COMP 322: Fundamentals of Parallel Programming

Lecture 26: Java Locks

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu

Worksheet #25: Bounded Buffer

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

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?

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?

What if you want to wait for shared state to satisfy a desired property? (Bounded Buffer Example)

```
public synchronized void insert(Object item) { // producer
    // TODO: wait till count < BUFFER SIZE
    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
    // TODO: notify consumers that an insert has been performed
}

public synchronized Object remove() { // consumer
    Object item;
    // TODO: wait till count > 0
    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    // TODO: notify producers that a remove() has been performed
    return item;
}
```

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(); // Either acquire lock (returns true), or return false if lock is not obtained.
                       // A call to tryLock() never blocks!

    Condition newCondition();  // associate a new condition
}
```

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: full and empty

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

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

   . . .
```

BoundedBuffer Example using Two Conditions: full and empty (contd)

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

BoundedBuffer Example using Two Conditions: full and empty (contd)

```
public Object take() throws InterruptedException
{
  lock.lock();
  try {
    while (count == 0) empty.await();
    Object x = items[takeptr];
    if (++takeptr == items.length) takeptr = 0;
    --count;
    full.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

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

- Hw 3 - entire written + programming (Checkpoint #2) is due today at 11:59pm
- Lab 6 is due tomorrow at 12pm (noon)
- No lab this week
- Quiz for Unit 6 is due Monday, April 12th at 11:59pm
- Hw 4 will be available today
  - —Checkpoint #1 is due Monday, April 19th at 11:59pm
  - —Entire written + programming (Checkpoint #2) is due Wednesday, April 28th at 11:59pm

Worksheet #26: Use of trylock()

Rewrite the transferFunds() method below to use j.u.c. locks with calls to tryLock (see slide 5) 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.

```
public void transferFunds(Account from, Account to, int amount) {
        synchronized (from) {
            synchronized (to) {
                    from.subtractFromBalance(amount);
                    to.addToBalance(amount);
            }
        }
}
```