Chapter 3

Concurrent Objects and Linearizability

3.1 Specifying Objects

An *object* in languages such as Java and C++ is a container for data. Each object provides a set of *methods* that are the only way to to manipulate that object's state. Each object has a *class* which describes how its methods behave. There are many ways to describe an object's behavior, ranging from formal specifications to plain English. The application programmer interface (API) documentation that we use every day lies somewhere between mathematical rigor and conversational chatter. This documentation typically says something like the following:

- If the object is in such-and-such a state before you call the method,
- then the object will be in some other state when the method returns,
- and the method itself will return a particular value or throw a particular an exception.

This kind of description divides naturally into a *precondition* (describing the object's state before invoking the method) and a *postcondition*, describing the object's state and return value after the method returns. For example, if the first-in-first-out (FIFO) queue object is non-empty (precondition), then the (`deq()` method will remove and return the first element (postcondition), and otherwise it will throw an exception (another pre- and postcondition).

This style of documentation is so familiar that it is easy to overlook how elegant and powerful it is. The size of the object's documentation is linear in the number of methods, because each method can be described in isolation. The

[0] *This chapter is part of the Manuscript* MULTIPROCESSOR SYNCHRONIZATION *by Maurice Herlihy and Nir Shavit copyright © 2003, all rights reserved.*

vast number of potential interactions among methods are captured implicitly through their side-effects on the object state. The object has a well-defined *state* between method calls (for example, a FIFO queue is a sequence of items). The object's documentation describes the object state before and after each call, and we can safely ignore any intermediate states the object may assume while the method call is in progress.

Defining objects in terms of preconditions and postconditions makes perfect sense in a *sequential* model computation where a single thread manipulates a collection of objects. For objects shared by multiple threads, this successful and familiar style of documentation falls apart. If an object's methods can be invoked by concurrent threads, then the method executions can overlap in time, and it no longer makes sense to talk about the order of method calls. What does it mean, in a multithreaded program, if x and y are enqueued on a FIFO queue during overlapping intervals? Which will be dequeued first? Can we continue to describe methods in isolation, via pre- and post-conditions, or must we provide explicit descriptions of every possible interaction among every possible collection of concurrent method calls?

Even the notion of an object's state becomes confusing. In single-thread programs, an object needs to assume a meaningful state only between method calls. For concurrent objects, however, overlapping method calls may be in progress at every instant, thus the object may never be "between" method calls. When implementing such a method, one must be prepared to encounter an object state that reflects the incomplete effects of concurrent method calls, a problem that just does not arise in single-threaded programs.

3.2 Linearizability

We propose the following way out of this dilemma, which we call *The Linearizability Manifesto.*

> Each method call should appear to "take effect" instantaneously at some moment between its invocation and response.

This manifesto is not a scientific statement, since it cannot be proved right or wrong. Nevertheless, since it was proposed in 1990, this property (called *linearizability*) has achieved widespread acceptance. There are certain specialized situations where other correctness properties are more appropriate, but for now they are the exception, not the rule.

One immediate advantage of linearizability is that there is no need to describe vast numbers of interactions among concurrent method calls, because we can still use the familiar and well-understood pre- and post-condition style of describing object methods. True, concurrency introduces additional uncertainty (what else did you expect?). If x and y are enqueued on an empty FIFO queue during overlapping intervals, then one method call will take effect before the other, so a later dequeue will return either x or y (either is correct), but it will not return some z, or throw an exception.

