Lecture 27
Non-blocking Algorithms
java.util.concurrent Performance

• java.util.concurrent classes often out-perform synchronized alternatives ... why?
  – Good engineering
  – Clever locking strategies (e.g. lock striping)
  – Atomic variables
  – Nonblocking algorithms

• Atomic variables
  Lock-free atomic operations

• Non-blocking algorithms
  Ensure thread safety without locks

• Both rely on use of low-level atomic hardware primitives
Problems with Locking

• When a thread fails to acquire lock it suspends
  – Context switching, resumption incur costs
  – Thread cannot perform other computation
• While a thread is holding a lock, no other thread that needs that lock can progress
• Locking is pessimistic
  If contention is infrequent, most locking is unneeded
Alternatives to Locking

• **Optimistic retrying**
  – No locking on reading
  – For updating
    • Make copy of state
    • Update copy
    • Commit copy if no collision, else retry
  – Especially useful for small single-address operations, e.g. incrementing integers, etc.

• Many processors provide hardware-level atomic operations to support optimistic trying
Compare-and-Swap (CAS)

• An atomic operation provided by some processors
  – Takes three arguments
    • Location V
    • Expected value E
    • New value N
  – Operation (in pseudo-code)
    ```
    tmp = V;
    if (V == E) V = N;
    return tmp;
    ```
• How can you tell if update happened?
  – If the value returned is the same as E, update happened
  – Otherwise, it did not
• Related operation: compare-and-set
  Returns boolean indicating if update was successful
Thread-Safe Incrementing Via CAS

• Suppose $i$ is a shared field, $\text{tmp}$ is local
• Here is how threads can increment $i$ using CAS
  
  ```
  do {
    tmp = i;
  } while (tmp != CAS (i, tmp, tmp+1))
  ```

• (Why) does this work?
  – CAS is atomic
  – Provided CAS results are visible (like volatile updates),
    while test holds only if update fails!
  – Keep retrying until update is successful

• No locking! But there is a possibility of starvation
Atomic Classes in Java

• Thread-safe holder classes for scalars, references

• Support atomic operations
  – Compare-and-set (CAS)
  – Get, set and arithmetic (where applicable)

• JVM uses best construct available on platform
  – CAS, load-linked/store-conditional, locks
Some Public Methods in AtomicInteger

- **int get()**
  Returns the current value

- **void set(int newValue)**
  Sets to the given value

- **int getAndSet(int newValue)**
  Atomically sets to the given value and returns the old value

- **boolean compareAndSet(int expected, int update)**
  Atomically sets the value to update if current value == expected
Other Atomic Classes

- AtomicInteger
- AtomicLong
- AtomicBoolean
- AtomicReference<T>
  - set(), get() operations
  - getAndSet() atomically sets to new value, returns old value
  - boolean compareAndSet(T expect, T new)
  - Arithmetic (increment, decrement) as appropriate
    - getAndAdd() – return old value
    - addAndGet() – return new value
    - getAndIncrement()
    - incrementAndGet()

- Also AtomicIntegerArray, AtomicLongArray, AtomicReferenceArray
  - Each element in array supports atomic operations
Summary of Atomic Variables

• Generalization of volatile variables
• Allows atomic read-modify-write operations without intrinsic locking
• Scope of contention limited to a single variable
• Faster than locking -- no scheduling impact
• In general, doesn’t support atomic check-then-act sequences
Performance Comparison

• In class
  – On simple examples involving incrementing counters ...
  – … with different levels of contention ...
  – … AtomicInteger is about 2-3x faster than synchronized implementations (Dell Latitude E6500, Windows 8.1)

• Two implementations of a pseudo-random number generator (PRNG)
  – One uses locks: ReentrantLockPseudoRandom.java
  – One is non-blocking: AtomicPseudoRandom.java

• PRNG issues
  – Next value based on last value, so you need to remember last value

• Next table summarizes results for this study
Comparing Performance

![Graph showing comparison of performance between Atomic and Sync methods across different numbers of threads.]

- **#Threads**: MacPro, OS X Snow Leopard, 8 cores, 8 GB of RAM
- **#Updates**

©2012-17 University of Maryland
Updating Complex Objects

• Suppose we have an invariant over two fields
  – Maintaining this in the past required locking
  – Can we do this using CAS?

• General strategy
  – Turn compound update into single update
  – Then use optimistic-retry strategy!

• This may require creating a “helper class” whose objects store the mutable state
  – AtomicReferences to these objects can then be updated using compare-and-set and looping
Example (from book): Integer Ranges

• Want: class of objects for representing ranges of integers
  – Mathematical notation: $[l, u]$ where $l$ is lower bound, $u$ is upper bound
  – Objects will have two fields: one for $l$, one for $u$
  – Invariant: $l \leq u$

• To make lock-free thread-safe version of ranges, we will use a helper class for representing pairs of integers
public class IntPair {

    private final int x;
    private final int y;

    IntPair(int x, int y) {
        this.x = x;
        this.y = y;
    }

    int getX() {
        return x;
    }

    int getY() {
        return y;
    }
}
public class IntRange {

    private final AtomicReference<IntPair> values;

    public IntRange(int lower, int upper) {
        if (lower > upper) { throw new IllegalArgumentException(); } else {
            values = new AtomicReference<IntPair> (new IntPair(lower,upper));
        }
    }

    public void setLower(int newLower) {
        IntPair oldV, newV;
        do {
            oldV = values.get(); // gets the current value atomically
            if (newLower > oldV.getY()) throw new IllegalArgumentException();
            newV = new IntPair(newLower, oldV.getY());
        } while (!values.compareAndSet(oldV,newV));
    }
}