COMP 322: Fundamentals of Parallel Programming

Lecture 26: Java Locks, Linearizability

Mack Joyner and Zoran Budimlić
{mjoyner, zoran}@rice.edu

http://comp322.rice.edu

Worksheet #25 solution: Bounded Buffer Example

Consider the case when multiple threads call insert() and remove() methods concurrently for a single BoundedBuffer instance with SIZE >= 1.

NOTE: the BoundedBuffer instance is the object used by the synchronized statements, not the objects being inserted/removed.

1) Can you provide an example in which the wait set includes a thread waiting at line 2 in insert() and a thread waiting at line 11 in remove(), in slide 12? If not, why not?

Yes, if notified threads in the wait set don't have higher priority over threads in the entry set

2) How would the code behave if all wait/notify calls (lines 2, 6, 11, 15) were removed from the insert() and remove() methods in slide 12?

insert() may overwrite existing elements when buffer is supposed to be full

remove() may return undefined values when buffer is supposed to be empty

Unit 7.3: Locks

- Use of monitor synchronization is just fine for most applications, but it has some shortcomings
  - Single wait-set per lock
  - No way to interrupt or time-out when waiting for a lock
  - Locking must be block-structured
    - Inconvenient to acquire a variable number of locks at once
    - Advanced techniques, such as hand-over-hand locking, are not possible
- Lock objects address these limitations
  - But harder to use: Need **`finally`** block to ensure release
  - So if you don't need them, stick with **`synchronized`**

Example of hand-over-hand locking:

- L1.lock() … L2.lock() … L1.unlock() … L3.lock() … L2.unlock() ….

java.util.concurrent.locks.Lock interface

```
1.  interface Lock {
2.    // key methods
3.    void lock(); // acquire lock
4.    void unlock(); // release lock
5.    boolean tryLock();
6.    // Either acquire lock and return true, or return false if lock is
7.    /// not obtained.  A call to tryLock() never blocks!
8.    Condition newCondition();  // associate a new condition
9.                               // variable with the lock
      }
```

- java.util.concurrent.locks.Lock interface is implemented by java.util.concurrent.locks.ReentrantLock class

Simple ReentrantLock() example

- Used extensively within `java.util.concurrent`

```
final Lock lock = new ReentrantLock();
...
lock.lock();
try {
  // perform operations protected by lock
}
catch(Exception ex) {
  // restore invariants & rethrow
}
finally {
  lock.unlock();
}
```

- **Must manually ensure lock is released**

==> Importance of including call to unlock() in finally clause!

java.util.concurrent.locks.condition interface

- Can be allocated by calling ReentrantLock.newCondition()
- Supports multiple condition variables per lock
- Methods supported by an instance of condition
  - void await() // NOTE: like wait() in synchronized statement
    - Causes current thread to wait until it is signaled or interrupted
    - Variants available with support for interruption and timeout
  - void signal() // NOTE: like notify() in synchronized statement
    - Wakes up one thread waiting on this condition
  - void signalAll() // NOTE: like notifyAll() in synchronized statement
    - Wakes up all threads waiting on this condition
- For additional details see
  - http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks/Condition.html

BoundedBuffer example using two conditions, notFull and notEmpty

```
1. class BoundedBuffer {
2.   final Lock lock = new ReentrantLock();
3.   final Condition notFull  = lock.newCondition();
4.   final Condition notEmpty = lock.newCondition();
5.
6.   final Object[] items = new Object[100];
7.   int putptr, takeptr, count;
8.
9.   . . .
```


BoundedBuffer example using two conditions, notFull and notEmpty (contd)

```
10.  public void put(Object x) throws InterruptedException
11.  {
12.    lock.lock();
13.    try {
14.      while (count == items.length) notFull.await();
15.      items[putptr] = x;
16.      if (++putptr == items.length) putptr = 0;
17.      ++count;
18.      notEmpty.signal();
19.    } finally {
20.      lock.unlock();
21.    }
22.  }
```