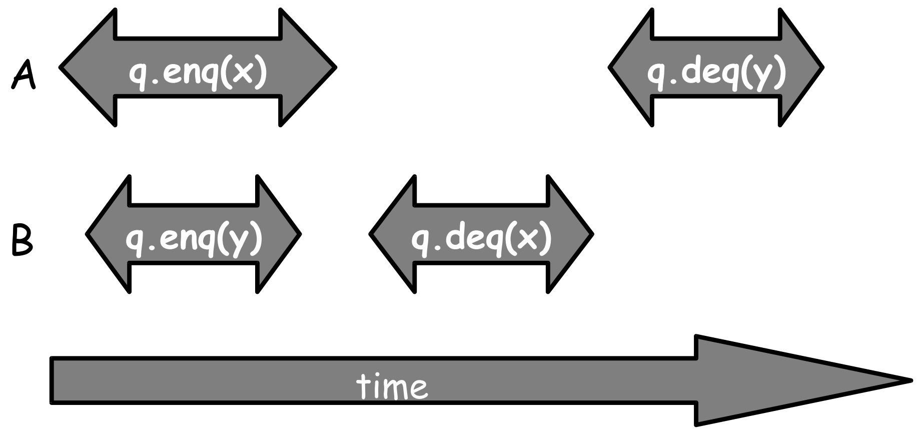


Figure 3.1: History H_1

Linearizability is a *local* property: a system is linearizable if and only if each individual object is linearizable. Locality enhances modularity and concurrency, since objects can be implemented and verified independently, and run-time scheduling can be completely decentralized.

Linearizability is also a *non-blocking* property: one method is never forced to wait to synchronize with another. Of course, a particular object implementation may force one method to wait for another, but such waiting is a product of the implementation, not forced by the correctness condition. (Methods may also wait for logical reasons unrelated to synchronization. For example, a dequeue applied to an empty queue may wait for another thread to enqueue something.)

Properties such as locality and being non-blocking may seem simple, but they should not be taken for granted. We will see that many other notions of concurrent correctness have been proposed that lack these important properties.

3.3 Examples

To reinforce out intution about linearizable executions, we will review a number of simple examples, mostly based on FIFO queues.

Method calls take time. An *operation* is the interval that starts with a method *invocation* event and ends with a method *response* event. Here is a key point. When we say that an operation "takes effect" instantaneously, we mean that we effectively reorder operations so that they occur sequentially, that is, methods are called, they execute, and they return, one at a time without overlapping. When we say that an operation *takes effect* between the method invocation and response, we meant that two operations that do not overlap must be ordered in that order (or, more formally, in an order consistent with their real-time precedence). Overlapping method calls can be ordered either way.

