Atomic Classes & Operations

(Part 1)

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
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
Overview of the Java Memory Model
Overview of the Java Memory Model

- Java’s memory model defines semantics of multi-threaded access to shared memory.

See gee.cs.oswego.edu/dl/cpj/jmm.html
Overview of the Java Memory Model

- Java’s memory model defines semantics of multi-threaded access to shared memory, e.g.
  - Which instruction reorderings are allowed in memory

There are a number of potential sources of reordering, e.g., the Java compiler, the JIT, & processor caches, etc.
Overview of the Java Memory Model

- Java’s memory model defines semantics of multi-threaded access to shared memory, e.g.
  - Which instruction reorderings are allowed in memory
  - Should not be overly restrictive, to enable hardware optimizations

```
Thread1

x = y = 0
x = 1
j = y

Thread2

y = 1
i = x

start threads

It can end up that i = 0 & j = 0 due to local caching effects in Thread1 & Thread2
```

See en.wikipedia.org/wiki/Memory_ordering
Overview of the Java Memory Model

Java’s memory model defines semantics of multi-threaded access to shared memory, e.g.

- Which instruction reorderings are allowed in memory
- Which program outputs may occur in a correct JVM implementation

See [docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.3](docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.3)
Overview of the Java Memory Model

- Java’s memory model defines semantics of multi-threaded access to shared memory, e.g.:
  - Which instruction reorderings are allowed in memory
  - Which program outputs may occur in a correct JVM implementation
  - Should not be too generous such that values appear randomly!

![Diagram showing two threads and their operations involving variables r1, x, y, and r2, illustrating a scenario where r1 = r2 = 42 is not allowed.]
Overview of the Java Memory Model

- Reading about Java’s memory model is as much fun as watching paint dry.

Overview of the Java Memory Model

• Reading about Java’s memory model is as much fun as watching paint dry.

Fortunately, you needn’t understand all these memory model details – you just need to know how to use Java synchronizers properly!!
Overview of Java

Atomic Classes
Overview of Java Atomic Classes

• The java.util.concurrent.atomic package several types of atomic actions on objects

---

**Package java.util.concurrent.atomic**

A small toolkit of classes that support lock-free thread-safe programming on single variables.

See: Description

**Class Summary**

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AtomicBoolean</td>
<td>A boolean value that may be updated atomically.</td>
</tr>
<tr>
<td>AtomicInteger</td>
<td>An int value that may be updated atomically.</td>
</tr>
<tr>
<td>AtomicIntegerArray</td>
<td>An int array in which elements may be updated atomically.</td>
</tr>
<tr>
<td>AtomicIntegerFieldUpdater&lt;T&gt;</td>
<td>A reflection-based utility that enables atomic updates to designated volatile int fields of designated classes.</td>
</tr>
<tr>
<td>AtomicLong</td>
<td>A long value that may be updated atomically.</td>
</tr>
<tr>
<td>AtomicLongArray</td>
<td>A long array in which elements may be updated atomically.</td>
</tr>
<tr>
<td>AtomicLongFieldUpdater&lt;T&gt;</td>
<td>A reflection-based utility that enables atomic updates to designated volatile long fields of designated classes.</td>
</tr>
</tbody>
</table>

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html)
Overview of Java Atomic Classes

• The java.util.concurrent.atomic package several types of atomic actions on objects
  • Atomic variables
    • Provide lock-free thread-safe operations on single variables

See docs.oracle.com/javase/tutorial/essential/concurrency/atomicvars.html
Overview of Java Atomic Classes

- The java.util.concurrent.atomic package several types of atomic actions on objects
  - **Atomic variables**
    - Provide lock-free thread-safe operations on single variables
    - e.g., AtomicLong supports atomic “compare-and-swap” operations

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)
Overview of Java Atomic Classes

• The java.util.concurrent.atomic package several types of atomic actions on objects
  • Atomic variables
  • LongAdder
    • Allows multiple threads to update a common sum efficiently under high contention

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html
Overview of Atomic Operations
Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers.
Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
- CAS – "compare-and-swap"

```c
int compareAndSwap(int *loc,
                   int expected,
                   int updated) {

    START_ATOMIC();
    int oldValue = *loc;
    if (oldValue == expected)
        *loc = updated;

    END_ATOMIC();
    return oldValue;
}
```

Compare-and-swap atomically compares the current contents of a memory location to a given value & iff they are the same it modifies the contents of that memory location to a given new value & returns the old value

Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.

