COMP 322: Fundamentals of Parallel Programming

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Lecture 30: Advanced locking in Java

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

Acknowledgments for Today’s Lecture

- Combined handout for Lectures 27-30 (to be updated)
- “Introduction to Concurrent Programming in Java”, Joe Bowbeer, David Holmes, OOPSLA 2007 tutorial slides
  - Contributing authors: Doug Lea, Brian Goetz
- “Java Concurrency Utilities in Practice”, Joe Bowbeer, David Holmes, OOPSLA 2007 tutorial slides
  - Contributing authors: Doug Lea, Tim Peierls, Brian Goetz
- ECE 3005 course slides from Georgia Tech
  - http://users.ece.gatech.edu/~copeland/jac/3055-05/ppt/ch07-sync-b.ppt
- A Sophomoric Introduction to Shared-Memory Parallelism and Concurrency, Lecture 6, Dan Grossman, U. Washington
  - http://www.cs.washington.edu/homes/djg/teachingMaterials/grossmanSPAC_lec6.pptx

Announcements

- Homework 6 deadline extended to 5pm on Wednesday, April 6th due to difficulties in accessing SUG@R nodes
  - Please use special COMP322 queue for SUG@R during lab hours

Complete Bounded Buffer using Java Synchronization (Recap)

```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER SIZE = 5;
    private int count, in, out;
    private Object[] buffer;
    public BoundedBuffer() { // buffer is initially empty
                count = 0;
                in = 0;
                out = 0;
                buffer = new Object[BUFFER SIZE];
    }
    public synchronized void insert(Object item) { // See previous slides
    }
    public synchronized Object remove() { // See previous slides
    }
}
```

insert() with wait/notify Methods

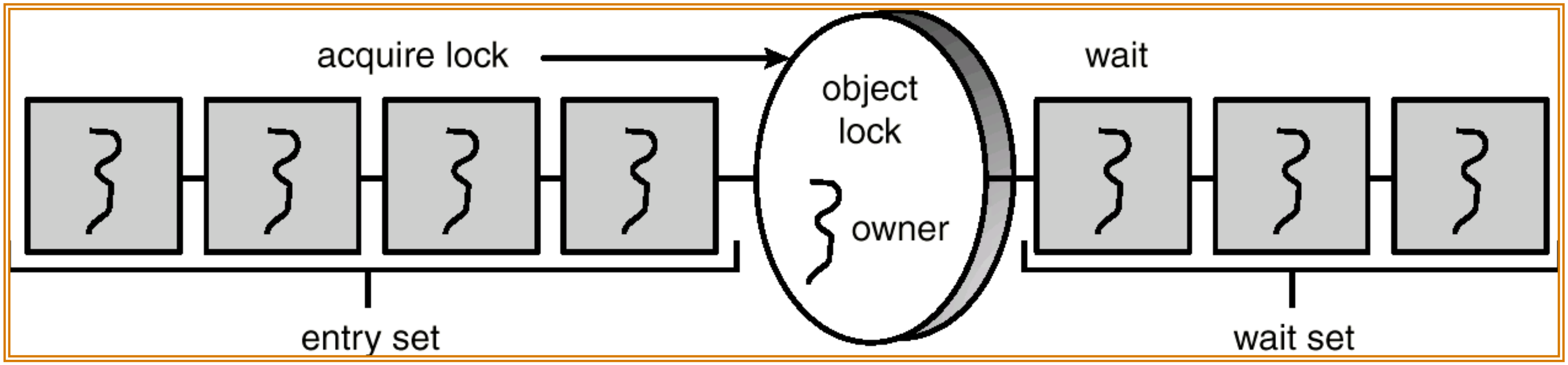
```
public synchronized void insert(Object item) {
   while (count == BUFFER SIZE) {
               try {
                       wait();
               }
               catch (InterruptedException e) { }
   }
   ++count;
   buffer[in] = item;
   in = (in + 1) % BUFFER SIZE;
   notify();
}
```

remove() with wait/notify Methods

```
public synchronized Object remove() {
   Object item;
   while (count == 0) {
                try {
                        wait();
                }
                catch (InterruptedException e) { }
   }
   --count;
   item = buffer[out];
   out = (out + 1) % BUFFER SIZE;
   notify();
   return item;
}
```