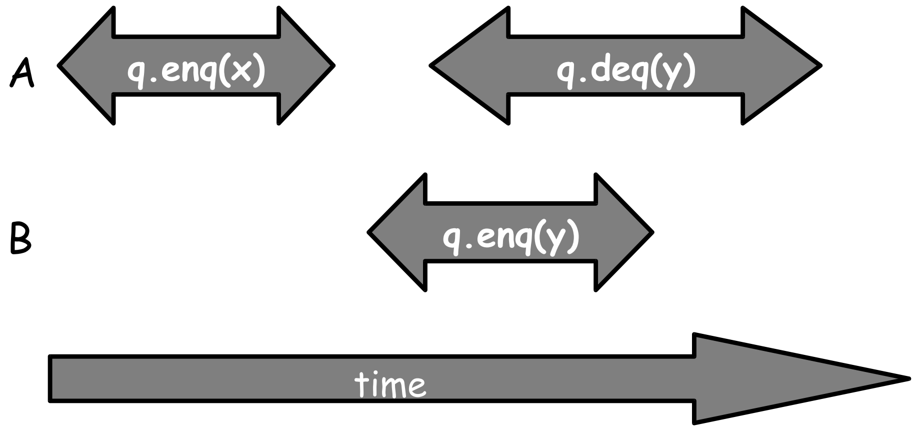


Figure 3.2: History H_2

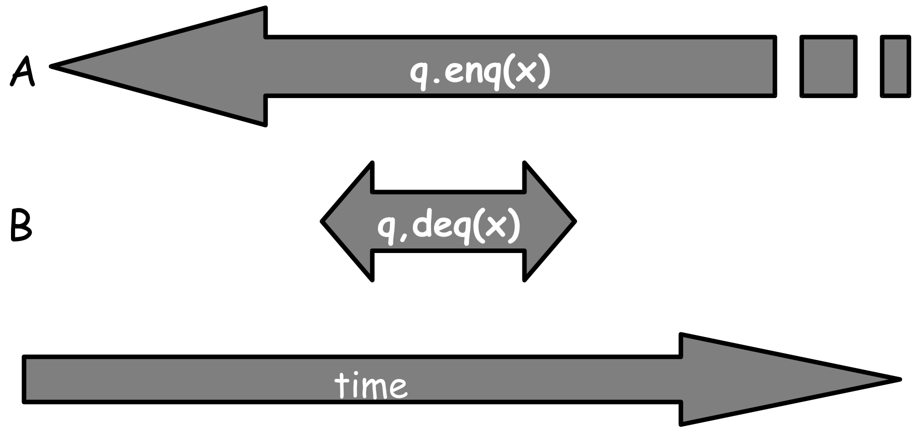


Figure 3.3: History H_3

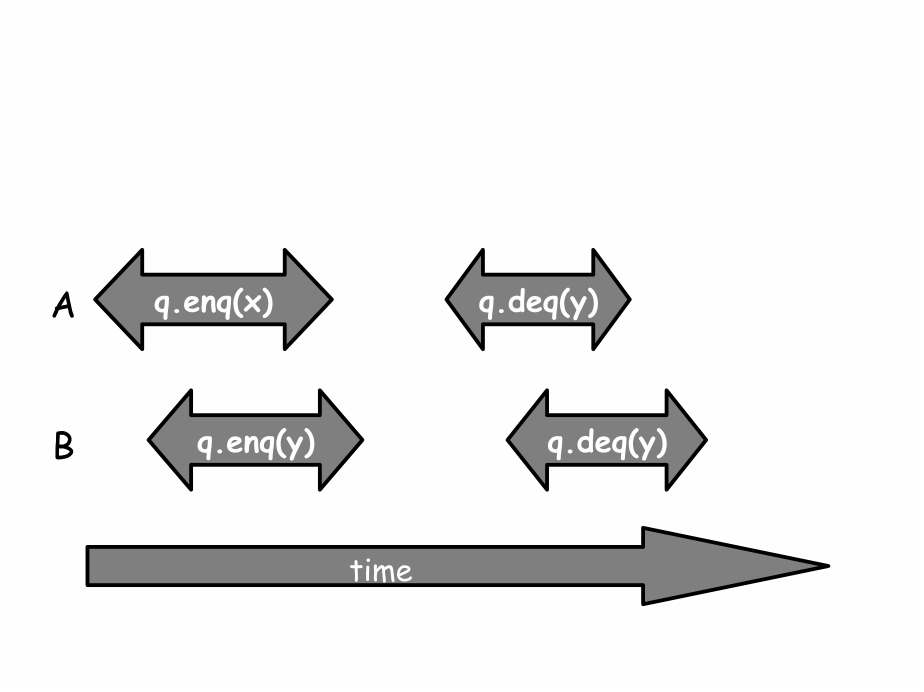


Figure 3.4: History H_4

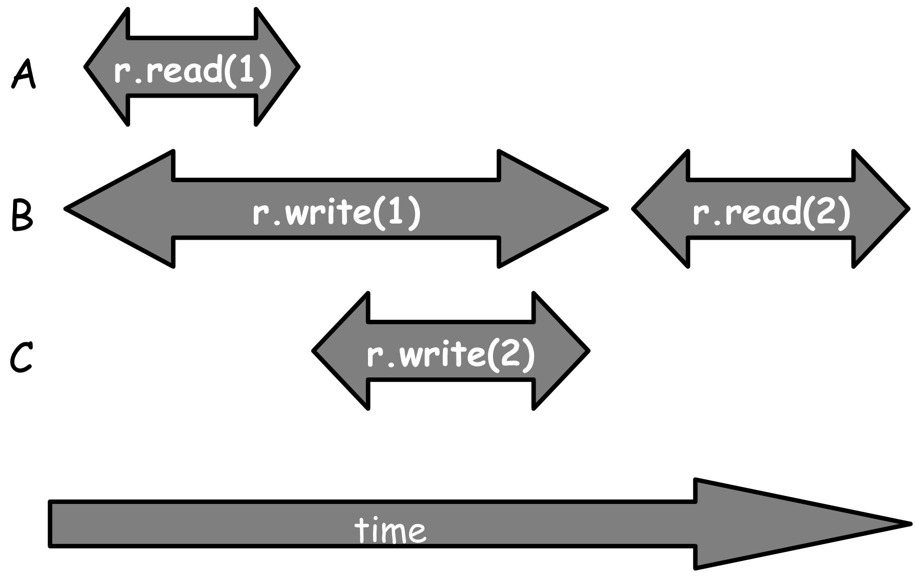
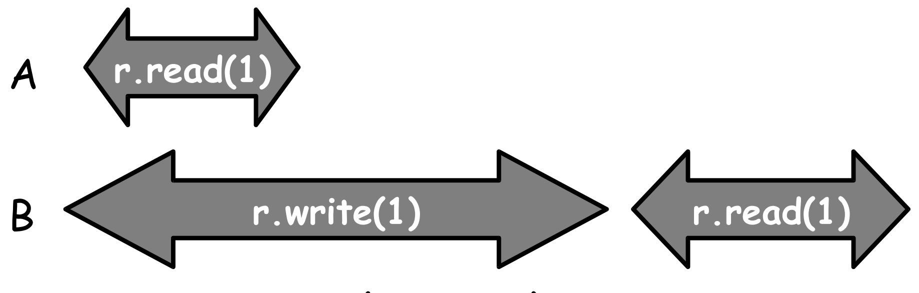