**CAS – “compare-and-swap”**

```java
int compareAndSwap(int *loc, 
                   int expected, 
                   int updated) {

    START_ATOMIC();
    int oldValue = *loc;
    if (oldValue == expected) 
        *loc = updated;
    END_ATOMIC();

    return oldValue;
}
```

**Compare-and-swap atomically** compares the current contents of a memory location to a given value & iff they are the same it modifies the contents of that memory location to a given new value & returns the old value.
Overview of Atomic Operations

• Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
• CAS – “compare-and-swap”

int compareAndSwap(int *loc, int expected, int updated) {
    START_ATOMIC();
    int oldValue = *loc;
    if (oldValue == expected)
        *loc = updated;
    END_ATOMIC();
    return oldValue;
}

Compare-and-swap atomically compares the current contents of a memory location to a given value & iff they are the same it modifies the contents of that memory location to a given new value & returns the old value.
Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
- CAS – “compare-and-swap”

```c
int compareAndSwap(int *loc, int expected, int updated) {
    START_ATOMIC();
    int oldValue = *loc;
    if (oldValue == expected)
        *loc = updated;
    END_ATOMIC();
    return oldValue;
}
```

Compare-and-swap atomically compares the current contents of a memory location to a given value & iff they are the same it modifies the contents of that memory location to a given new value & returns the old value
Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
- CAS – “compare-and-swap”

```c
int compareAndSwap(int *loc, int expected, int updated) {
    START_ATOMIC();
    int oldValue = *loc;
    if (oldValue == expected) {
        *loc = updated;
    }
    END_ATOMIC();
    return oldValue;
}
```

*Compare-and-swap atomically compares the current contents of a memory location to a given value & iff they are the same it modifies the contents of that memory location to a given new value & returns the old value*
Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
- CAS – “compare-and-swap”

```c
void lock(int *mutex) {
    while (compareAndSwap(mutex, 0, 1) == 1)
        continue;
}
```

```c
int compareAndSwap(int *loc, int expected, int updated) {
    START_ATOMIC();
    int oldValue = *loc;
    if (oldValue == expected)
        *loc = updated;
    END_ATOMIC();
    return oldValue;
}
```

See [en.wikipedia.org/wiki/Spinlock](en.wikipedia.org/wiki/Spinlock)
Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
  
- CAS – “compare-and-swap”

```c
int compareAndSwap(int *loc, int expected, int updated) {
    START_ATOMIC();
    int oldValue = *loc;
    if (oldValue == expected)
        *loc = updated;
    END_ATOMIC();
    return oldValue;
}
```

```c
void lock(int *mutex) {
    while (compareAndSwap(mutex, 0, 1) == 1)
        continue;
}
```

*The lock() method uses `compareAndSwap()` to implement mutual exclusion (mutex) via a “spin-lock”*
Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.

- **CAS – “compare-and-swap”**

```c
int compareAndSwap(int *loc, int expected, int updated) {
    START_ATOMIC();
    int oldValue = *loc;
    if (oldValue == expected)
        *loc = updated;
    END_ATOMIC();
    return oldValue;
}
```

```c
void lock(int *mutex) {
    while (compareAndSwap(mutex, 0, 1) == 1)
        continue;
}
```

*compareAndSwap() checks if the location pointed to by **mutex** is 0 & iff that’s true it sets the value to 1*
void lock(int *mutex) {
    while (compareAndSwap(mutex, 0, 1) == 1)
        continue;
}

• Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
• CAS – “compare-and-swap”

int compareAndSwap(int *loc, int expected, int updated) {
    START_ATOMIC();
    int oldValue = *loc;
    if (oldValue == expected)
        *loc = updated;
    END_ATOMIC();
    return oldValue;

Overview of Atomic Operations

compareAndSwap() checks if the location pointed to by mutex is 0 & iff that’s true it sets the value to 1
• Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
• CAS – “compare-and-swap”

```c
int compareAndSwap(int *loc,
                   int expected,
                   int updated) {
    START_ATOMIC();
    int oldValue = *loc;
    if (oldValue == expected)
        *loc = updated;
    END_ATOMIC();
    return oldValue;
}
```

```c
void lock(int *mutex) {
    while (compareAndSwap(mutex, 0, 1) == 1)
        continue;
}
```

`compareAndSwap()` checks if the location pointed to by mutex is 0 & iff that’s true it sets the value to 1
Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
- **CAS** – “compare-and-swap”

```c
void lock(int *mutex) {
    while (compareAndSwap(mutex, 0, 1) == 1)
        continue;
}

int compareAndSwap(int *loc, int expected, int updated) {
    START_ATOMIC();
    int oldValue = *loc;
    if (oldValue == expected)
        *loc = updated;
    END_ATOMIC();
    return oldValue;
}
```

*If compareAndSwap() returns 1 that means the mutex is “acquired” so the loop keeps spinning*
void lock(int *mutex) {
    while (compareAndSwap(mutex, 0, 1) == 1)
        continue;
}

void unlock(int *mutex) {
    START_ATOMIC();
    *mutex = 0;
    END_ATOMIC();
}

int compareAndSwap(int *loc, int expected, int updated) {
    START_ATOMIC();
    int oldValue = *loc;
    if (oldValue == expected)
        *loc = updated;
    END_ATOMIC();
    return oldValue;
}

Overview of Atomic Operations

• Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
• CAS - “compare-and-swap”

The *unlock() method atomically resets the mutex value to 0
Overview of Atomic Operations

- Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.
- CAS – “compare-and-swap”

```c
int compareAndSwap(int *loc, int expected, int updated) {
    START_ATOMIC();
    int oldValue = *loc;
    if (oldValue == expected)
        *loc = updated;
    END_ATOMIC();
    return oldValue;
}
```

```c
void lock(int *mutex) {
    while (compareAndSwap(mutex, 0, 1) == 1)
        continue;
}
```

```c
void unlock(int *mutex) {
    START_ATOMIC();
    *mutex = 0;
    END_ATOMIC();
}
```

The unlock() method atomically resets the mutex value to 0.
Atomics operations in Java are implemented in hardware with some support at the OS & VM layers, e.g.

