Atomic Classes & Operations
(Part 2)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

• Be aware of the Java memory model
• Understand how Java atomic operations provide concurrent programs with lock-free & thread-safe mechanisms to read from & write to single variables
• Recognize how Java AtomicLong is implemented
Using Atomic Operations in Practice
Java uses CAS extensively in the JVM & portions of java.util.concurrent*

Using Atomic Operations in Practice

- Applications
- Additional Frameworks & Languages
- Threading & Synchronization Packages
- Java Virtual Machine
- System Libraries
- Operating System Kernel

C++/C
Java/JNI
Using Atomic Operations in Practice

- Java uses CAS extensively in the JVM & portions of java.util.concurrent*, e.g.
  - compareAndSwapLong()

```java
public final class Unsafe {
    public final native boolean compareAndSwapLong(Object o,
                                                    long offset,
                                                    long expected,
                                                    long updated);
}
```

Using Atomic Operations in Practice

- Java uses CAS extensively in the JVM & portions of java.util.concurrent*, e.g.
  - `compareAndSwapLong()`

```java
public final class Unsafe {
    public final native boolean compareAndSwapLong(Object o, long offset, long expected, long updated) {
        START_ATOMIC();
        int *base = (int *) o;
        int oldValue = base[offset];
        if (oldValue == expected) {
            base[offset] = updated;
            END_ATOMIC();
        }
        return oldValue;
    }
    ...

This C-like pseudo-code atomically compares the contents of memory with an expected value, modifies the contents to an updated value iff they are the same, & returns the old value.
```

Java uses CAS extensively in the JVM & portions of java.util.concurrent*, e.g.
• compareAndSwapLong()

This C-like pseudo-code *atomically* compares the contents of memory with an expected value, modifies the contents to an updated value iff they are the same, & returns the old value.

```java
public final class Unsafe {
    public final native boolean compareAndSwapLong(Object o,
                                                   long offset,
                                                   long expected,
                                                   long updated) {
        START_ATOMIC();
        int *base = (int *) o;
        int oldValue = base[offset];
        if (oldValue == expected)
            base[offset] = updated;
        END_ATOMIC();
        return oldValue;
    }
    ...
}
```
Java uses CAS extensively in the JVM & portions of java.util.concurrent*, e.g.
- `compareAndSwapLong()`

```java
public final class Unsafe {
    public final native boolean compareAndSwapLong(Object o,
                                                  long offset,
                                                  long expected,
                                                  long updated) {
        START_ATOMIC();
        int *base = (int *) o;
        int oldValue = base[offset];
        if (oldValue == expected)
            base[offset] = updated;
        END_ATOMIC();
        return oldValue;
    }
    ...
```

This C-like pseudo-code atomically compares the contents of memory with an expected value, modifies the contents to an updated value iff they are the same, & returns the old value.
Using Atomic Operations in Practice

- Java uses CAS extensively in the JVM & portions of java.util.concurrent*, e.g.
  - compareAndSwapLong()

```
public final class Unsafe {

    public final native boolean compareAndSwapLong(Object o,
        long offset, long expected, long updated) {

        START_ATOMIC();
        int *base = (int *) o;
        int oldValue = base[offset];
        if (oldValue == expected) {
            base[offset] = updated;
        }
        END_ATOMIC();
        return oldValue;
    }

    ...
```

This C-like pseudo-code atomically compares the contents of memory with an expected value, modifies the contents to an updated value iff they are the same, & returns the old value.
Java uses CAS extensively in the JVM & portions of java.util.concurrent*, e.g.

- compareAndSwapLong()

public final class Unsafe {
    public final native boolean compareAndSwapLong(Object o,
            long offset,
            long expected,
            long updated) {

        START_ATOMIC();
        int *base = (int *) o;
        int oldValue = base[offset];
        if (oldValue == expected)
            base[offset] = updated;
        END_ATOMIC();
        return oldValue;
    }
    ...

This C-like pseudo-code atomically compares the contents of memory with an expected value, modifies the contents to an updated value iff they are the same, & returns the old value.
Using Atomic Operations in Practice

- Java uses CAS extensively in the JVM & portions of java.util.concurrent*, e.g.
  - `compareAndSwapLong()`
- AtomicLong uses method `compareAndSwapLong()`

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicLong.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicLong.html)
Java uses CAS extensively in the JVM & portions of java.util.concurrent*, e.g.

