COMP 322: Fundamentals of Parallel Programming

Lecture 26: Linearizability (contd), Java locks

Vivek Sarkar, Shams Imam
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu, shams.imam@twosigma.com

http://comp322.rice.edu/

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

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

Time	Task *A*	Task *B*
0	Invoke q.enq(x)	
1	Return from q.enq(x)	
2		Invoke q.enq(y)
3	Invoke q.deq()	Work on q.enq(y)
4	Work on q.deq()	Return from q.enq(y)
5	Return y from q.deq()	

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

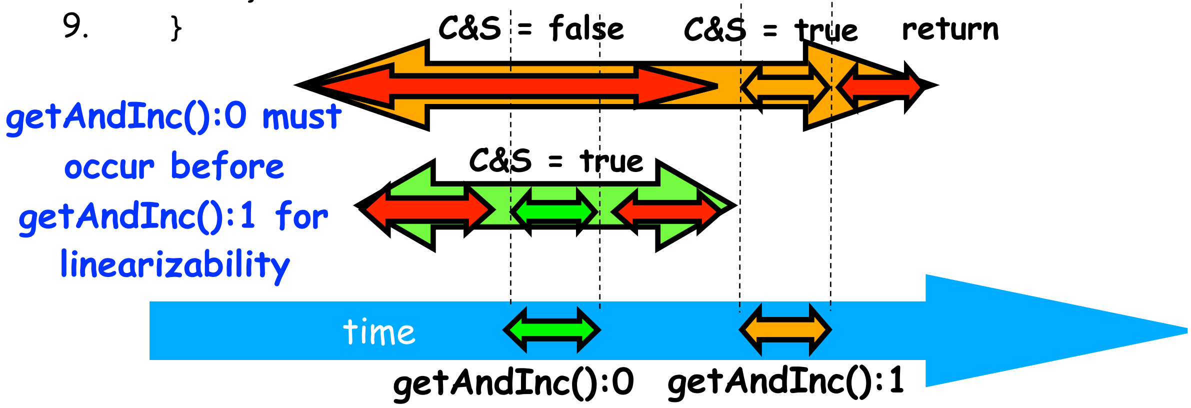
Why is Linearizability important?

- 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()

```
public final int getAndIncrement() {
        int current = get();
        int next = current + 1;
        compareAndSet(current, next);
        return current;
    }
```

A Linearizable Implementation of getAndIncrement() using compareAndSet()

```
public final int getAndIncrement() {
     while (true) {
          int current = get();
          int next = current + 1;
          if (compareAndSet(current, next))
              // success!
              return current;
       }
   }
```



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

```
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`

```
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

BoundedBuffer example using two conditions, notFull and notEmpty

```
class BoundedBuffer {
  final Lock lock = new ReentrantLock();
  final Condition notFull  = lock.newCondition();
  final Condition notEmpty = lock.newCondition();

  final Object[] items = new Object[100];
  int putptr, takeptr, count;

. . .
```


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

```
  public void put(Object x) throws InterruptedException
  {
    lock.lock();
    try {
      while (count == items.length) notFull.await();
      items[putptr] = x;
      if (++putptr == items.length) putptr = 0;
      ++count;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
```

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

```
public Object take() throws InterruptedException
{
  lock.lock();
  try {
    while (count == 0) notEmpty.await();
    Object x = items[takeptr];
    if (++takeptr == items.length) takeptr = 0;
    --count;
    notFull.signal();
    return x;
  } finally {
    lock.unlock();
  }
}
```


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();
  }
}
```