BoundedBuffer example using two conditions, notFull and notEmpty (contd)

```
23.  public Object take() throws InterruptedException
24.  {
25.    lock.lock();
26.    try {
27.      while (count == 0) notEmpty.await();
28.      Object x = items[takeptr];
29.      if (++takeptr == items.length) takeptr = 0;
30.      --count;
31.      notFull.signal();
32.      return x;
33.    } finally {
34.      lock.unlock();
35.    }
36.  }
```


Reading vs. writing

- Recall that the use of synchronization is to protect interfering accesses
  - Concurrent reads of same memory: Not a problem
  - Concurrent writes of same memory: Problem
  - Concurrent read & write of same memory: Problem

So far:

- If concurrent write/write or read/write might occur, use synchronization to ensure one-thread-at-a-time

But:

- This is unnecessarily conservative: we could still allow multiple simultaneous readers (as in object-based isolation)

Consider a hashtable with one coarse-grained lock

- Only one thread can perform operations at a time

But suppose:

- There are many simultaneous `lookup` operations and `insert` operations are rare

java.util.concurrent.locks.ReadWriteLock interface

```
interface ReadWriteLock {
  Lock readLock();
  Lock writeLock();
}
```

- Even though the interface appears to just define a pair of locks, the semantics of the pair of locks is coupled as follows
  - Case 1: a thread has successfully acquired writeLock().lock()
    - No other thread can acquire readLock() or writeLock()
  - Case 2: no thread has acquired writeLock().lock()
    - Multiple threads can acquire readLock()
    - No other thread can acquire writeLock()
- java.util.concurrent.locks.ReadWriteLock interface is implemented by java.util.concurrent.locks.ReadWriteReentrantLock class

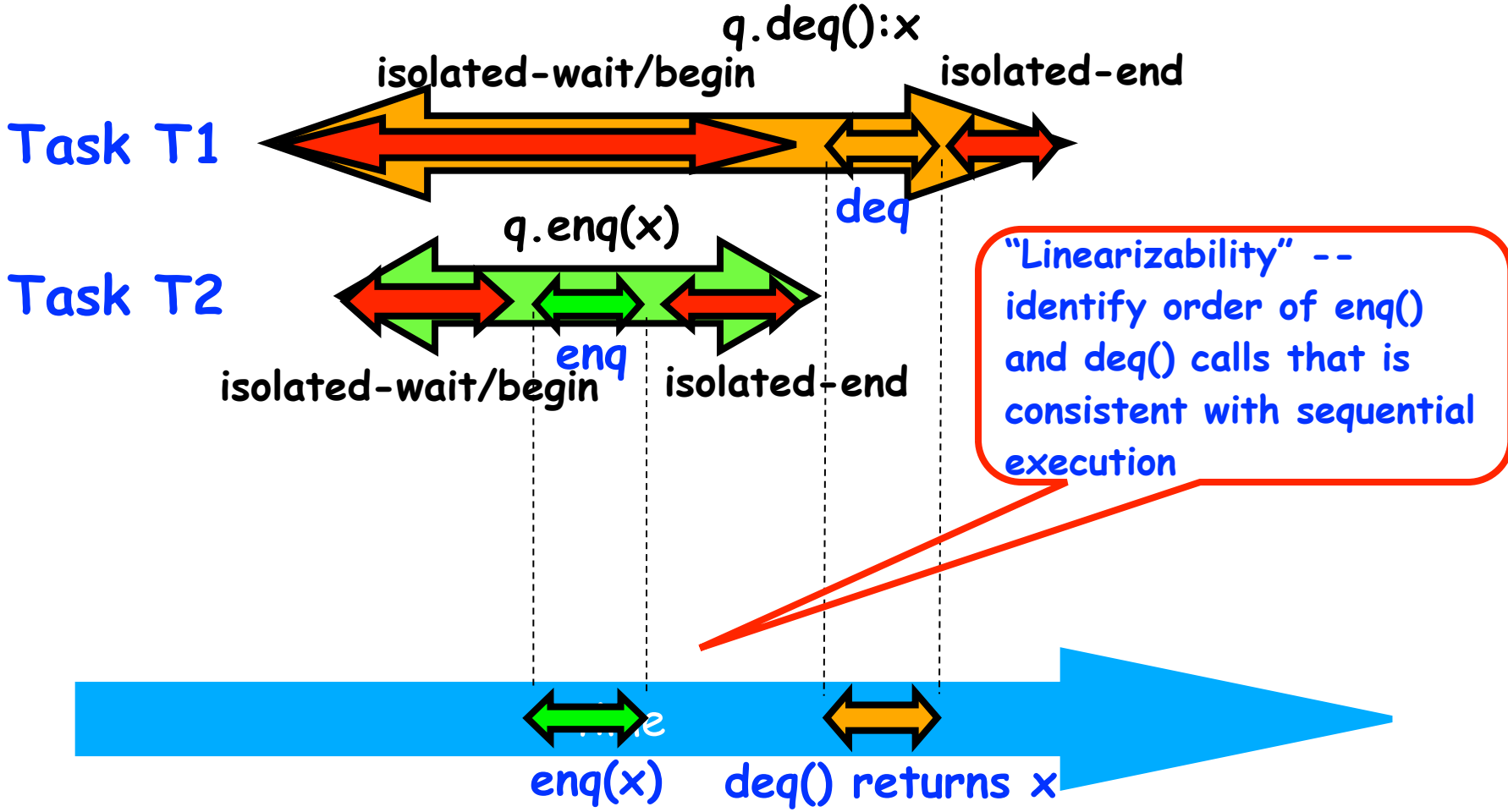
Example code

```
class Hashtable<K,V> {
  …
  // coarse-grained, one lock for table
  ReadWriteLock lk = new ReentrantReadWriteLock();
  V lookup(K key) {
    int bucket = hasher(key);
    lk.readLock().lock(); // only blocks writers
    … read array[bucket] …
    lk.readLock().unlock();
  }
  void insert(K key, V val) {
    int bucket = hasher(key);
    lk.writeLock().lock(); // blocks readers and writers
    … write array[bucket] …
    lk.writeLock().unlock();
  }
}
```


