SFU SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

CMPT 431 Distributed Systems

Fall 2019

Reasoning Correctness

https://www.cs.sfu.ca/~keval/teaching/cmpt431/fall19/

Instructor: Keval Vora

Concurrent Programs

- How to design concurrent programs?
- What does concurrency mean when ordering is important?
  - E.g., concurrent queues, concurrent stacks
- How to verify whether concurrent solution is correct?
  - Naïve: Check for all possible results. Infeasible.

- Properties that can help in:
  - developing concurrent solutions
  - analyzing concurrent solutions
  - provide correctness and progress guarantees

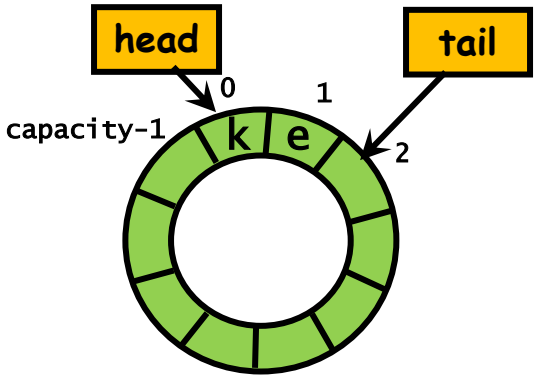
Reading

- [AMP] Chapter 3
  - Upto 3.7

- [Paper] Linearizability: A Correctness Condition for Concurrent Objects: https://cs.brown.edu/~mph/HerlihyW90/p463-herlihy.pdf
  - Upto section 3

- [Paper] How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs: https://www.microsoft.com/en-us/research/uploads/prod/2016/12/How-to-Make-a-Multiprocessor-Computer-That-Correctly-Executes-Multiprocess-Programs.pdf

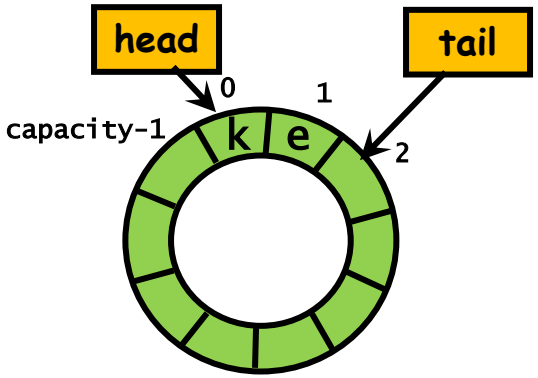
Single-Enqueuer/Single-Dequeuer Queue



- Sequential code is available

```
1  class Queue<T> {
2    volatile int head = 0, tail = 0;
3    T[] items;
4    public Queue(int capacity) {        {
5      items = (T[])new Object[capacity];
6      head = 0; tail = 0;
7    }
8    public void enq(T x) throws FullException {
9      if (tail - head == items.length)
10       throw new FullException();
11     items[tail % items.length] = x;
12     tail++;
13   }
14   public T deq() throws EmptyException {
15     if (tail - head == 0)
16       throw new EmptyException();
17     T x = items[head % items.length];
18     head++;
19     return x;
20   }
21 }
```

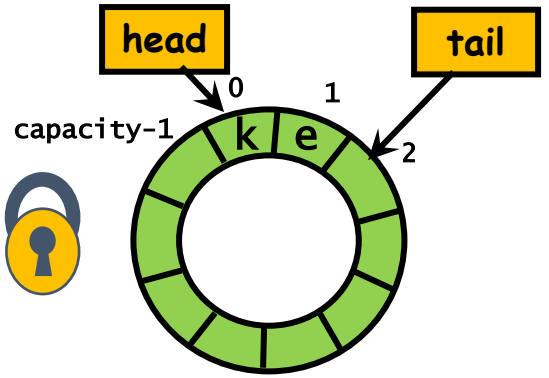
Single-Enqueuer/Single-Dequeuer Queue



- Concurrent code is same!
  - How is it correct?
  - enq() & deq() can happen at same time?

```
1  class WaitFreeQueue<T> {
2    volatile int head = 0, tail = 0;
3    T[] items;
4    public WaitFreeQueue(int capacity) {
5      items = (T[])new Object[capacity];
6      head = 0; tail = 0;
7    }
8    public void enq(T x) throws FullException {
9      if (tail - head == items.length)
10       throw new FullException();
11     items[tail % items.length] = x;
12     tail++;
13   }
14   public T deq() throws EmptyException {
15     if (tail - head == 0)
16       throw new EmptyException();
17     T x = items[head % items.length];
18     head++;
19     return x;
20   }
21 }
```

