CS4231
Parallel and Distributed Algorithms

Lecture 3

Instructor: Haifeng YU

Review of Last Lecture

- Why do we need synchronization primitives
  - Busy waiting waste CPU
  - But synchronization primitives need OS support
- Semaphore:
  - Using semaphore to solve dining philosophers problem
  - Avoiding deadlocks
- Monitor:
  - Easier to use than semaphores / more popular
  - Two kinds of monitors
  - Using monitor to solve producer-consumer and reader-writer problem

Today's Roadmap

- Chapter 4 "Consistency Conditions", Chapter 5.1, 5.2 "Wait-Free Synchronization"
- What is consistency? Why do we care about it?
- Sequential consistency
- Linearizability
- Consistency models for registers

Abstract Data Type

- Abstract data type: A piece of data with allowed operations on the data
  - Integer X, read(), write()
  - Integer X, increment()
  - Queue, enqueue(), dequeue()
  - Stack, push(), pop()

- We consider shared abstract data type
  - Can be accessed by multiple processes
  - Shared object as a shorthand

What Is Consistency?

- Consistency – a term with thousand definitions
- Definition in this course:
  - Consistency specifies what behavior is allowed when a shared object is accesses by multiple processes
  - When we say something is “consistent”, we mean it satisfies the specification (according to some given spec)
- Specification:
  - No right or wrong – anything can be a specification
- But to be useful, must:
  - Be sufficiently strong – otherwise the shared object cannot be used in a program
  - Can be implemented (efficiently) – otherwise remains a theory
  - Often a trade-off

In Lecture #1: Mutual Exclusion Problem with $x = x+1$

- Data type 1:
  - Integer x
  - read(), write()
- Data type 2:
  - Integer x
  - increment()
- We were using type 1 to implement type 2
  - By doing x = x+1;

process 0	*process 1*
read x into a register (value read: 0)	
increment the register (1)	we implicitly assumed the value read must be the value written – one possible def of consistency
write value in register back to x (1)	
	read x into a register (value read: 1)
	increment the register (2)
	write value in register back to x (2)

In Lecture #1: Mutual Exclusion Problem with $x = x+1$

- Data type 1:
  - Integer x
  - read(), write()
- Data type 2:
  - Integer x
  - increment()
- By saying the execution is not "good", we implicitly assumed some consistency definition for Data type 2

process 0	*process 1*
read x into a register (value read: 0)	
increment the register (1)	
	read x into a register (value read: 0)
	increment the register (1)
write value in register back to x (1)	
	write value in register back to x (1)

Why do we care about consistency?

- Mutual exclusion is actually a way to ensure consistency (based on some consistency definition)
  - Thus we actually already dealt with consistency…

- Our goal in the lecture
  - To formalize such consistency definition
  - Explore some other useful consistency definitions

Sequential Consistency

- First defined by Lamport: "…the results … is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program"
  - Use sequential order as a comparison point
  - Focus on results only – that is what users care about (what if we only allow sequential histories?)
  - Require that program order is preserved

- We need a lot of formalism to formalize the above…

Formalizing a Parallel System

- *Operation*: A single invocation/response pair of a single method of a single shared object by a process
  - *e* being an operation
  - proc(*e*): The invoking process
  - obj(*e*): The object
  - inv(*e*): Invocation event (*start time*)
  - resp(*e*): Reply event (*finish time*)

wall clock time / physical time / real time

- Two invocation events are the same if invoker, invokee, parameters are the same – inv(p, read, X)
- Two response events are the same if invoker, invokee, response are the same – resp(p, read, X, 1)

(Execution) History

- (The definitions in the book is based on operations are slightly harder to understand. We will use a different kind of definitions. But the two are equivalent.)
- A *history* H is a sequence of invocations and responses ordered by wall clock time
  - For any invocation in H, the corresponding response is required to be in H
  - Each execution of a parallel system corresponds to a history, and vice versa.

inv(p, read, X) inv(q, write, X, 1) resp(p, read, X, 0) resp(q, write, X, OK)

Sequential History and Concurrent History

- A history H is *sequential* if
  - Any invocation is always *immediately* followed by its response
  - I.e. No interleaving
  - Otherwise called *concurrent*

Sequential:

inv(p, read, X) resp(p, read, X, 0) inv(q, write, X, 1) resp(q, write, X, OK)

concurrent:

inv(p, read, X) inv(q, write, X, 1) resp(p, read, X, 0) resp(q, write, X, OK)

Sequential Legal History and Subhistory

- A sequential history H is *legal* if
  - All responses satisfies the *sequential semantics* of the data type
  - *Sequential semantics*: The semantics you would get if there is only one process accessing that date type.
  - It is possible for a sequential history not to be legal

- Process p's *process subhistory* of H (H | p):
  - The subsequence of all events of p
  - Thus a process subhistory is always sequential
- Object o's object *subhistory* of H (H | o):
  - The subsequence of all events of o

