Solution to Worksheet #25:
Linearizability of method calls on a concurrent object

Is this a linearizable execution for a FIFO queue, q?

<table>
<thead>
<tr>
<th>Time</th>
<th>Task A</th>
<th>Task B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Invoke q.enq(x)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Return from q.enq(x)</td>
<td>Invoke q.enq(y)</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Work on q.enq(y)</td>
</tr>
<tr>
<td>3</td>
<td>Invoke q.deq()</td>
<td>Return from q.enq(y)</td>
</tr>
<tr>
<td>4</td>
<td>Work on q.deq()</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Return y from q.deq()</td>
<td></td>
</tr>
</tbody>
</table>

No! q.enq(x) must precede q.enq(y) in all linear sequences of method calls invoked on q. It is illegal for the q.deq() operation to return y.
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 (schedule) 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

Why is Linearizability important?

- Linearizability is a correctness condition for concurrent objects
- For example, is the following implementation of AtomicInteger.getAndIncrement() linearizable?
  - Motivation: many processors provide hardware support for get() and compareAndSet(), but not for getAndAdd()

1. public final int getAndIncrement() {
2.     int current = get();
3.     int next = current + 1;
4.     compareAndSet(current, next);
5.     return current;
6. }

4. COMP 322, Spring 2016 (V. Sarkar, S. Imam)
A Linearizable Implementation of getAndIncrement() using compareAndSet()

1. public final int getAndIncrement() {
2.     while (true) {
3.         int current = get();
4.         int next = current + 1;
5.         if (compareAndSet(current, next))
6.             // success!
7.             return current;
8.         }
9.     }

getAndInc():0 must occur before getAndInc():1 for linearizability

c&S = false
c&S = true
return

time
getAndInc():0 getAndInc():1

Locks and Conditions in java.util.concurrent library

- Atomic variables
  - Key primitives for writing lock-free algorithms
  - Can be used from HJlib programs without any restrictions

- Concurrent Collections
  - Queues, blocking queues, concurrent hash map, ...
  - Only nonblocking methods can safely be used from HJlib

- Locks and Conditions (focus of today’s lecture)
  - More flexible synchronization control
  - Read/write locks

- Executors, Thread pools and Futures
  - Execution frameworks for asynchronous tasking
  - Low-level APIs used to implement HJlib and Java ForkJoin framework

- Synchronizers: Semaphore, Latch, Barrier, Exchanger
  - Ready made tools for thread coordination
  - Low-level APIs used to implement HJlib and Java ForkJoin framework
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

```java
interface Lock {
    // key methods
    void lock(); // acquire lock
    void unlock(); // release lock
    boolean tryLock(); // return false if lock is not obtained
    boolean tryLock(long timeout, TimeUnit unit)
        throws InterruptedException
    Condition newCondition(); // associate a new condition
        // 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`
  ```java
  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

**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](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.   final Object[] items = new Object[100];
6.   int putptr, takeptr, count;
7.   . . .

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

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