COMP 322: Fundamentals of Parallel Programming

Lecture 27: Java Locks

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

Worksheet #25 Solution: Bounded Buffer

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 8? 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 8?

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

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

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(); // Either acquire lock (returns true), or return false if lock is not obtained.
6.                                 // A call to tryLock() never blocks!
7.
8.     Condition newCondition();  // associate a new condition
9. }
```

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)

```
1.  public void put(Object x) throws InterruptedException
2.   {
3.     lock.lock();
4.     try {
5.       while (count == items.length) notFull.await();
6.       items[putptr] = x;
7.       if (++putptr == items.length) putptr = 0;
8.       ++count;
9.       notEmpty.signal();
10.     } finally {
11.       lock.unlock();
12.     }
13.   }
```


BoundedBuffer Example using Two Conditions: notFull and notEmpty (contd)

```
1.  public Object take() throws InterruptedException
2.  {
3.    lock.lock();
4.    try {
5.      while (count == 0) notEmpty.await();
6.      Object x = items[takeptr];
7.      if (++takeptr == items.length) takeptr = 0;
8.      --count;
9.      notFull.signal();
10.       return x;
11.   } finally {
12.     lock.unlock();
13.   }
14. }
```


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

Hashtable Example

```
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();
  }
}
```


Announcements & Reminders

- The entire written + programming (Checkpoint #3) is due by Friday, April 3rd at 11:59pm
- Quiz for Unit 6 is available, due Monday, April 6th at 11:59pm

Worksheet #27: Use of trylock()

Rewrite the transferFunds() method below to use j.u.c. locks with calls to tryLock (see slide 4) instead of synchronized.

Your goal is to write a correct implementation that never deadlocks, unlike the buggy version below (which can deadlock).

Assume that each Account object already contains a reference to a ReentrantLock object dedicated to that object e.g., from.lock() returns the lock for the from object. Sketch your answer using pseudocode.

```
1.  public void transferFunds(Account from, Account to, int amount) {
2.        synchronized (from) {
3.            synchronized (to) {
4.                    from.subtractFromBalance(amount);
5.                    to.addToBalance(amount);
6.            }
7.        }
8.  }
```