Lock-based Queue




```
1   class LockBasedQueue<T> {
2     int head, tail;
3     T[] items;
4     Lock lock;
5     public LockBasedQueue(int capacity) {
6       head = 0; tail = 0;
7       lock = new ReentrantLock();
8       items = (T[])new Object[capacity];
9     }
10    public void enq(T x) throws FullException {
11      lock.lock();
12      try {
13        if (tail - head == items.length)
14          throw new FullException();
15        items[tail % items.length] = x;
16         tail++;
17      }  finally {
18        lock.unlock();
19      }
20    }
```

```
21    public T deq() throws EmptyException {
22      lock.lock();
23      try {
24        if (tail == head)
25          throw new EmptyException();
26        T x = items[head % items.length];
27        head++;
28        return x;
29      } finally {
30        lock.unlock();
31      }
32    }
33  }
```

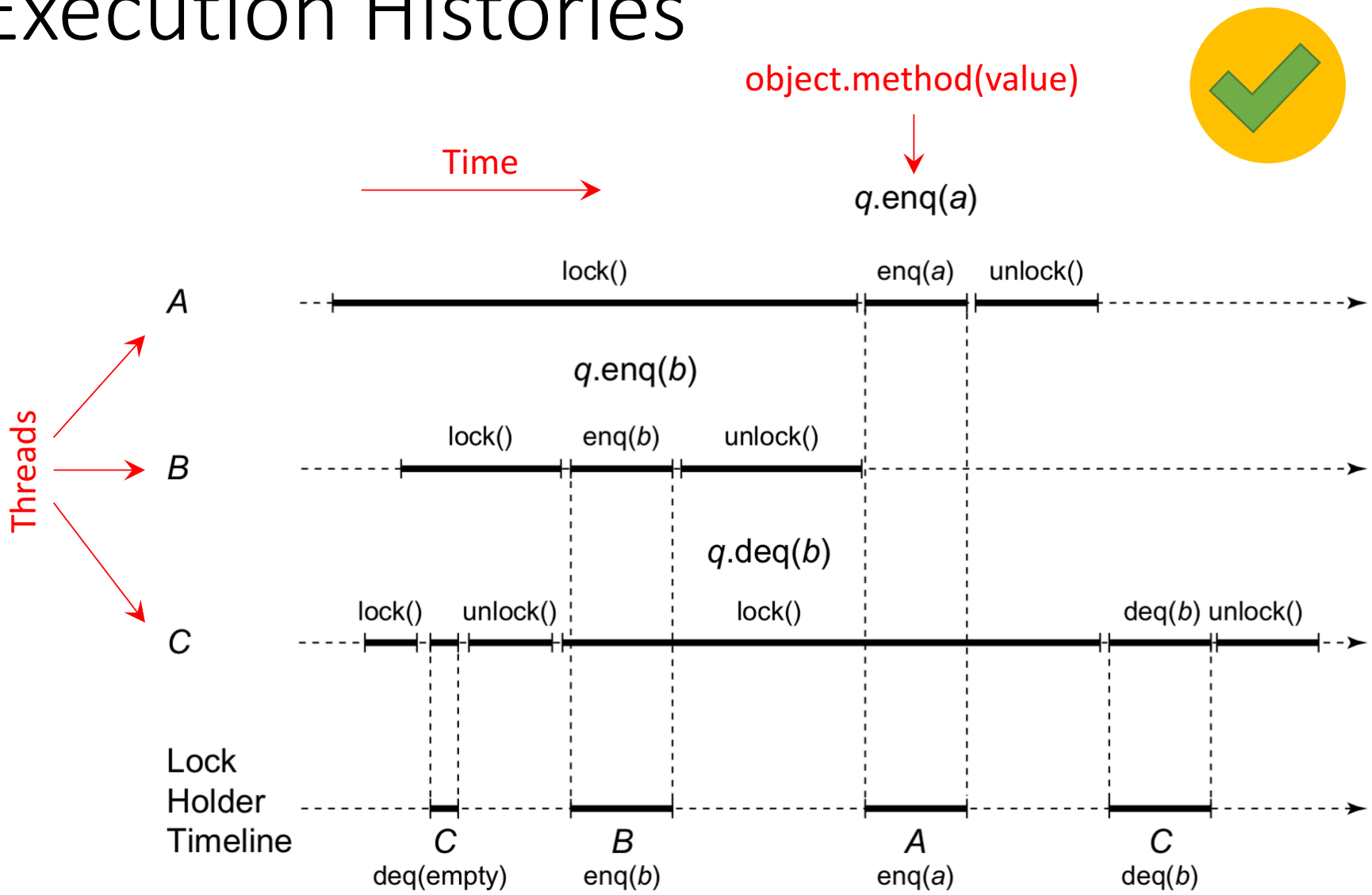
Easy to reason correctness:
no two operations happen at the **`same time'**