Equivalency and Process Order

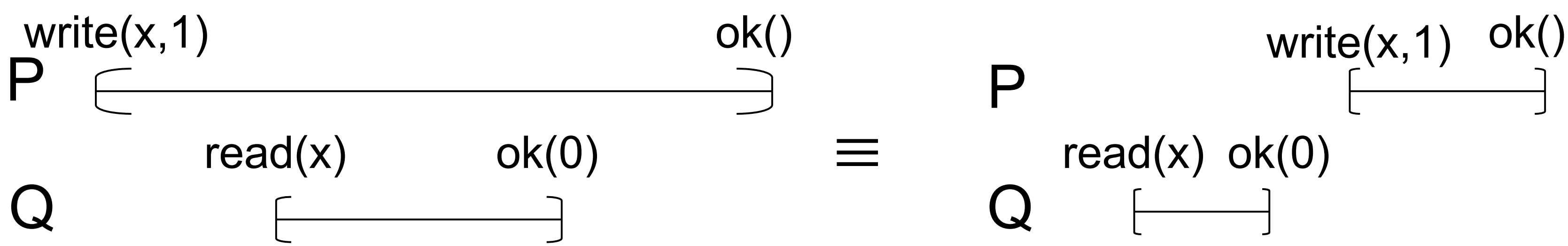
- Two histories are *equivalent* if they have the exactly same set of events
  - Same events imply all responses are the same
  - Ordering of the events may be different
  - User only cares about responses

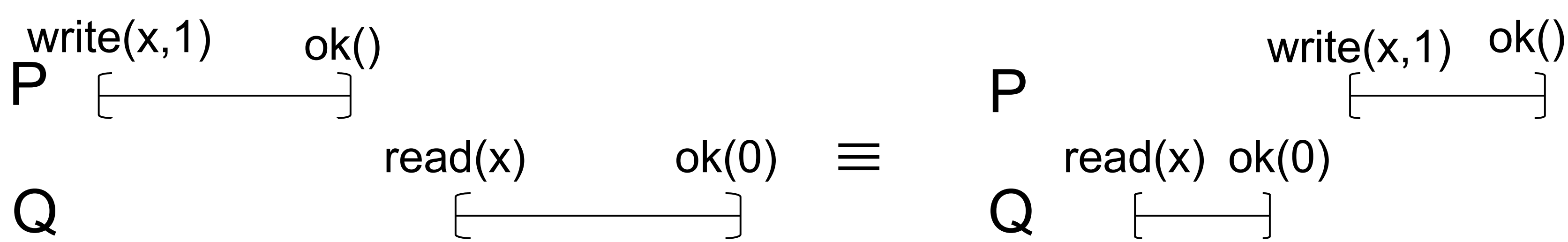
- *Process order* is a partial order among all events
  - For any two events of the same process, process order is the same as execution order
  - No other additional orderings

Sequential Consistency

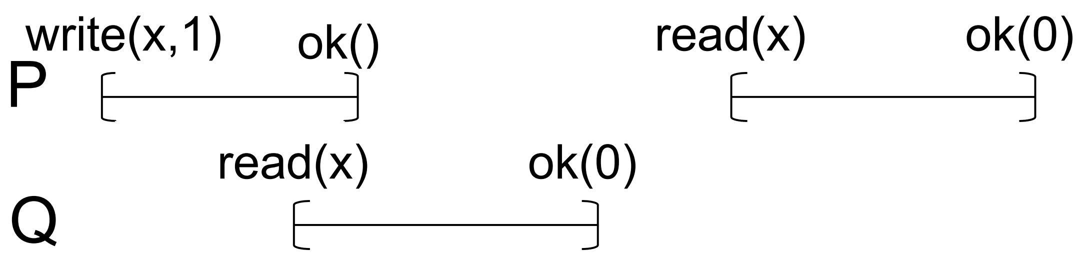
- A history H is sequentially consistent if it is equivalent to some legal sequential history S that preserves process order
  - Use sequential history as a comparison point
  - Focus on results only – that is what users care about (what if we only allow sequential histories?)
  - Require that program order be preserved

Examples

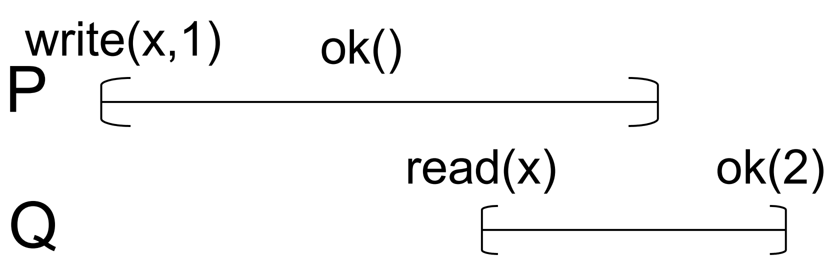

write(x,1)
ok()
P
read(x)
ok(0)
Q
≡
write(x,1)
ok()
P
read(x)
ok(0)
Q


write(x,1)
ok()
P
read(x)
ok(0)
Q
≡
write(x,1)
ok()
P
read(x)
ok(0)
Q

Examples



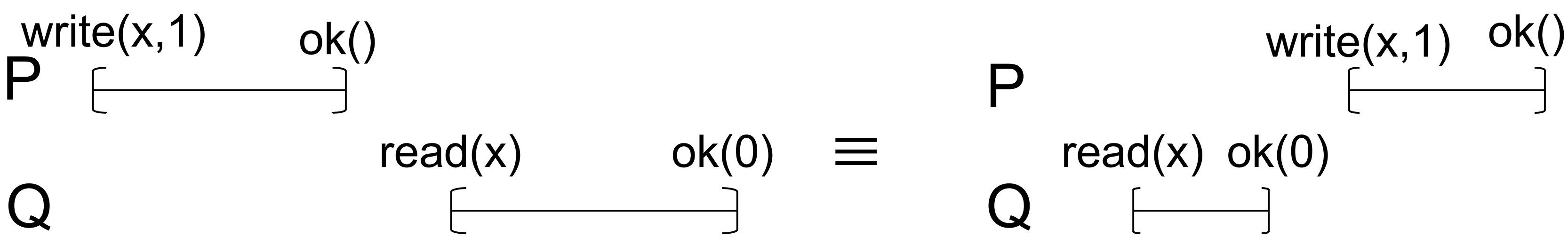
≡ ?