- `compareAndSwapLong()`
- AtomicLong uses the method `compareAndSwapLong()`

```java
public class AtomicLong
    ...
    {
        private volatile long value;
        ...

        private static final Unsafe unsafe
        = Unsafe.getUnsafe();

        private static final long valueOffset;

        static {
            ...
            valueOffset = unsafe.
                objectFieldOffset
                (AtomicLong.class.
                getDeclaredField("value"));
            ...
        }

    }
    ...
```

**Java reflection is used to determine & store the offset of volatile ‘value’**

See [docs.oracle.com/javase/tutorial/reflect](docs.oracle.com/javase/tutorial/reflect)
Java uses CAS extensively in the JVM & portions of java.util.concurrent*, e.g.
- `compareAndSwapLong()`
- AtomicLong uses the method `compareAndSwapLong()`

`Unsafe.getAndAddLong()` atomically updates a value at an offset in the object

```java
public final class Unsafe {
    public final int getAndAddLong
        (Object o, long offset, long delta) {
        int v;
        do {
            v = getIntVolatile
                (o, offset);
        } while (!compareAndSwapLong
            (o, offset,
             v, v + delta));
        return v;
    }
}
```
Using Atomic Operations in Practice

- Java uses CAS extensively in the JVM & portions of java.util.concurrent*, e.g.
  - `compareAndSwapLong()`
  - AtomicLong uses the method `compareAndSwapLong()`

```java
public final class Unsafe {
    public final int getAndAddLong
        (Object o, long offset, long delta) {
        int v;
        do {
            v = getIntVolatile
                (o, offset);
        } while (!compareAndSwapLong
                (o, offset, v, v + delta));
        return v;
    }
}
```

This “lock-free” call runs atomically
Java uses CAS extensively in the JVM & portions of java.util.concurrent*, e.g.

- compareAndSwapLong()
- AtomicLong uses the method compareAndSwapLong()

```
public final class Unsafe {
    public final int getAndAddLong
        (Object o, long offset, long delta) {
        int v;
        do {
            v = getIntVolatile (o, offset);
        } while (!compareAndSwapLong
            (o, offset, v, v + delta));
        return v;
    }
}
```

The ‘offset’ is relative to the start of object ‘o’
Using Atomic Operations in Practice

- Java uses CAS extensively in the JVM & portions of java.util.concurrent*, e.g.
  - `compareAndSwapLong()`
  - AtomicLong uses the method `compareAndSwapLong()`

```java
public final class Unsafe {
    public final int getAndAddLong
        (Object o,
         long offset,
         long delta) {
        int v;
        do {
            v = getIntVolatile
                 (o, offset);
        } while (!compareAndSwapLong
                  (o, offset,
                   v, v + delta));
        return v;
    }
}
```

‘v’ is the value at ‘offset’ into object ‘o’
public final class Unsafe {
    public final int getAndAddLong
            (Object o,
             long offset,
             long delta) {

        int v;
        do {
            v = getIntVolatile
                  (o, offset);
        } while (!compareAndSwapLong
                    (o, offset,
                     v, v + delta));

        return v;
    }
}

• Java uses CAS extensively in the JVM & portions of java.util.concurrent*, e.g.
  • compareAndSwapLong()
  • AtomicLong uses the method compareAndSwapLong()

‘delta’ is atomically added to value ‘v’ iff ‘v’ hasn’t changed since it was read
Using Atomic Operations in Practice

- Java uses CAS extensively in the JVM & portions of java.util.concurrent*, e.g.
  - `compareAndSwapLong()`
  - AtomicLong uses the method `compareAndSwapLong()`

```java
public class AtomicLong {
  ...

  private volatile long value;
  ...

  public final long getAndIncrement() { return unsafe
    .getAndAddLong(this, valueOffset, 1L); }

  public final long getAndDecrement() { return unsafe
    .getAndAddLong(this, valueOffset, -1L); }
}
```

The `Unsafe.getAndAddLong()` method is used to increment & decrement values atomically!
Using Atomic Operations in Practice

- Concurrent programs should use atomic operations carefully since they “busy wait”

See en.wikipedia.org/wiki.Busy_waiting
Using Atomic Operations in Practice

• Concurrent programs should use atomic operations carefully since they “busy wait”
  • However, some “spinning” can be useful in modern multi-core processors
    • e.g., due to context switching overhead needed for sleep locks

See [www.youtube.com/watch?v=sq0MX3fHkro](http://www.youtube.com/watch?v=sq0MX3fHkro)
End of Atomic Classes & Operations (Part 2)
1. Why is “spinning” effective for modern multi-core processors?
   a. “Lock-free” synchronizers are very efficient
   b. Modern multi-core processes don’t need to mask hardware interrupts explicitly
   c. Significant context switching overhead is incurred when putting a core to sleep & waking it up
   d. Multi-core processes leverage the benefits of Moore’s Law