Operations & Execution Histories

- Operations defined based on the context
  - Queue example: enq() and deq() are primary operations
- Operations are ones that are performed by concurrent tasks, and that we want to analyze

- Plot Execution Histories on timelines to visually analyze when operations (can) occur
  - Execution Histories: finite sequence of operations

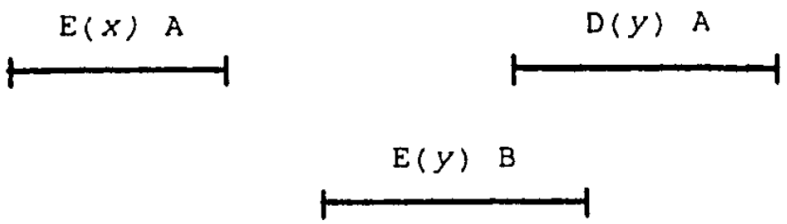
Execution Histories

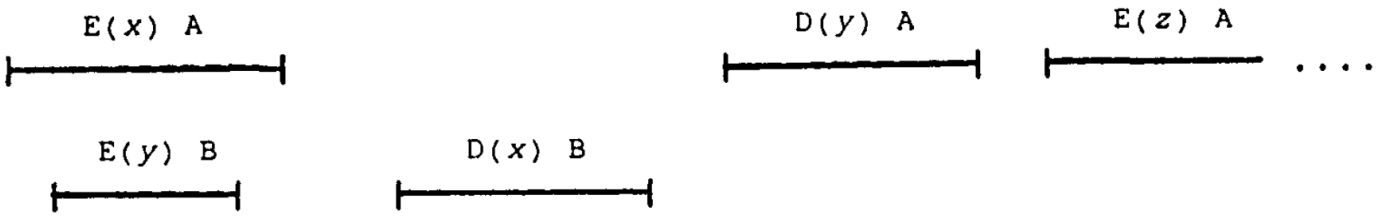


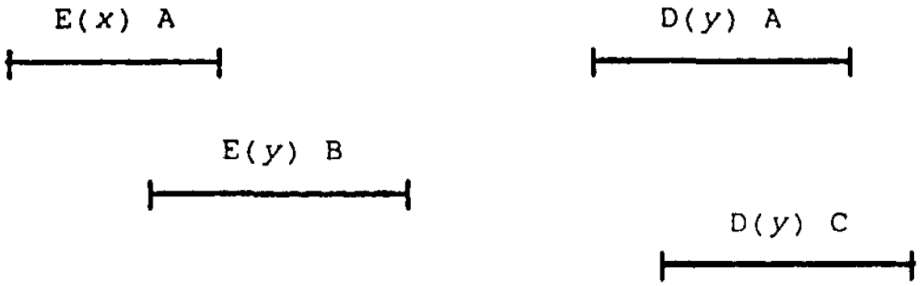
Execution Histories

operation(value) Thread

E = Enqueue
D = Dequeue







Correctness Conditions

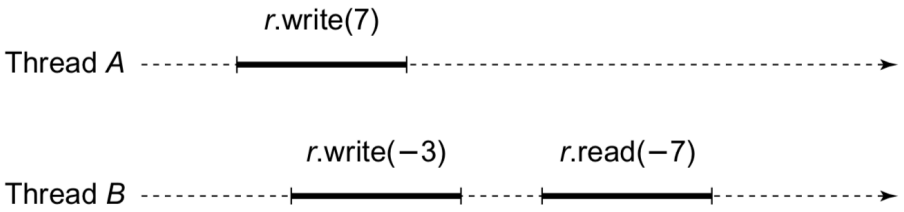
- Defining the set of allowable histories
- "allowable" → Depends on the context
- Degree of concurrency directly affected
  - Allow more histories → higher concurrency