Figure 3.5: History H_5



Let us examine some ways in which a concurrent FIFO queue might behave. Our figures are based on the following conventions. Time moves from left to right, and arrows indicate intervals. Each thread's intervals are displayed along a horizontal line. The thread name appears on the left. A double-headed arrow represents an interval with a fixed start time and stop time. A single-headed arrow represents an interval with a fixed start time and an unknown stop time. The label $q.enq(a)A$ means that thread A enqueues item a at object q, while $q.deq(a)A$ means that thread A dequeues item a from object q.

In H_1 (Figure 3.1), thread A and B concurrently enqueue items x and y. Later, B dequeues x, and then A dequeues y. Since the dequeue interval for x precedes the dequeue inteval for y, their enqueues must have taken effect in that order. In fact, their enqueues were concurrent, thus they could indeed have taken effect in that order.

By contrast, the behavior shown in H_2 (Figure 3.2) is not linearizable. Here, x must have been enqueued before y (their intervals do not overlap), yet y is dequeued without x having been dequeued. To be linerizable, A should have dequeued x.

H_3 (Figure 3.3) is linerarizable, even though x is dequeued before its enqueuing inteval is complete. Intuitively, the enqueue of x "took effect" before it completed. H_4 (Figure 3.4) is not linearizable because y is dequeued twice.

Linearizable behavior can also be defined for other concurrent data structures such as read/write registers, stacks, sets, directories, and so on. Figures 3.5 and 3.6 show histories H_5 and H_6 for a `Register` object. In both, threads B and C write concurrently, but A's read (of 1) implies that B's write (of 1) takes effect before C's write (of 0). H_5 is linearizable because B's final read (of 1) is consistent with this order, while H_6 is not, because B's final read (of 0) is inconsistent.

3.4 Queue Implementations

Figure 3.7 shows a standard Java implementation of a shared FIFO queue. Figure **??** shows an execution in which A enqueues a, B enqueues b, and C dequeues b. Overlapping intervals indicate concurrent operations. All three method calls overlap in time.

The time arrow shows which thread holds the lock. (For simplicity, we consider lock entry and exit intervals to be instantaneous.) Here, C acquires the lock, observes the queue to be empty, and releases the lock. B acquires the lock, inserts b, and releases the lock. A acquires the lock, inserts a, and releases the lock. C reacquies the lock, dequeues b, releases the lock and returns. With so much operation overlapping and acquiring, releasing, and reacquiring of locks, how can we tell whether this object is linearizable?

In fact, this particular execution is indeed linearizable because the actual updates to the object occur sequentially, in distinct critical sections, even though the method calls overlap. Thread C's opeation takes effect the second time it acquires the lock, while the other threads' operations take effect the first (and

```
class Queue {
  int head = 0;                 // next item to dequeue
  int tail = 0;                 // next empty slot
  Item[QSIZE] items;

  public synchronized void enq(Item x) {
    while (this.tail - this.head == QSIZE) {
      try {
        this.wait();                       // wait until not full
      } catch (InterruptedException e) {}; // ignore exceptions
    }
    this.items[this.tail++ % QSIZE] = x;
    this.notifyAll();
  }

  public synchronized Item deq() {
    while (this.tail - this.head == 0) {
      try {
        this.wait();                       // wait until non-empty
      } catch (InterruptedException e) {}; // ignore exceptions
    }
    Item x = this.items[this.head++];
    this.notifyAll();
    return x;
  }
}
```