Unit 7.4: Linearizability, Correctness of Concurrent Objects

- A *concurrent object* is an *object* that can correctly handle *methods* invoked *concurrently* by different tasks or threads
  - e.g., AtomicInteger, ConcurrentHashMap, ConcurrentLinkedQueue, …
- For the discussion of linearizability, we will assume that the body of each method in a concurrent object is itself sequential
  - Assume that methods do not create threads or async tasks
- Consider a simple FIFO (First In, First Out) queue as a canonical example of a concurrent object
  - Method q.enq(o) inserts object o at the tail of the queue
    - Assume that there is unbounded space available for all enq() operations to succeed
  - Method q.deq() removes and returns the item at the head of the queue.
    - Throws EmptyException if the queue is empty.
- Without seeing the implementation of the FIFO queue, we can tell if an execution of calls to enq() and deq() is correct or not, in a sequential program
- *How can we tell if the execution is correct for a parallel program?*

Linearization: identifying a sequential order of concurrent method calls

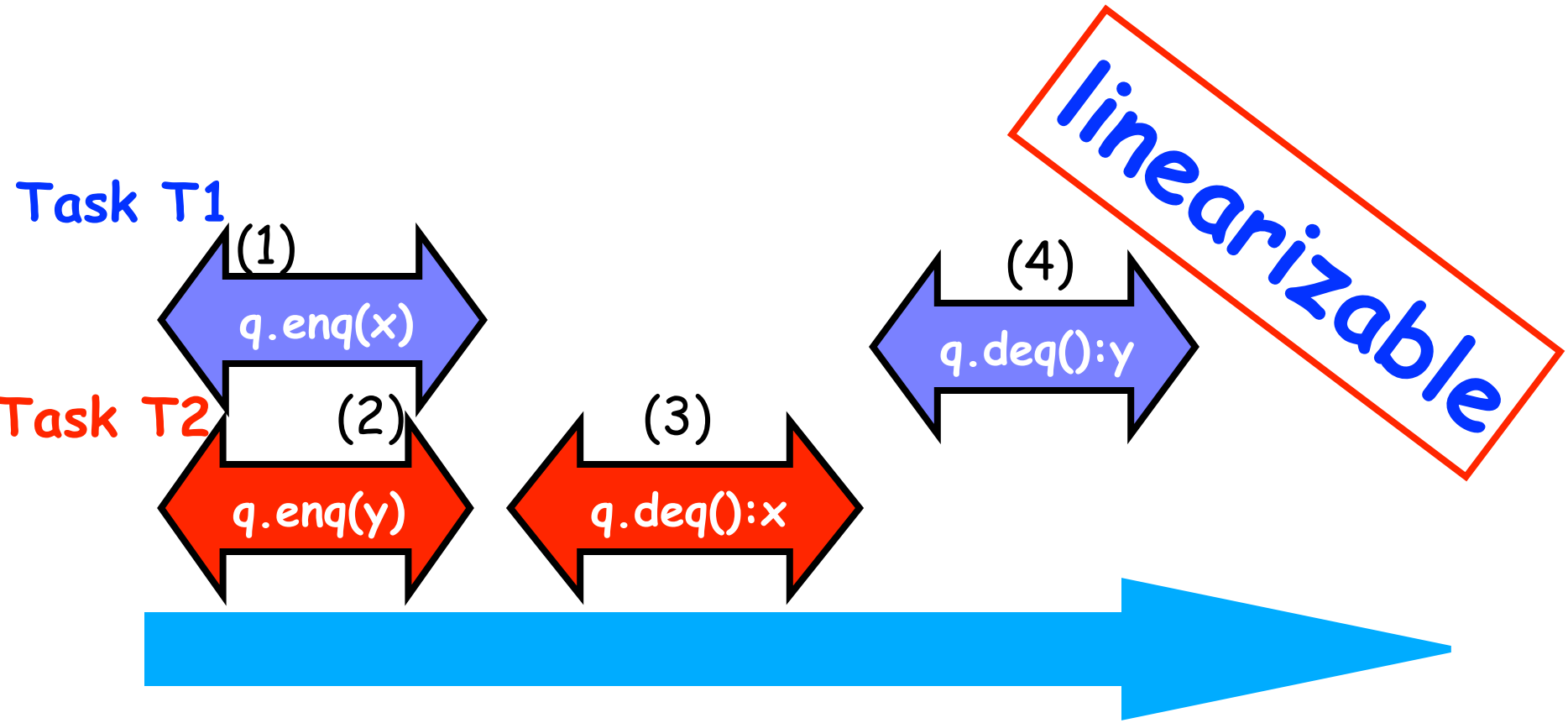


Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Informal definition of Linearizability