- Sequential Consistency
- Linearizability
- (Serializability)

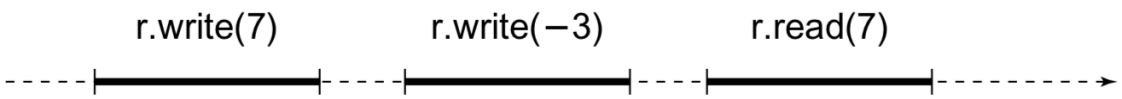
Sequential Executions

- Sequential methods have clear specifications in form of preconditions and postconditions
- With concurrency, multiple threads might invoke methods at the same time, making it difficult to reason about preconditions and postconditions in "isolation"
  - E.g., what does it mean for two overlapping enqueue operations? Which element will be dequeued first?
- Solution: show equivalence of histories

Method calls should appear to happen in a one-at-a-time sequential order



Sequential Consistency



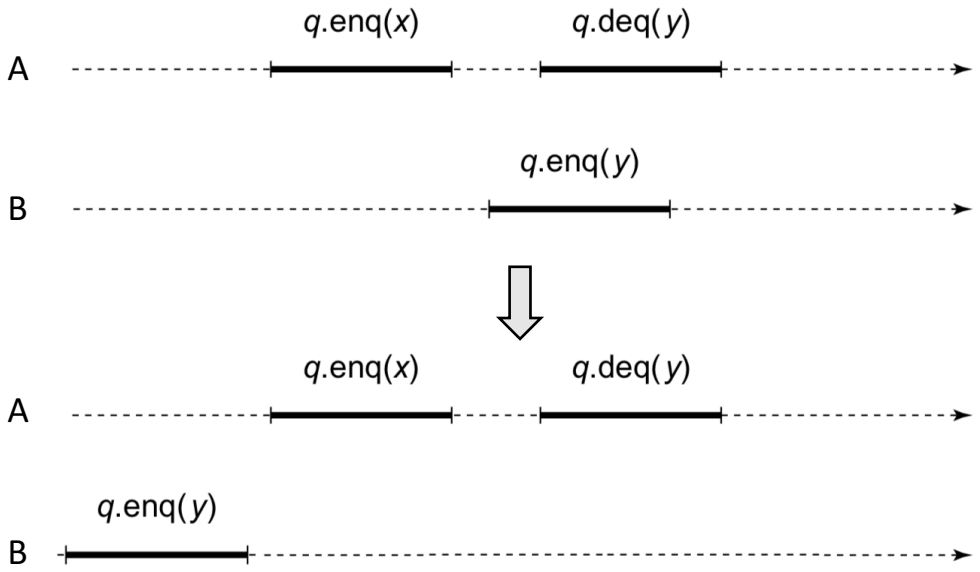
- [P1] Method calls should appear to happen in a one-at-a-time sequential order
- [P2] Method calls should appear to take effect in program order (i.e., order defined by single thread's code)
  - Purely sequential computations behave the way we would expect

- To verify, reorder method calls sequentially so that:
  - they are consistent with program order
  - meet the object's sequential specification

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, Leslie Lamport, IEEE TC 1979

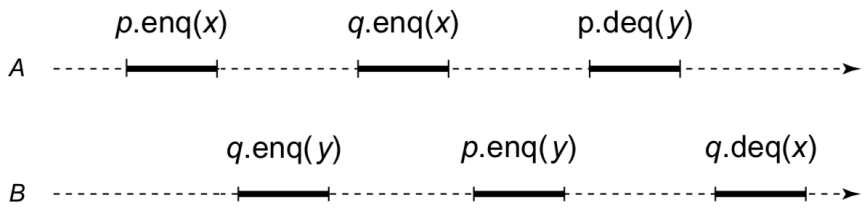
Sequential Consistency

- Counter-intuitive because it seems to violate FIFO



- Note: concurrent operations are allowed
  - as long as the two properties are satisfied