≡ ?

Motivation for Linearizability

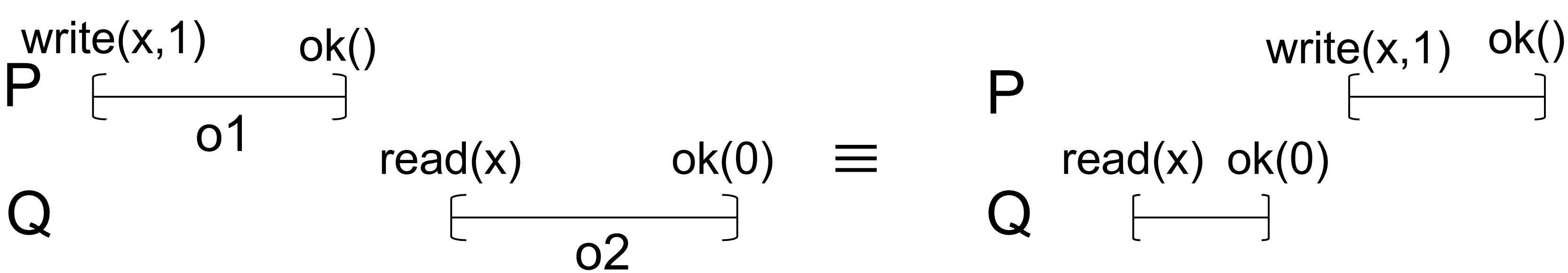
- Sequential consistency arguably the most widely used consistency definition
  - E.g. Almost all commercial databases
  - Many multi-processors
- But sometimes not strong enough



More Formalism

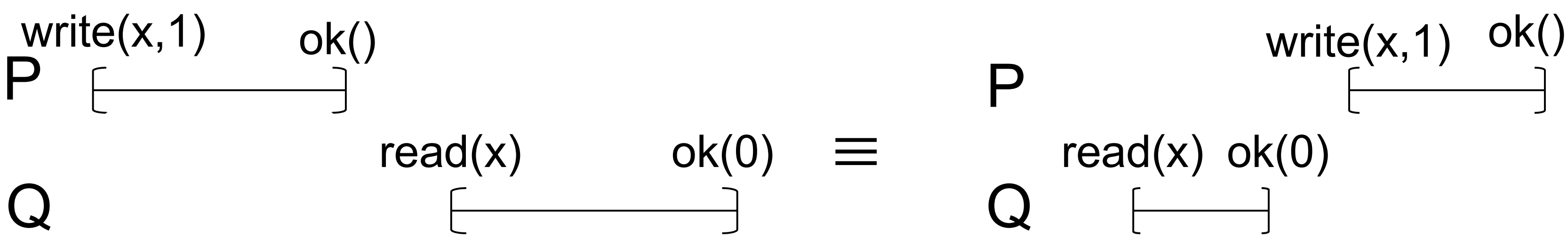
- A history H uniquely induces the following “<“ partial order among operations
  - o1 “<“ o2 iff response of o1 appears in H before invocation of o2
  - We call this *external order* (the textbook calls this *occurred-before order*)

- What is the external order induced by a sequential history?



Linearizability

- Definition #1: The execution is equivalent to some execution such that each operation happens instantaneously at some point (called linearization point) between the invocation and response
  - Can be made rigorous – will be your homework



The above history is sequentially consistent but not linearizable

Linearizability

- Definition #1 (repeated): The execution is equivalent to some execution such that each operation happens instantaneously at some point between the invocation and response

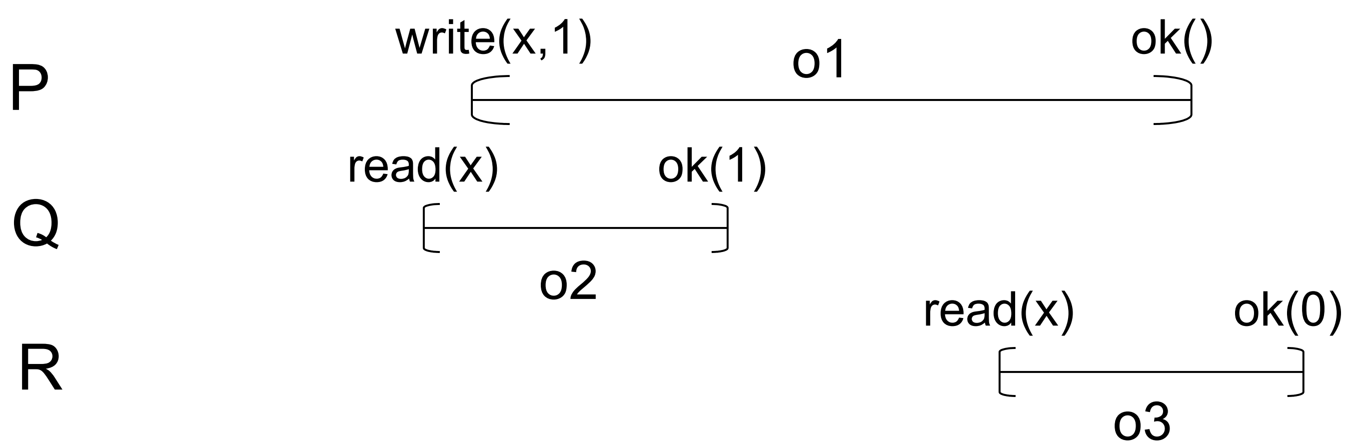
- Definition #2: History H is linearizable if
  - It is equivalent to some legal sequential history S, and
  - S preserves the external order in H – namely, the partial order induced by H is a subset of the partial order induced by S