Figure 3.7: Lock-based FIFO Queue

```
class Queue {

  int head = 0;                // next item to dequeue
  int tail = 0;                // next empty slot
  Item[QSIZE] items;

  public void enq(Item x) {
    while (this.tail - this.head == QSIZE); // busy-wait
    this.items[this.tail % QSIZE] = x;
    this.tail++;
  }

  public Item deq() {
    while (this.tail == this.head);   // busy-wait
    Item x = this.items[this.head % QSIZE];
    this.head++;
    return x;
  }
}
```

Figure 3.8: FIFO Queue without Mutual Exclusion

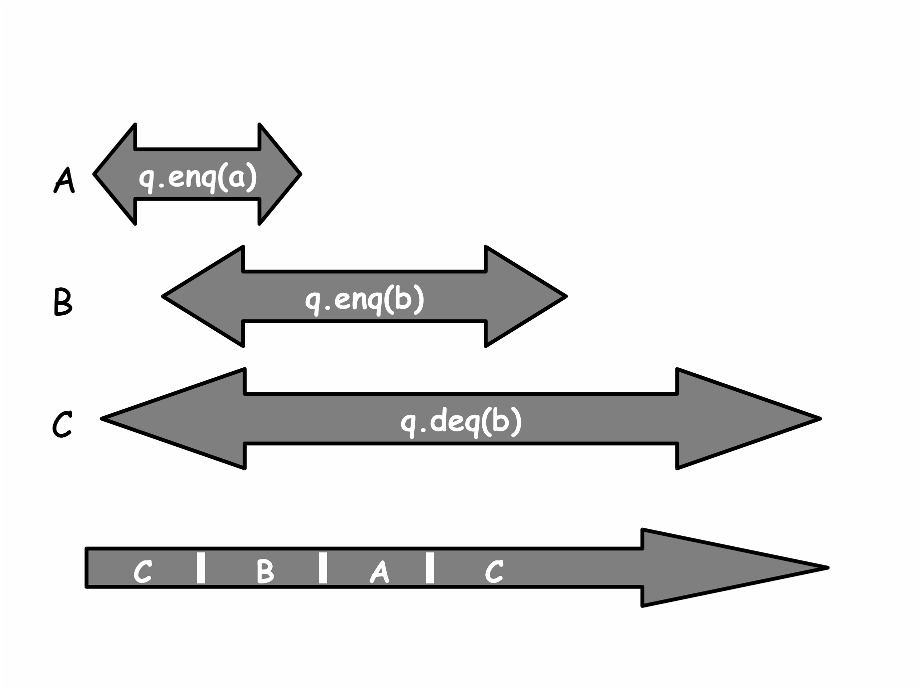


Figure 3.9: Locking queue execution

only) time they acquire the lock. This example suggests one approach: can we identify an instant where each operation takes effect?

Figure 3.8 shows an alternative queue implementation, intended to be shared by two threads only, that does not use mutual exclusion. This implementation is also linearizable. We will not give a formal proof, but if you think about it, you may see that each `enq()` operation "takes effect" when the `tail` field is updated, and each (`deq()` operation "takes effect" when the `head` field is updated. Here, instead of identifying the coarse-grained critical section in which the operation takes effect, we identify the fine-grained field update where it takes effect.

This approach of identifying the atomic step where an operation "takes effect" is the most common way to show that an implementation is linearizable. There are special cases where this approach does not work, but they are rare, and you are unlikely to encounter them.

3.5 Precise Definitions

Informally, we know that a concurrent object is linearizable if each operation appears to "take effect" instantaneously at some moment between that method's invocation and return events. This statement is probably enough for most informal reasoning, but a more precise formulation is needed to take care of some tricky cases (like operations that haven't returned), and for more rigorous styles of argument.

An execution of a concurrent system is modeled by a *history*, which is a finite sequence of method *invocation* and *response events*. A *subhistory* of a history H is a subsequence of the events of H.