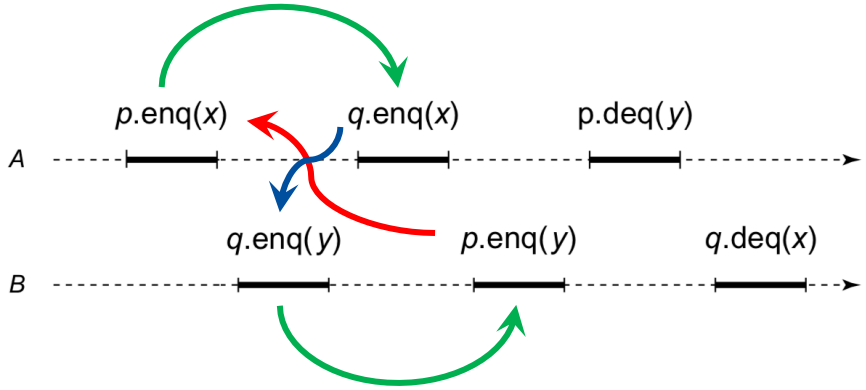
Sequential Consistency: Compositional?



- Is the sub-history for p sequentially consistent?
  - Reorder so that p.enq(y) comes before p.enq(x)
- Is the sub-history for q sequentially consistent?
  - Reorder so that q.enq(x) comes before q.enq(y)

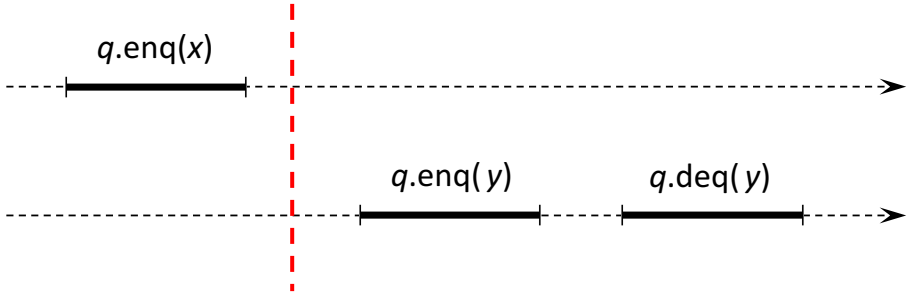
- Is the entire history sequentially consistent?
  - Need to explicitly check because SC is not compositional

Sequential Consistency: Not Compositional



- Is the entire history sequentially consistent?
  - p.deq(y) suggests: p.enq(y) → p.enq(x) [red arrow]
  - q.deq(x) suggests: q.enq(x) → q.enq(y) [blue arrow]
  - Program order [green arrows]
    q.enq(y) → p.enq(y) and p.enq(x) → q.enq(x)
  - Above orderings form a cycle (i.e., cannot be sequential)

Capturing Real-Time Precedence

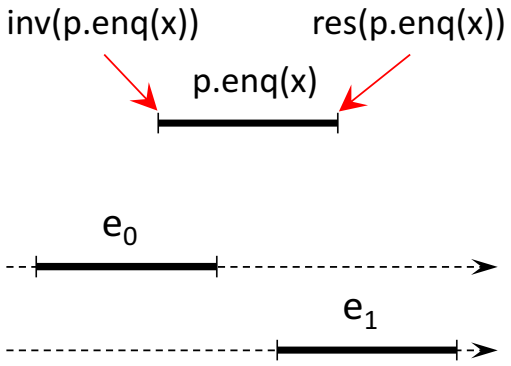


- Above history is sequentially consistent
- The real-time precedence ordering: q.enq(x) → q.enq(y)
- Intuition
  - Updates should become visible
  - deq operation should return x and not y
- Sequential consistency fails to capture real-time progress

Linearizability

- Captures real-time precedence ordering
- Informally: Operations should take effect instantaneously between its invocation and response

- Definitions:
  - inv(e): invocation of event e
  - res(e): response of event e
  - $e_0 <_H e_1$: $res(e_0)$ precedes $inv(e_1)$ in H
- $<_H$ is a partial order
  - Operations unrelated by $<_H$ are said to be concurrent



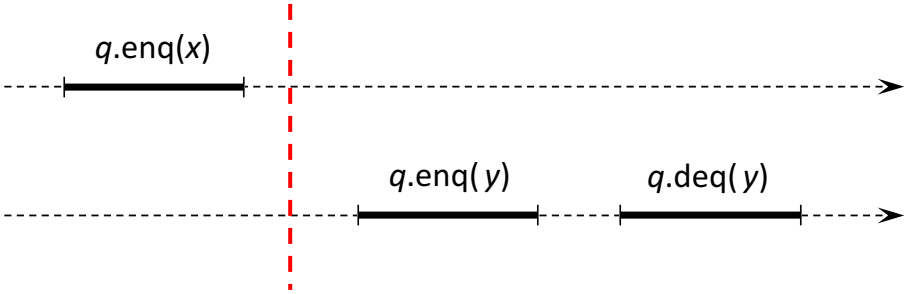
Linearizability: A Correctness Condition for Concurrent Objects, M. Herlihy and J. Wing, 1990