- Our textbook uses definition #2

- Are the two definitions the same? This question will be your homework today…

Linearizability

- A linearizable history must be sequentially consistent
  - Linearizability is arguable the strongest form of consistency people have ever defined



Another interesting example

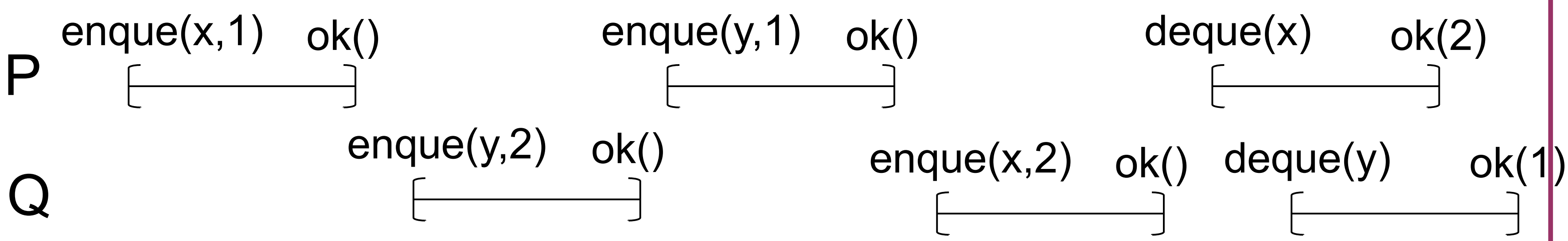
Local Property

- Linearizability is a local property in the sense that H is linearizable if and only if for any object *x*, H | *x* is linearizable

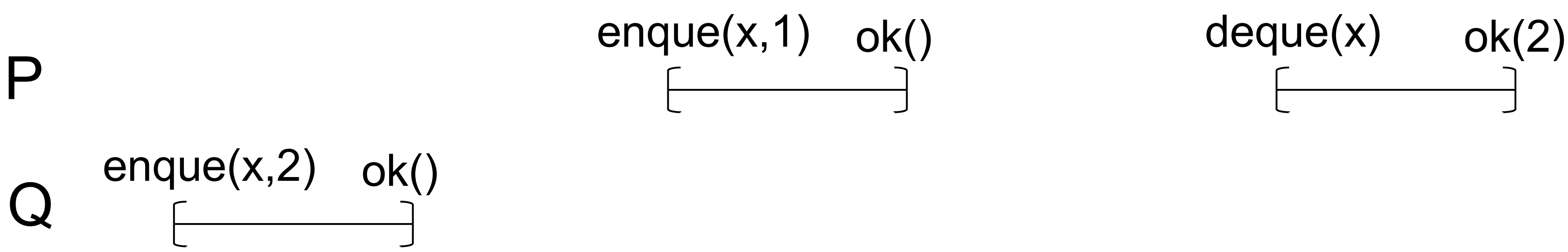
- Sequential consistency is not a local property

Sequential Consistency is Not Local Property

x, y are initially empty queues



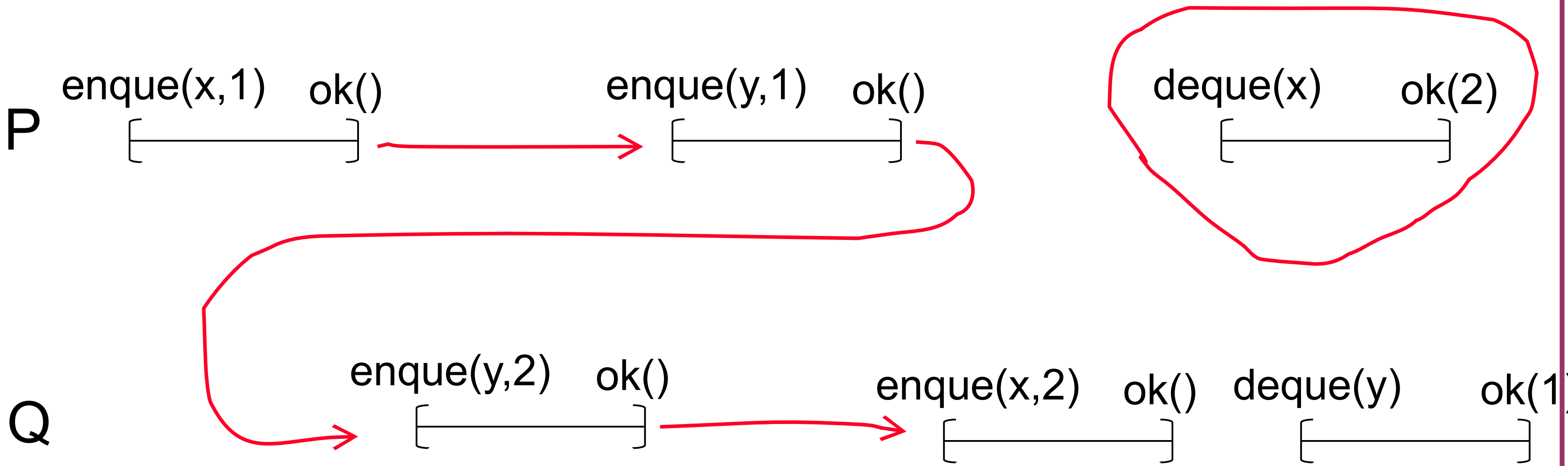
H | x is sequentially consistent:



