COMP 322: Fundamentals of Parallel Programming

Lecture 28: Advanced Locking

Vivek Sarkar, Eric Allen
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu

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

Locks and Conditions in java.util.concurrent library

- 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

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:

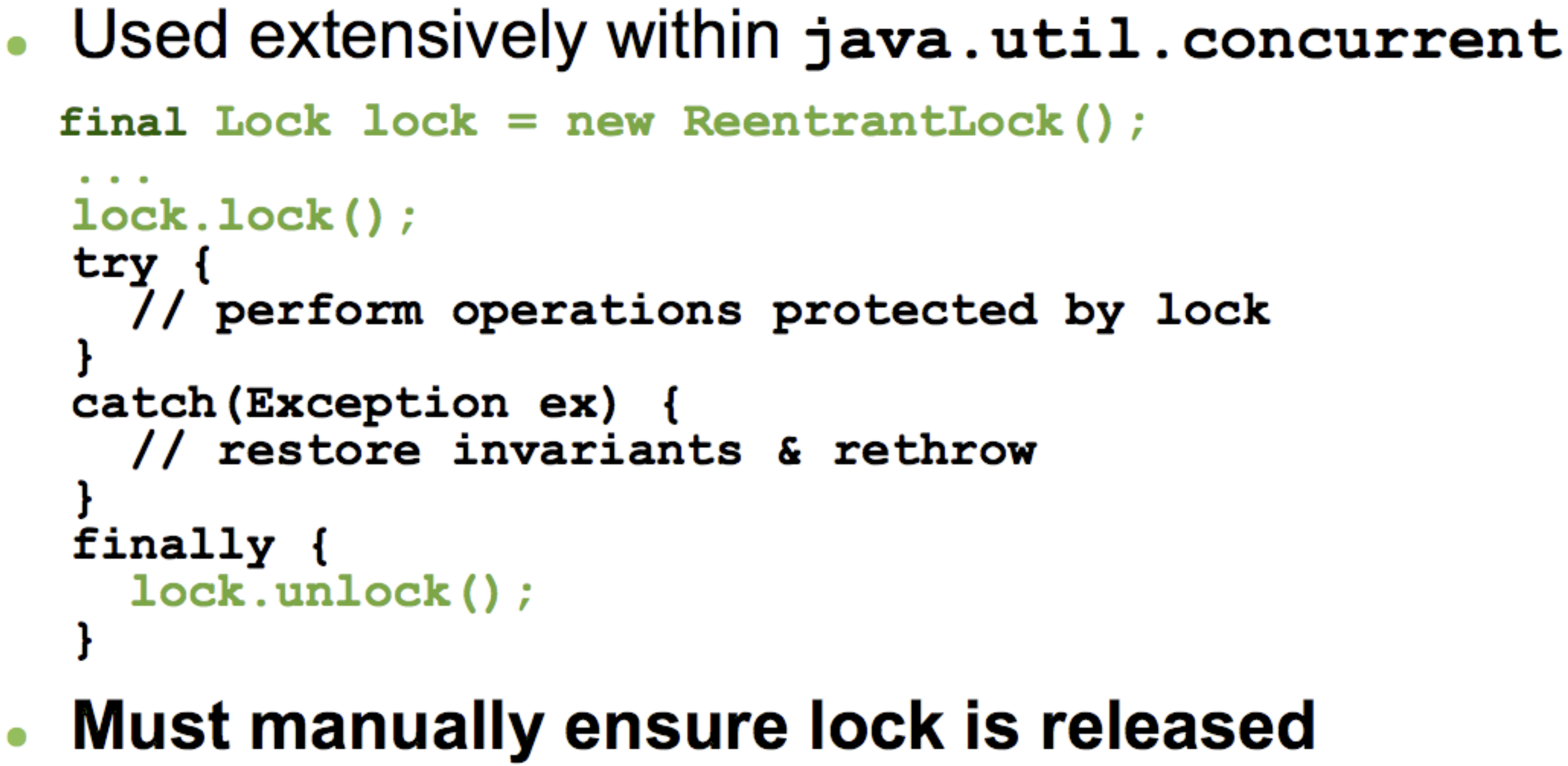
- L1.lock() … L2.lock() … L1.unlock() … L3.lock() … L2.unlock() ….

java.util.concurrent.locks.Lock interface

```
interface Lock {
   void lock();
   void lockInterruptibly() throws InterruptedException;
   boolean tryLock(); // return false if lock is not obtained
   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

```
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

```
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 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();
  }
}
```