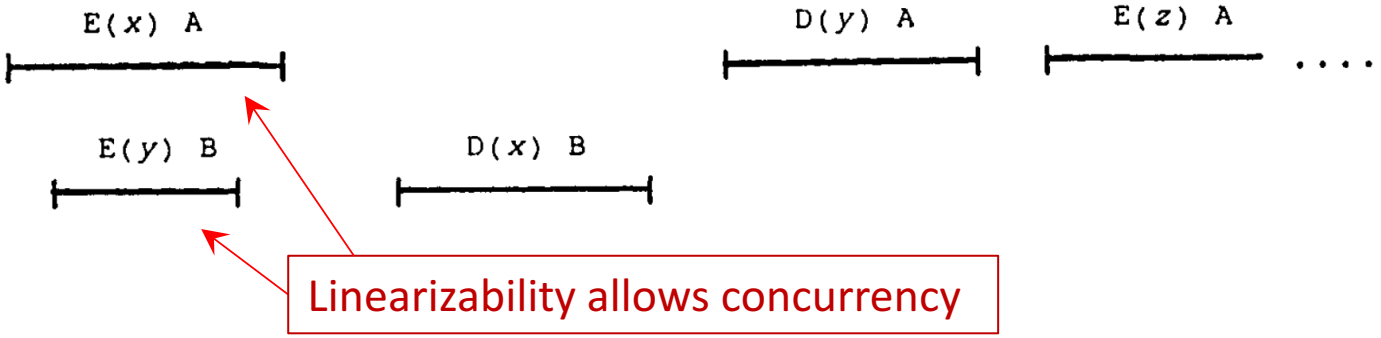
Linearizability

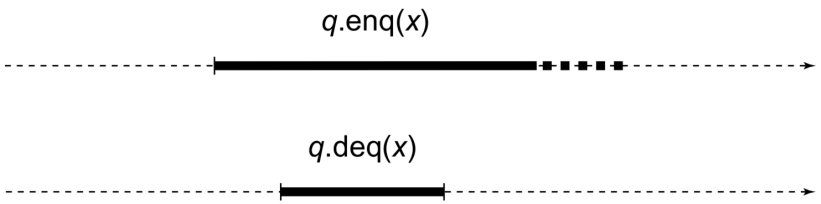
- [P1] Equivalent to some legal sequential history S
  (informally same as [P1] for SC)
- [P2] $<_H \subseteq <_S$
  (informally, real-time precedence ordering should be preserved)

- Note: equivalence between complete(H') and S
  - Read text/paper for formalisms