Similarly, H | y is sequentially consistent as well

Sequential Consistency is Not Local Property

x, y are initially empty queues



But H is not sequentially consistent

Proving That Linearizability is a Local Property

- Definition #1 (repeated): The execution is equivalent to some execution such that each operation happens instantaneously at some point between the invocation and response

- The proof is trivial if we use the above definition (after formalizing it)

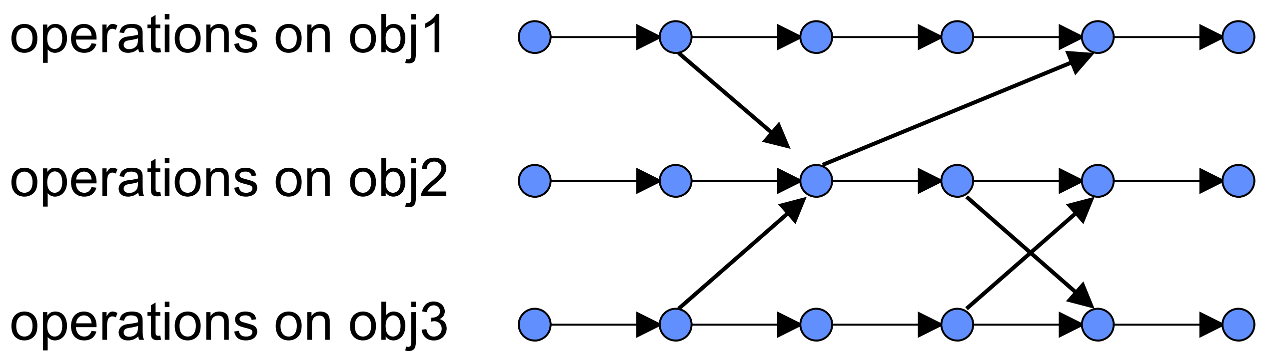
- But we'll prove the statement while using Definition #2 (i..e, the definition from our textbook…)

Proving That Linearizability is a Local Property

- Definition #2 (repeated): A history H is linearizable if
  1. It is equivalent to some legal sequential history S, and
  2. S preserves the external order in H

- Want to show H is linearizable if for any object x, H | x is linearizable
- We construct a directed graph among all operations as following: A directed edge is created from o1 to o2 if
  - o1 and o2 is on the same obj x and o1 is before o2 when linearizing H | x (we say o1→o2 due to obj), OR
  - o1 < o2 in external order (i.e., response of o1 appears before invocation of o2 in H) (we say o1→o2 due to H)

Proving That Linearizability is a Local Property



- Lemma: The resulting directed graph is acyclic (we prove this later)
- Any topological sorting of the above graph gives us a legal sequential history S, and the edges in the graph is a subset of the total order induced by S – Theorem proved

Linearizability definition:

1. H is equivalent to some legal sequential history S, and
2. S preserves the external order in H

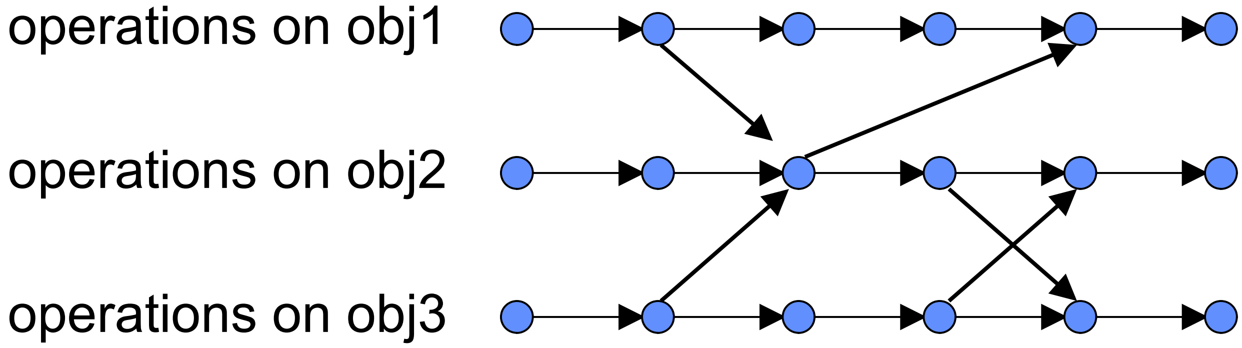
Legal + Legal = Legal

- H | x is linearizable, and let the linearization be O1 O2
- H | y is linearizable, and let the linearization be O3 O4
- Suppose the topological sort be O1 O3 O2 O4

- O1 O2 is legal (for x), O3 O4 is legal (for y)
- Is S = O1 O3 O2 O4 legal (for both x and y)?
  - In particular, we are inserting O3 between O1 and O2, will O2's behavior change due to some value read in O3?