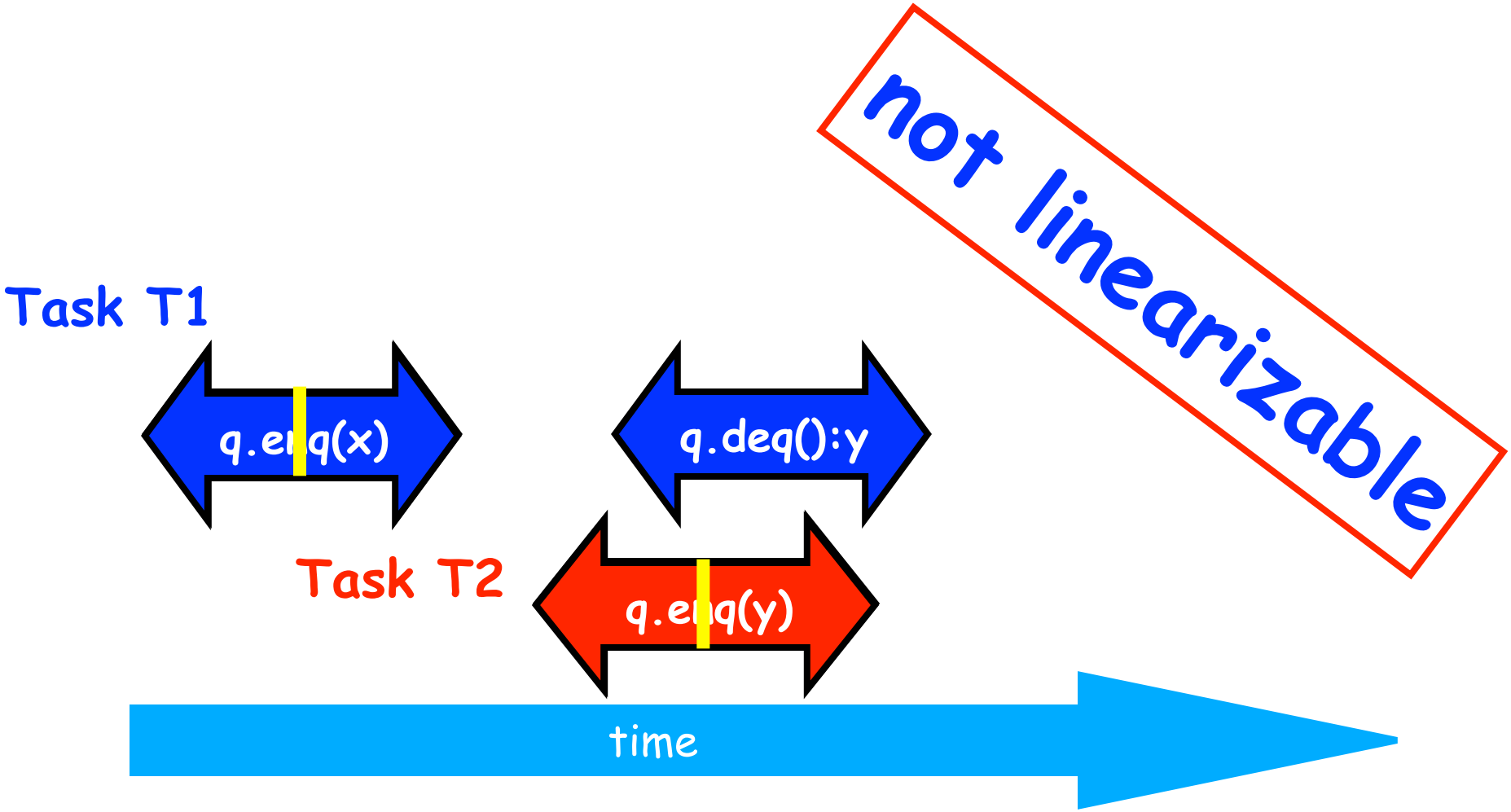
- Assume that each method call takes effect "instantaneously" at some point in time between its invocation and return.
- An *execution (schedule) is linearizable* if we can choose *one set of* instantaneous points that is consistent with a sequential execution in which methods are executed at those points
  - It's okay if some other set of instantaneous points is not linearizable
- A *concurrent object is linearizable* if all its executions are linearizable
  - Linearizability is a "black box" test based on the object's behavior, not its internals

Example 1: is this execution linearizable?



Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

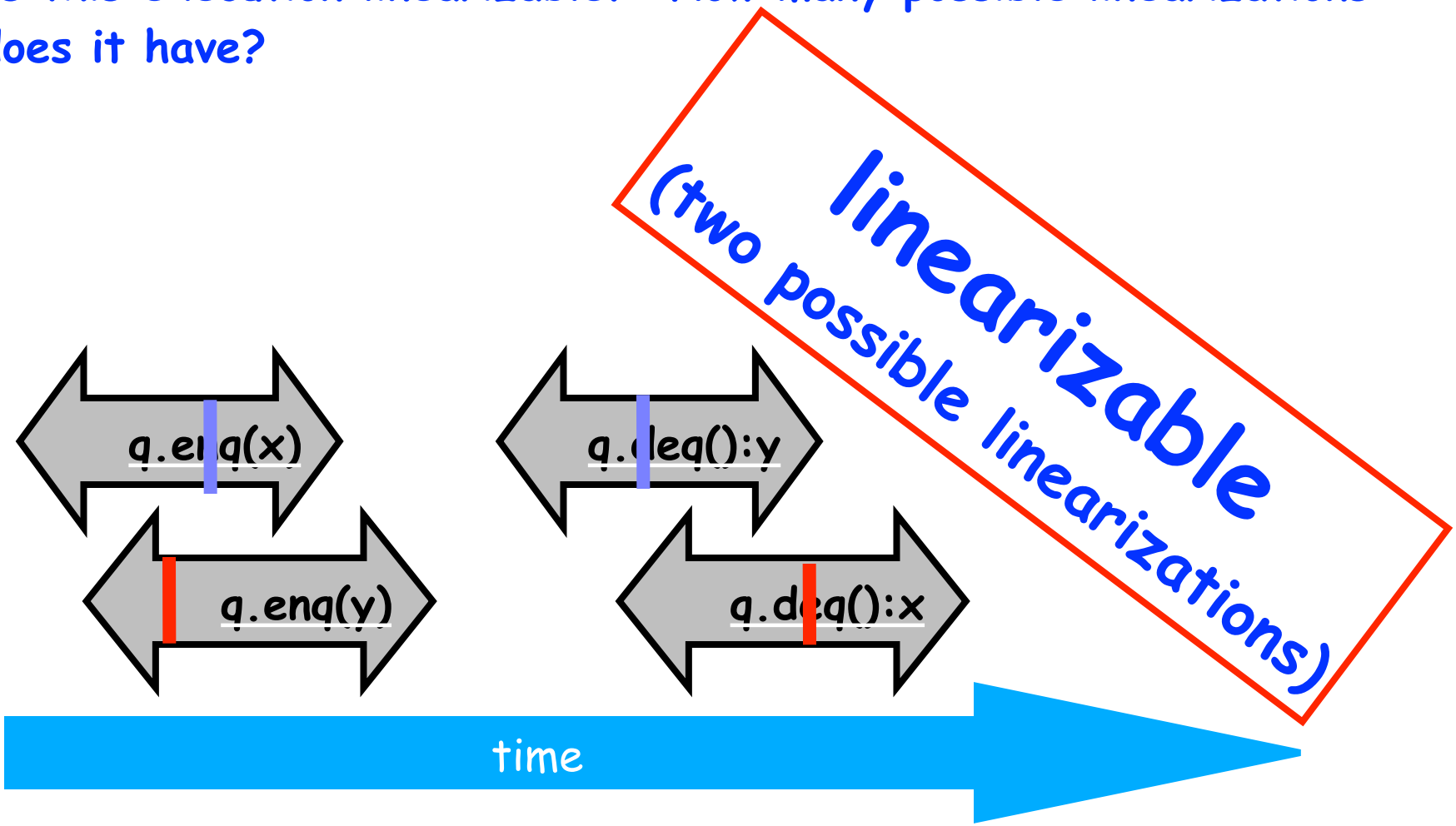
Example 2: is this execution linearizable?



Source: http://www.elsevierdirect.com/companions/9780123705914/Lecture%20Slides/03~Chapter_03.ppt

Example 3

Is this execution linearizable? How many possible linearizations does it have?



Example 4: execution of an isolated implementation of FIFO queue q

Is this a linearizable execution?

Time	Task A	Task B
0	Invoke `q.enq(x)`	
1	Work on `q.enq(x)`	
2	Work on `q.enq(x)`	
3	Return from `q.enq(x)`	
4		Invoke `q.enq(y)`
5		Work on `q.enq(y)`
6		Work on `q.enq(y)`
7		Return from `q.enq(y)`
8		Invoke `q.deq()`
9		Return `x` from `q.deq()`

Yes! Can be linearized as "q.enq(x) ; q.enq(y) ; q.deq():x".

Linearizability of Concurrent Objects (Summary)

Concurrent object

- A concurrent object is an object that can correctly handle methods invoked in parallel by different tasks or threads
  - Examples: Concurrent Queue, AtomicInteger

Linearizability

- Assume that each method call takes effect "instantaneously" at some distinct point in time between its invocation and return.
- An execution is linearizable if we can choose instantaneous points that are consistent with a sequential execution in which methods are executed at those points
- An object is linearizable if all its possible executions are linearizable