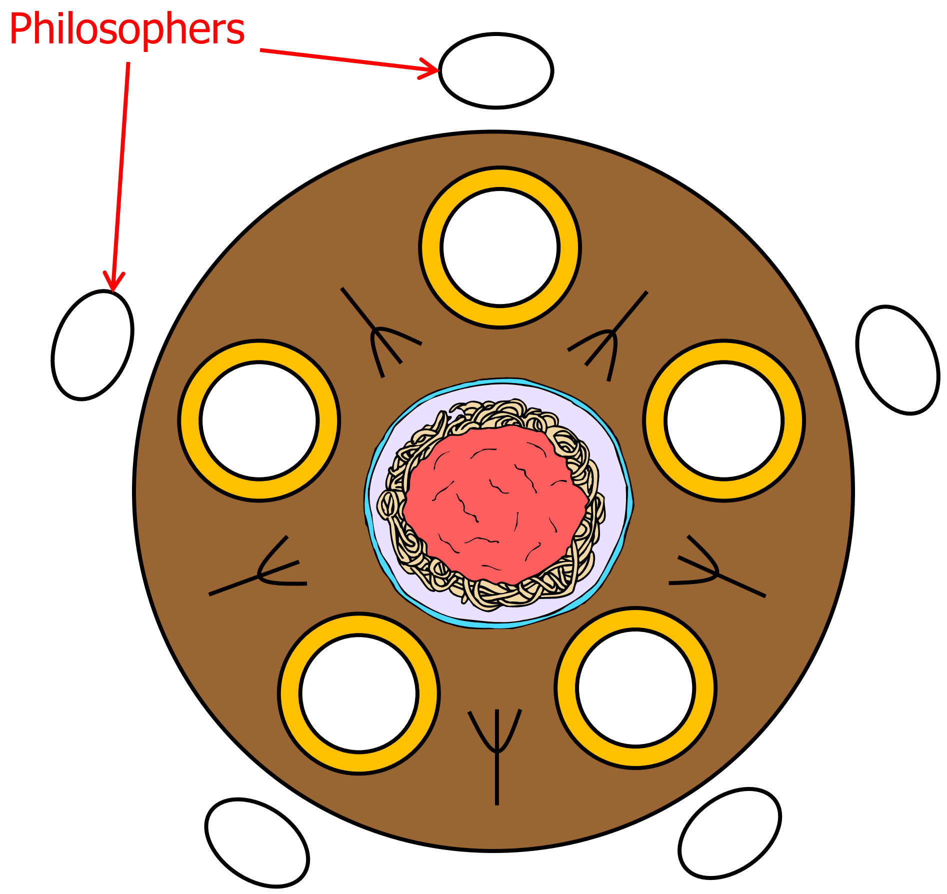
```
1) LOCK(Alice)
2) LOCK(Bob)
3) A=Balance(Alice)
4) B=Balance(Bob)
5) SetBalance(Alice,A+500)
6) SetBalance(Bob,B-500)
7) UNLOCK(Bob)
8) UNLOCK(Alice)
```



1
2
blocked (waiting for lock on Alice)
Time
1
2
blocked (waiting for lock on Bob)

- Neither processor can make progress!

The dining philosophers problem



```
Philosopher:

repeat
   think
   pick up left fork
   pick up right fork
   eat
   put down forks
forever
```

What to do about deadlocks

- Many possible solutions, including:
  - Lock manager: Hire a waiter and require that philosophers must ask the waiter before picking up any forks
    - Consequences for scalability?
  - Resource hierarchy: Number forks 1-5 and require that each philosopher pick up the fork with the lower number first
    - Problem?
  - Chandy/Misra solution:
    - Forks can either be dirty or clean; initially all forks are dirty
    - After the philosopher has eaten, all his forks are dirty
    - When a philosopher needs a fork he can't get, he asks his neighbor
    - If a philosopher is asked for a dirty fork, he cleans it and gives it up
    - If a philosopher is asked for a clean fork, he keeps it