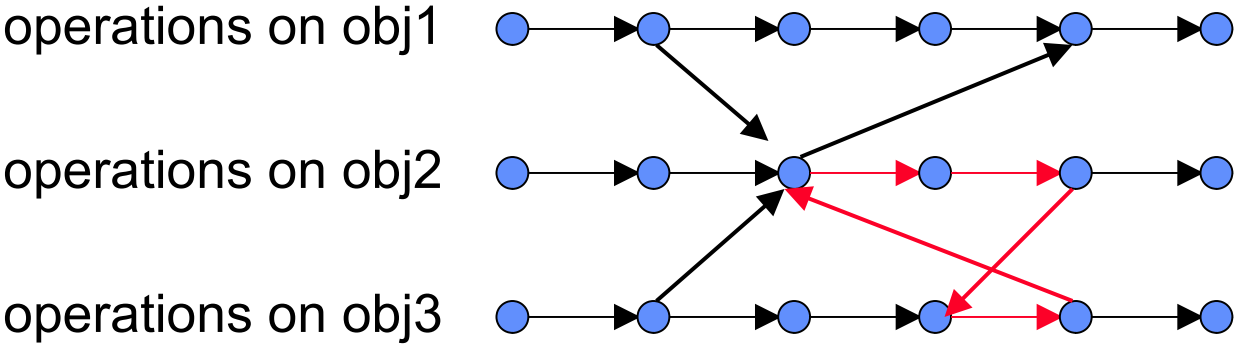
- All events in S are from H – including all invocation events and their parameters
  - So if in H, the program code updates y based on the value of x, the new value for y will already be captured in the parameters of the corresponding invocation event – we do not re-run the program code

Lemma: The resulting graph does not have a cycle



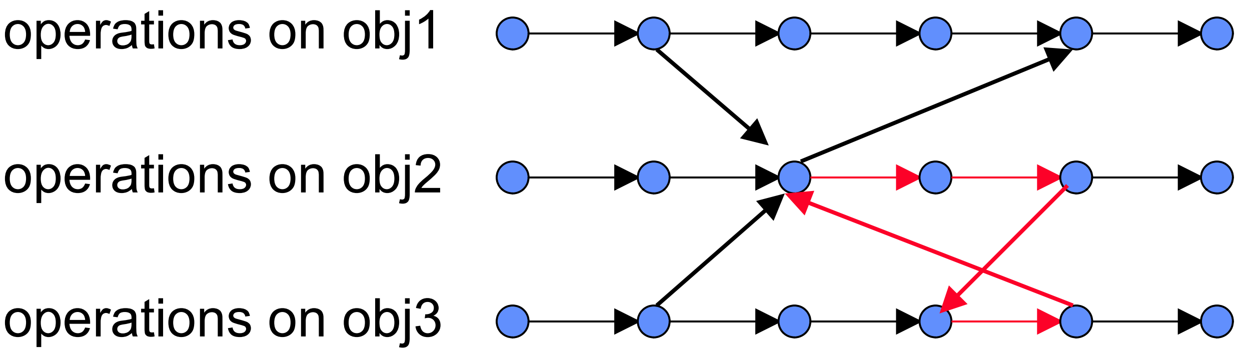
- Prove by contradiction

Lemma: The resulting graph does not have a cycle



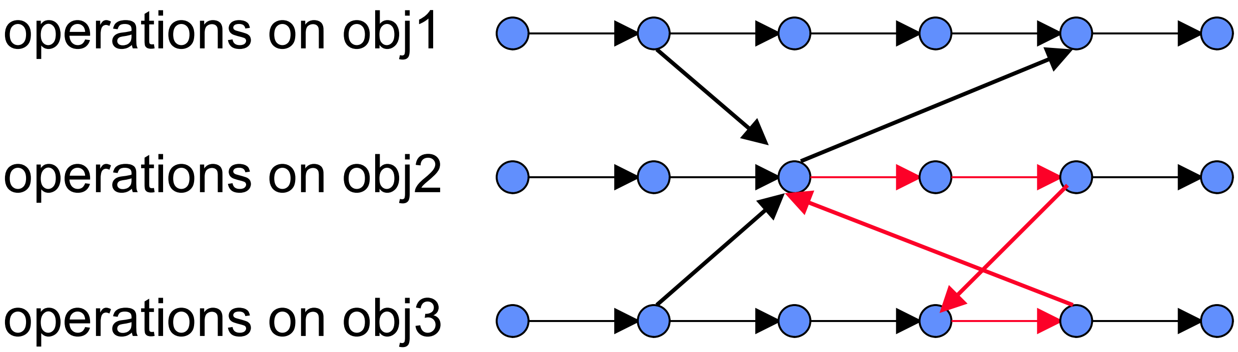
- Example: The red cycle is composed of
  - Edges due to obj3, followed by
  - Edges due to H, followed by
  - Edges due to obj2, followed by
  - Edges due to H, followed by