Entry and Wait Sets

Scenario in which multiple producers and consumers can be in wait set for BUFFER_SIZE = 1



Time-step	Entry set	Buffer state	Wait set
t	P0	EMPTY	C0, C1
t+1	C0, P1	FULL	C1
t+2	C0	FULL	P1, C1

java.util.concurrent

- General purpose toolkit for developing concurrent applications
  - import java.util.concurrent.*

- Goals: "Something for Everyone!"
  - Make some problems trivial to solve by everyone

    Develop thread-safe classes, such as servlets, built on concurrent building blocks like `ConcurrentHashMap`
  - Make some problems easier to solve by concurrent programmers

    Develop concurrent applications using thread pools, barriers, latches, and blocking queues
  - Make some problems possible to solve by concurrency experts

    Develop custom locking classes, lock-free algorithms

- HJ approach
  - Build HJ runtime on top of java.util.concurrent library

List of j.u.c. libraries

- Atomics: java.util.concurrent.atomic
  - Atomic[Type]
  - Atomic[Type]Array
  - Atomic[Type]FieldUpdater
  - Atomic{Markable,Stampable}Reference
- Concurrent Collections
  - ConcurrentMap
  - ConcurrentHashMap
  - CopyOnWriteArray{List,Set}
- Locks: java.util.concurrent.locks
  - Lock
  - Condition
  - ReadWriteLock
  - AbstractQueuedSynchronizer
  - LockSupport
  - ReentrantLock
  - ReentrantReadWriteLock
- Synchronizers
  - CountDownLatch
  - Semaphore
  - Exchanger
  - CyclicBarrier
- Executors
  - Executor
  - ExecutorService
  - ScheduledExecutorService
  - Callable
  - Future
  - ScheduledFuture
  - Delayed
  - CompletionService
  - ThreadPoolExecutor
  - ScheduledThreadPoolExecutor
  - AbstractExecutorService
  - Executors
  - FutureTask
  - ExecutorCompletionService
- Queues
  - BlockingQueue
  - ConcurrentLinkedQueue
  - LinkedBlockingQueue
  - ArrayBlockingQueue
  - SynchronousQueue
  - PriorityBlockingQueue
  - DelayQueue

Key Functional Groups in j.u.c.

- Atomic variables
  - The key to writing lock-free algorithms
- Concurrent Collections:
  - Queues, blocking queues, concurrent hash map, …
  - Data structures designed for concurrent environments
- Locks and Conditions
  - More flexible synchronization control
  - Read/write locks
- Executors, Thread pools and Futures
  - Execution frameworks for asynchronous tasking
- Synchronizers: Semaphore, Latch, Barrier, Exchanger
  - Ready made tools for thread coordination

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 {
  void lockInterruptibly() throws InterruptedException;
  boolean tryLock();
  boolean tryLock(long timeout, TimeUnit unit)
                           throws InterruptedException;
  void unlock();
  Condition newCondition();
   // can associate multiple condition vars with 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: not wait
    - Causes current thread to wait until it is signaled or interrupted
    - Variants available with support for interruption and timeout
  - void signal() // NOTE: not notify
    - Wakes up one thread waiting on *this* condition
  - void signalAll() // NOTE: not notifyAll()
    - 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 implementation 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 implementation 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 implementation 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
  - Multiple concurrent reads of same memory: *Not* a problem
  - Multiple concurrent writes of same memory: Problem
  - Multiple 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

Consider a hashtable with one coarse-grained lock

- So only one thread can perform operations at a time

But suppose:

- There are many simultaneous `lookup` operations
- `insert` operations are very 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 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();
   }
}
```