A method invocation is written as $< x.m(args^*)A >$, where x is an object, m a method name, $args^*$ a sequence of arguments values, and A a thread. The invocation's response is written as $< x : term(res^*)A >$ where $term$ is a termination condition and res^* is a sequence of results. We use "Ok" for normal termination.

Definition A response *matches* an invocation if their objects names agree and their thread names agree.

Definition An invocation is *pending* in a history if no matching response follows the invocation.

Definition If H is a history, *complete*(H) is the maximal subsequence of H consisting only of invocations and matching responses.

Definition A history H is *sequential* if (1) The first event of H is an invocation. (2) Each invocation, except possibly the last, is immediately followed by a matching response. A history that is not sequential is *concurrent.*

Definition A *thread subhistory*, $H|P$ (H at P), of a history H is the subsequence of all events in H whose thread names are P. (An *object subhistory* $H|x$

is similarly defined for an object x.)

Definition Two histories H and H' are *equivalent* if for every thread A, $H|A = H'|A$.

Definition A history H is *well-formed* if each thread subhistory $H|A$ of H is sequential.

All histories considered here are well-formed. Notice that thread subhistories of a well-formed history are always sequential, but object subhistories need not be.

An *operation* is a pair consisting of an invocation $x.m(a)A$, and the next matching response $x : t(r)A$. A set S of histories is *prefix-closed* if whenever H is in S, every prefix of H is also in S.

A *sequential specification* for an object is a prefix-closed set of sequential histories for the object. A sequential history H is *legal* if each object subhistory $H \mid x$ belongs to the sequential specification for x.

An method is *total* if it is defined for every object state, otherwise it is *partial.* For example, consider the following specification for an unbounded sequential FIFO queue. one can always enqueue another item, but one can dequeue only from a non-empty queue. In this specification, method `enq()` is total, since its effects are defined in every queue state. Method (`deq()`, however, is partial, since its effects are defined only for non-empty queues.

A history H induces an irreflexive partial order $\longrightarrow_H$ on methods: $e_1 \longrightarrow_H e_2$ if $res(e_1)$ precedes $inv(e_2)$ in H. If H is sequential, then $\longrightarrow_H$ is a total order.

3.5.1 Linearizability

Definition A history H is *linearizable* if it can be extended (by appending zero or more response events) to a history H' such that:

L1 : *complete*(H') is equivalent to some legal sequential history S, and

L2 : $\longrightarrow_H \;\subseteq\; \longrightarrow_S$.

Informally, extending H to H' captures the idea that some pending invocations may have taken effect even though their responses have not yet been returned to the caller. Extending H to H' while restricting attention to complete(H') makes it possible to complete pending methods, or just to ignore them. *L2* states that this apparent sequential interleaving respects the precedence ordering of methods.

3.6 Properties of Linearizability

Theorem 3.6.1 *H is linearizable and only if for each object x, $H|x$ is linearizable.*

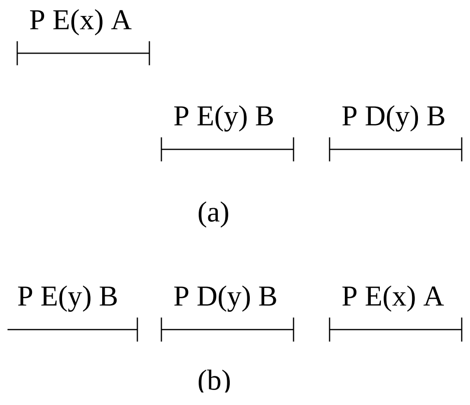


Figure 3.10: History H_7 and its equivalent sequential history.

This property of linearizability is called *locality*. *Locality* is important because it allows concurrent systems to be designed and constructed in a modular fashion; linearizable objects can be implemented, verified, and executed independently. Locality should not be taken for granted; as the following discussion shows, the literature includes proposals for alternative correctness properties that are not local.

Definition [Lamport] A history is *sequentially consistent* if and only if it is equivalent to a legal sequential history.

This condition is weaker than linearizability since it only requires condition *L1* of the definition of the linearizability. Figure **??** shows a sequentailly consistent FIFO queue history.