Lemma: The resulting graph does not have a cycle



- Impossible to have
  - Edges due to some object x, followed by
  - Edges due to some object y

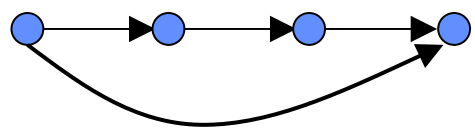
Lemma: The resulting graph does not have a cycle



- To generalize, any cycle must be composed of:
  - Edges due to some object x, followed by
  - Edges due to H, followed by
  - Edges due to some object y, followed by
  - Edges due to H, followed by
  - ….

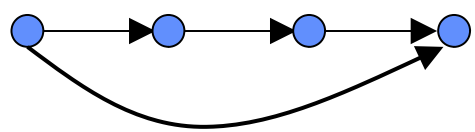
Lemma: The resulting graph does not have a cycle

edges due to x



H | x is equivalent to a serial history S – S is a total order

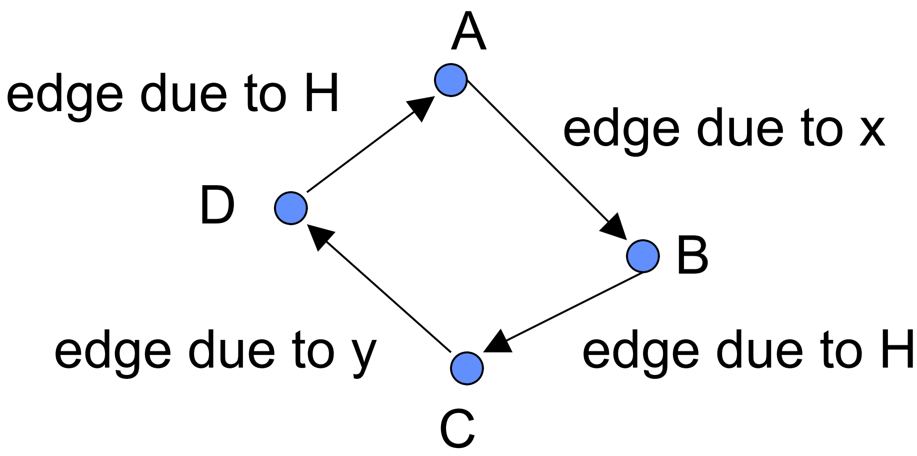
edges due to H



the partial order induced by H is transitive

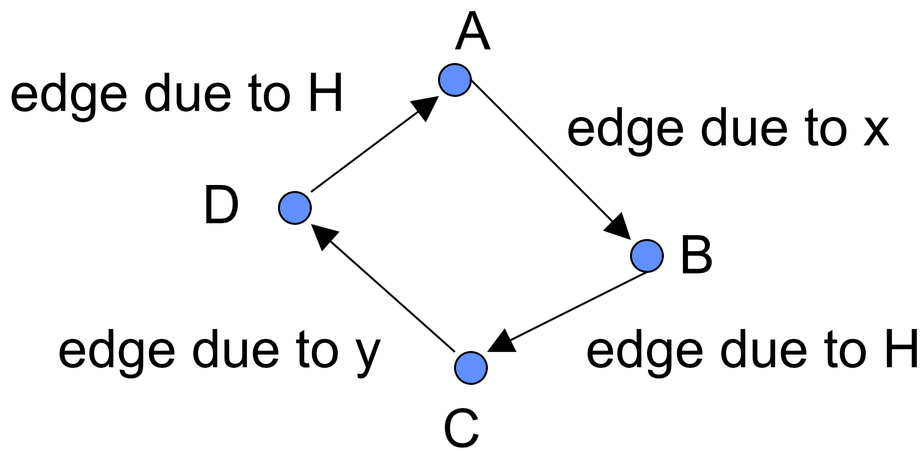
- To generalize, any cycle must be composed of:
  - Edges due to some object x $\Rightarrow$ A single edge due to x
  - Edges due to H $\Rightarrow$ A single edge due to H
  - Edges due to some object y $\Rightarrow$ A single edge due to y
  - Edges due to H $\Rightarrow$ A single edge due to H
  - ….

Lemma: The resulting graph does not have a cycle

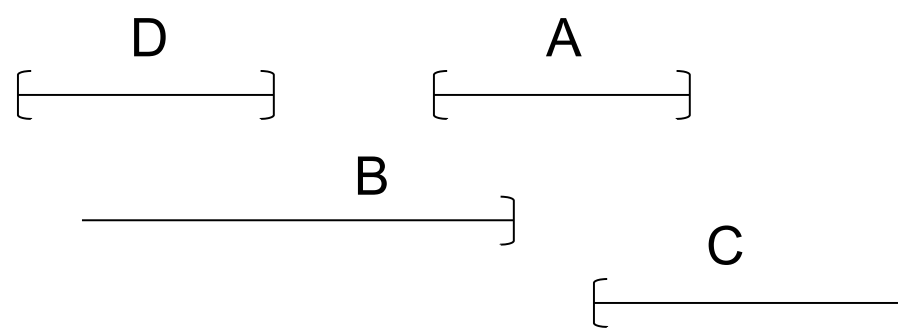


- We must have a cycle in the form of
  - A single edge due to x, followed by
  - A single edge due to H, followed by
  - A single edge due to y, followed by
  - A single edge due to H , followed by
  - ….