Linearizability

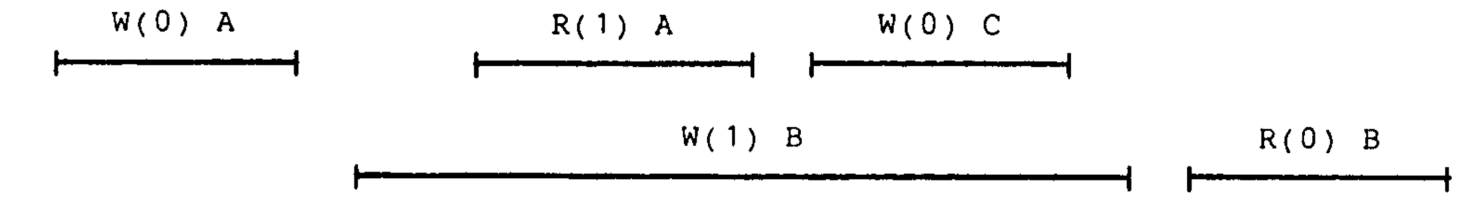


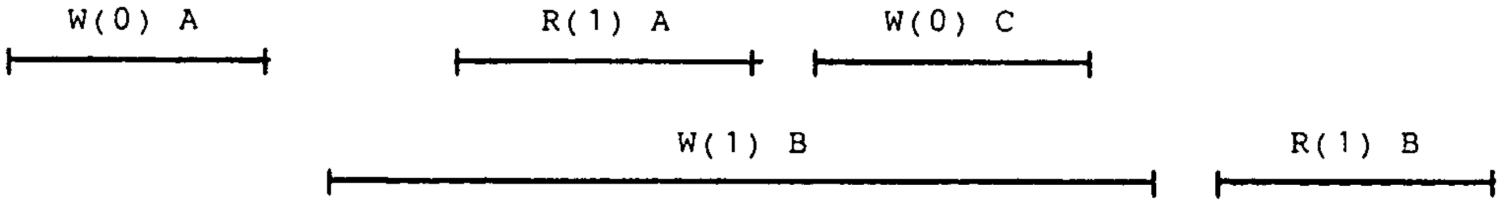


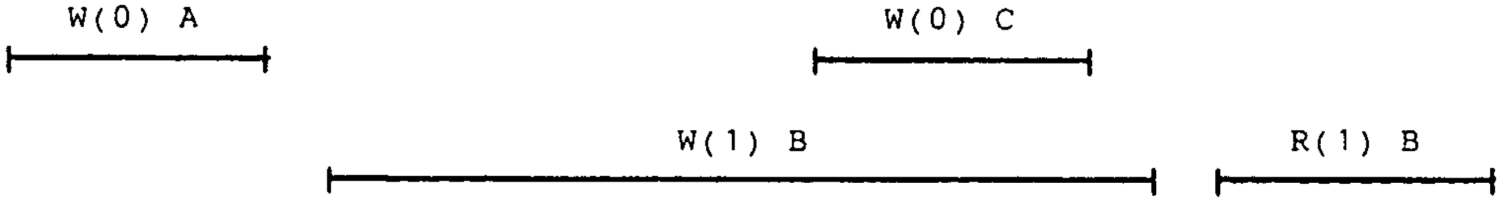


Linearizability

- Read-Write variable







Linearization Points

- Linearization points are points where method takes effect
- Useful to verify and design concurrent solutions

- For lock-based implementations, the critical section is its linearization point
- For implementations without any lock, linearization point is typically a single step where effects of method call become available to other method calls

Linearization Points

- For implementations without any lock, linearization point is typically a single step where effects of method call become available to other method calls
- enq(): if full, when exception gets thrown otherwise, when tail gets updated
- deq(): if empty, when exception gets thrown otherwise, when head is updated

```
public void enq(T x) throws FullException {
  if (tail - head == items.length)
    throw new FullException();
  items[tail % items.length] = x;
  tail++;
}
```

```
public T deq() throws EmptyException {
  if (tail - head == 0)
    throw new EmptyException();
  T x = items[head % items.length];
  head++;
  return x;
}
```

Linearizability

- Every linearizable execution is sequentially consistent, but not vice versa

- Linearizability is composable
  - Formally, H is linearizable if, and only if, for each object x, H|x is linearizable
  - where H|x means sub-history of H containing operations over x
  - Proof in text/paper

Serializability

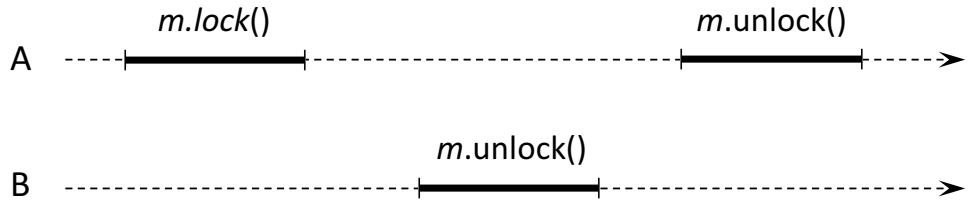
- Often considered in database transactions
- History is serializable if it is equivalent to a serial execution

- Similar to Linearizability, except doesn't capture real-time

- Data-centric perspective
  - Preserve semantics of transactions
  - Don't care about program order (or threads)

Serializabiltiy: https://en.wikipedia.org/wiki/Serializability

Serializability



- Intuitively incorrect
  - lock is held by both threads at the same time
- Serializable
  - B's m.unlock() can move before A's m.lock()
- Not linearizable – Why?
- Typically:
  - Linearizability useful for components like data structures
  - Serializability useful for complex components like transactions