The history H_8 for objects p and q, shown in Figure **??**, shows that sequential consistency is not a local property. $H_8|p$ is sequentially consistent (reorder $op1$ and $op2$). $H_8|q$ is also sequentially consisten (reorder $op3$ and $op4$). H_8 itself is not sequentially consistent, because there is no total order consistent with both subhistories.

3.6.1 Comparison to Other Correctness Conditions

Much work on databases and distributed systems uses *serializability* as the basic correctness condition for concurrent computations. In this model, a *transaction* is a "thread of control" that applies a finite sequence of methods to a set of objects shared with other transactions.

Definition A history is *serializable* if it is equivalent to one in which transactions appear to execute sequentially, that is, without interleaving.

Definition A history is *strictly serializable* if the transactions' order in the sequential history is compatible with their precedence order: if every method

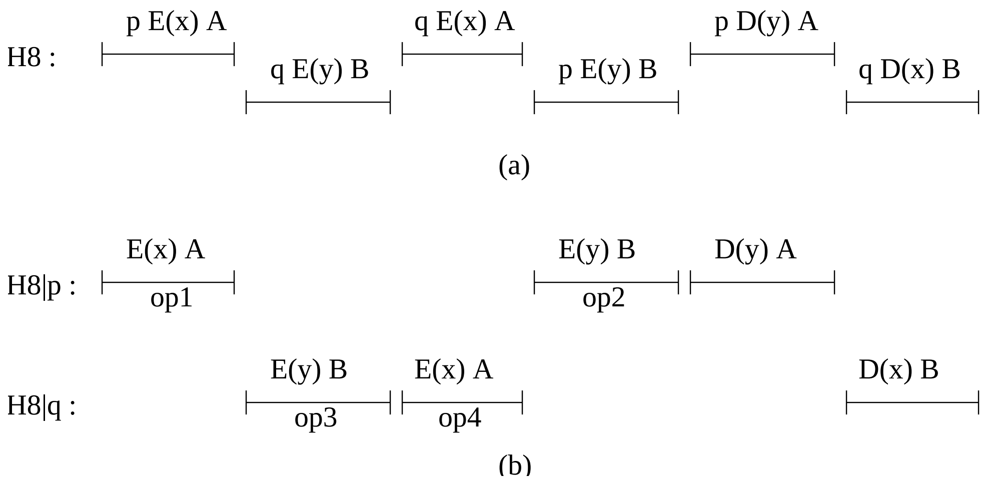


Figure 3.11: History H_8.

call of one transaction preceeds every method call of another, the former is serialized first.

Linearizability can be viewed as a special case of strict serializability where transactions are restricted to consist of a single method applied to a single object. Nevertheless, this single-operation restriction has far-reaching consequences. Neither serializability nor strict serializability is a local property. For example, in history H_8 shown above, if we interpret A and B as transactions instead of threads, then it is easily seen that both $H_p|A$ and $H_8|q$ are strictly serializable but H_8 is not. (Because A and B overlap at each object, they are unrelated by transaction precedence in either subhistory.) Moreover, since A and B each dequeues an item enqueued by the other, H_8 is not even serializable. A practical consequence of this observation is that implementors of objects in serializable systems must rely on convention to ensure that all objects' concurrency control mechanisms are compatible with one another.

Another important difference between linearizability and serializability is that both serializability and strict serializability are *blocking* properties, while linearizability is non-blocking. Under certain circumstances, a transaction may be unable to complete a pending method without violating serializability. Such a transaction must be rolled back and restarted, implying that additional mechanism must be provided for that purpose. For example, consider the history H_9, involving two register objects x and y, and two transactions A and B. A and B respectively read x and y and then attempt to write new values to y and x. It is easy to see that both pending invocations cannot be completed without violating serializability.

Perhaps the major distinction between serializability and linearizability is

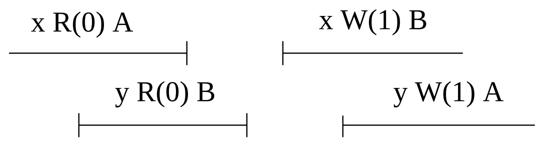


Figure 3.12: History H9

that the two notions are appropriate for different problem domains. Serializability is appropriate for database systems, in which it must be easy for application programmers to preserve complex application-specific invariants spanning multiple objects. Linearizability is appropriate for generalized individual shared object implementations.