Lemma: The resulting graph does not have a cycle



in H:



From the edge D→A:

From the edge A→B:

From the edge B→C:

It is impossible to have C→D due to y

Consistency Definitions for Registers

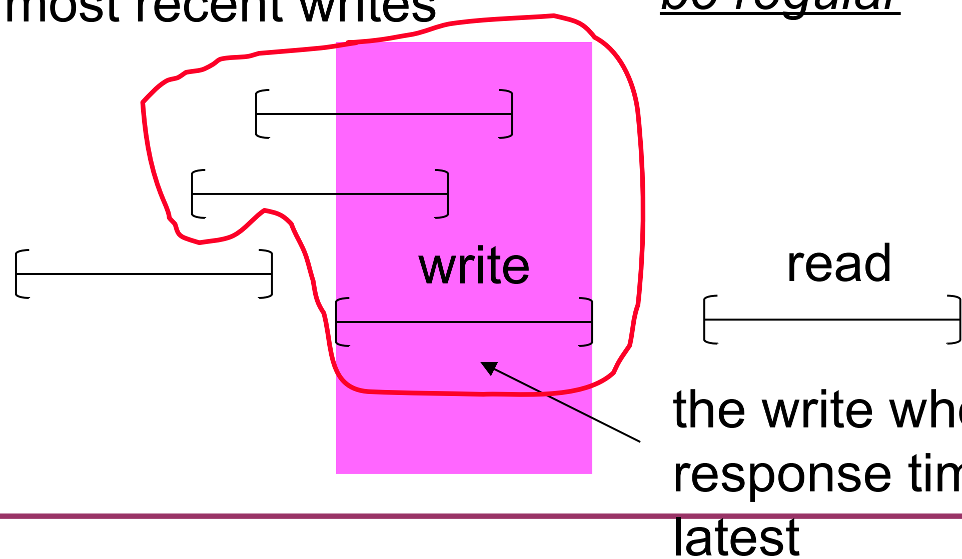
- Register is a kind of abstract data type: A single value that can be read and written
- An (implementation of a) register is called atomic if the implementation always ensures linearizability of the history
- An (implementation of a) register is called ?? if the implementation always ensures sequential consistency of the history

Consistency Definitions for Registers

- An (implementation of a) register is called regular if
  - When a read does not overlap with any write, the read returns the value written by one of the most recent writes
  - When a read overlaps with one or more writes, the read returns the value written by one of the most recent writes or the value written by one of the overlapping writes

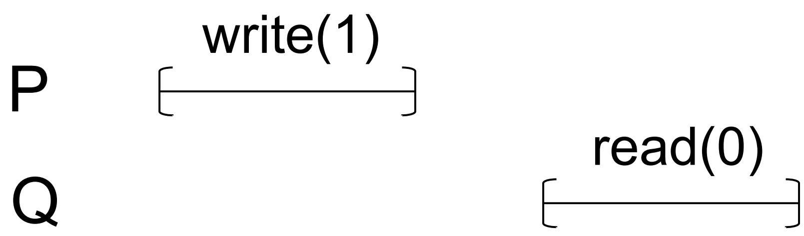
An atomic register must be regular

- Definition of most recent writes

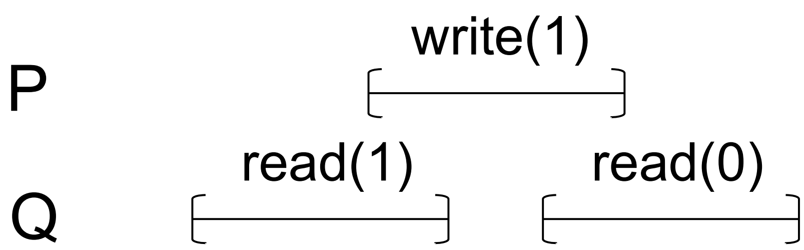


Regular → Sequential Consistency ?
Sequential Consistency → Regular ?

assuming the initial value of the register is 0



Sequentially consistent but not regular



Regular but not sequentially consistent

Consistency Definitions for Registers

- An (implementation of a) register is called safe if the implementation always ensures that
  - When a read does not overlap with any write, then it returns the value written by one of the most recent writes
  - When a read overlaps with one or more writes, it can return anything

A regular register hence must be safe

- Safe $\rightarrow$ Sequential Consistency ?
- Sequential Consistency $\rightarrow$ Safe ?

Summary

- What is consistency? Why do we care about it?
- Sequential consistency
- Linearizability
  - Linearizability is a local property
- Consistency models for registers

Homework Assignment

- Page 62:
  - Problem 4.1
  - If you believe a history is linearizable/serializable, give the equivalent sequential history
  - If you believe a history is not linearizable/serializable, prove that no equivalent sequential history exists
- Slide 26: Prove that linearizability is a local property using the definition on slide 26 (formalize the definition first)
- Think about Slide 21: Prove that the two definitions of linearizability are equivalent
- Homework due a week from today

- Read Chapter 7