- **CAS** – “compare-and-swap”

```c
int compareAndSwap(int *loc, int expected, int updated) {
    START_ATOMIC();
    int oldValue = *loc;
    if (oldValue == expected)
        *loc = updated;
    END_ATOMIC();
    return oldValue;
}
```

```c
void lock(int *mutex) {
    while (compareAndSwap(mutex, 0, 1) == 1)
        continue;
}
```

```c
void unlock(int *mutex) {
    START_ATOMIC();
    *mutex = 0;
    END_ATOMIC();
}
```

The unlock() method atomically resets the mutex value to 0.
Overview of Atomic Operations

- Atomic operations can be implemented other ways
  - e.g., “test-and-set”

```c
int testAndSet(int *loc) {
    int oldValue;
    START_ATOMIC();
    oldValue = *loc;
    *loc = 1; // 1 == locked
    END_ATOMIC();
    return oldValue;
}
```

Test-and-set atomically modifies the contents of a memory location & returns its old value

Overview of Atomic Operations

• Atomic operations can be implemented other ways
  • e.g., “test-and-set”

```c
int testAndSet(int *loc) {
    int oldValue;
    START_ATOMIC();
    oldValue = *loc;
    *loc = 1; // 1 == locked
    END_ATOMIC();
    return oldValue;
}
```

Test-and-set **atomically** modifies the contents of a memory location & returns its old value
Overview of Atomic Operations

• Atomic operations can be implemented other ways
• e.g., “test-and-set”

Test-and-set atomically modifies the contents of a memory location & returns its old value

```c
int testAndSet(int *loc) {
    int oldValue;
    START_ATOMIC();
    oldValue = *loc;
    *loc = 1; // 1 == locked
    END_ATOMIC();
    return oldValue;
}
```
Overview of Atomic Operations

- Atomic operations can be implemented other ways
  - e.g., “test-and-set”

```c
int testAndSet(int *loc) {
    int oldValue;
    START_ATOMIC();
    oldValue = *loc;
    *loc = 1; // 1 == locked
    END_ATOMIC();
    return oldValue;
}
```

*Test-and-set atomically modifies the contents of a memory location & returns its old value*
void lock(int *loc) {
    while (testAndSet(loc) == 1);
}

void unlock(int *loc) {
    START_ATOMIC();
    *loc = 0; // 1 == locked
    END_ATOMIC();
}

Test-and-set can also be used to implement a spin-lock mutex

int testAndSet(int *loc) {
    int oldValue;
    START_ATOMIC();
    oldValue = *loc;
    *loc = 1; // 1 == locked
    END_ATOMIC();
    return oldValue;
}
Overview of Atomic Operations

- compareAndSwap() provides a more general solution than the testAndSet()

```c
int compareAndSwap(int *loc, int updated) {
    START_ATOMIC();
    int oldValue = *loc;
    *loc = updated;
    END_ATOMIC();
    return oldValue;
}
```

```c
int testAndSet(int *loc) {
    int oldValue;
    START_ATOMIC();
    oldValue = *loc;
    *loc = 1; // 1 == locked
    END_ATOMIC();
    return oldValue;
}
```

```c
int compareAndSwap(int *loc, int expected, int updated) {
    START_ATOMIC();
    int oldValue = *loc;
    if (oldValue == expected) {
        *loc = updated;
    }
    END_ATOMIC();
    return oldValue;
}
```
Overview of Atomic Operations

• `compareAndSwap()` provides a more general solution than the `testAndSet()`
• e.g., it can set the value to something other than 1 or 0

```c
int compareAndSwap(int *loc, int expected, int updated) {
    START_ATOMIC();
    int oldValue = *loc;
    *loc = updated;
    END_ATOMIC();
    return oldValue;
}
```

```c
int testAndSet(int *loc) {
    START_ATOMIC();
    int oldValue = *loc;
    *loc = 1; // 1 == locked
    END_ATOMIC();
    return oldValue;
}
```

This capability is used by various Atomic* classes in Java
Human Known Use of Atomic Operations
Human Known Use of Atomic Operations

- A “human” known use of atomic operations is a transporter

See [en.wikipedia.org/wiki/Transporter_(Star_Trek)](en.wikipedia.org/wiki/Transporter_(Star_Trek))
Human Known Use of Atomic Operations

- A “human” known use of atomic operations is a transporter
- Converts a person or object into an energy pattern, which is then "beams" to a destination & reconverted back into matter
Human Known Use of Atomic Operations

• A “human” known use of atomic operations is a transporter
  • Converts a person or object into an energy pattern, which is then "beams“ to a destination & reconverted back into matter
  • The transportation process must occur atomically or a horrible accident will occur!

See en.wikipedia.org/wiki/Transporter_(Star_Trek)#Transporter_accidents
End of Atomic Classes & Operations (Part 1)