Built-in Concurrency Primitives in Java Programming Language

by Yourii Martiak and Mahir Atmis
Overview

One of the many strengths of Java is the built into the programming language support for concurrency and multi-threading. With multi-threading and concurrency, comes a need for coordinating activities and data access to shared resources among multiple threads. Java provides several mechanisms for such communication. The most basic form of communication between threads can be achieved by using synchronization, which is in turn implemented using monitors. Other alternatives include the use of volatile modifier and a selection of many thread-safe classes in the java.util.concurrent package. They provide many alternative ways to do inter-thread communication, which includes atomics, wait-free and lock-free data structures.

Purpose

The purpose of this paper is to examine many different concurrency primitives and data structures built into the Java programming language, look into their implementation details and highlight some obvious strengths, weaknesses and differences between them.

Synchronization

The most basic method of communication between threads in Java is synchronization, which is implemented using monitors. A unique monitor is assigned to each object in Java and any thread dealing with the objects reference can lock and unlock the monitor. A locked monitor ensures exclusive access and any other threads trying to lock the same monitor are blocked until the monitor is unlocked.

Java's monitor supports two kinds of thread synchronization: mutual exclusion and cooperation. Mutual exclusion, which is supported in the Java virtual machine via object locks, enables multiple threads to independently work on shared data without interfering with each other. Cooperation, which is supported in the Java virtual machine via the wait and notify methods of class Object, enables threads to work together towards a common goal.

Figure 1.1 A Sample Java Monitor

Perhaps the most common form of synchronization in Java is the use of synchronized keyword, more
specifically synchronized method. In order for JVM to identify synchronized blocks of code, the runtime constant pool is examined for an ACC_SYNCHRONIZED flag, which is checked by the method invocation instructions.

The body of synchronized method is executed only after the lock action on the monitor has been performed. If the method is an instance method, the monitor associated with an instance of that class is used. If the method is static, the monitor associated with the Class object representing the class type is used. A single thread that already owns the lock, can lock the same object multiple times. When this happens, JVM maintains a count of times the lock on the monitor was called and the count is incremented with each lock and decremented with each unlock. Unlock action is automatically performed after the method body finished executing or the exception was thrown.

Once the method is determined to be synchronized, a set of additional bytecode instructions is added during the compilation. For example:

```java
void onlyMe(Foo f) {
    synchronized(f) {
        doSomething();
    }
}
```

is compiled into the following Java bytecode:

```java
Method void onlyMe(Foo)
0  aload_1                        // Push f
1  dup                           // Duplicate it on the stack
2  astore_2                      // Store duplicate in local variable 2
3  monitorenter                  // Enter the monitor associated with f
4  aload_0                       // Holding the monitor, pass this and...
5  invokevirtual #5              // ...call Example.doSomething()V
8  aload_2                       // Push local variable 2 (f)
9  monitorexit                   // Exit the monitor associated with f
10 goto 18                      // Complete the method normally
13 astore_3                     // In case of any throw, end up here
14 aload_2                       // Push local variable 2 (f)
15 monitorexit                   // Be sure to exit the monitor!
16 aload_3                       // Push thrown exception...
17 athrow                       // ...then rethrow the value to the invoker
18 return                       // Return in the normal case
```

Exception table:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Target</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>10</td>
<td>13</td>
<td>any</td>
</tr>
<tr>
<td>13</td>
<td>16</td>
<td>13</td>
<td>any</td>
</tr>
</tbody>
</table>

**Volatile Modifier**

A weaker form of synchronization is possible in Java using volatile field modifier. The effect of declaring a field volatile is somewhat similar to using a fully synchronized class protecting that field, as follows:
final class VFloat {
    private float value;
    //final synchronized void set(float f) { value = f; }
    final synchronized float get() { return value; }
}

The difference is that no locking is involved. Also, composite operations, such as “++”, on volatile fields
are not performed atomically. The only guarantee that a Java memory model provides is that if a field is declared
volatile, any value written to it is flushed and made visible before writer thread can perform any further memory
operation. Then, the reader threads must reload the value of volatile field upon each access. This way, Java
guarantees that any update made to a volatile field is propagated and is visible to other threads, in other words a
volatile write happens before any consecutive read to that field (a write is synchronized with all subsequent reads
by any other thread).

The main benefit of protecting fields with volatile modifier as opposed to synchronization is
performance, because no locking is involved. A volatile is likely to be more efficient and in the worst case
scenario, not more expensive than access to the same field surrounded with a synchronized block. It can also be
useful to declare fields volatile in the following situations:

- the field need not obey any invariants with respect to others
- writes to the field do not depend on its current value
- no thread ever writes an illegal value with respect to intended semantics
- the results of reordering do not depend on values of other non-volatile fields

To see this in practice, we had simple class:

class VolatileExample {
    int x = 0;
    volatile boolean v = false;
    public void writer() {
        x = 42;
        v = true;
    }

    public void reader() {
        if (v == true) {
            //uses x - guaranteed to see 42.
        }
    }
}

Which results in the following bytecode:

Compiled from "VolatileExample.java"
class VolatileExample extends java.lang.Object{
    int x;

    volatile boolean v;
VolatileExample();

Code:
0:   aload_0
1:   invokespecial #1; //Method java/lang/Object."<init>":()V
4:   aload_0
5:   iconst_0
6:   putfield   #2; //Field x:I
9:   aload_0
10:  iconst_0
11:  putfield  #3; //Field v:Z
14:  return

public void writer();

Code:
0:   aload_0
1:   bipush  42  // if this was not volatile, compiler would be free to
2:   putfield  #2; //Field x:I
3:   aload_0
4:   iconst_1
5:   putfield  #3; //Field v:Z
8:   return

public void reader();

Code:
0:   aload_0
1:   getfield   #3; //Field v:Z
4:   iconst_1
5:   if_icmpne  8 // at this point, using volatile ensures correct
6:     behaviour
7:   return

}

Wait and Notify

Wait and notify mechanism is an elegant mechanism which is a better alternative than polling (constantly
stealing CPU time by asking if the desired conditions are met) and which helps threads to coordinate. There
are two forms of wait( ). The first takes an argument in milliseconds which pauses the thread for that period of
time while the thread still has the lock. The second one takes no argument and is different from the first one in
following ways:

1. The object lock is released during the wait( ). Also, when a thread issues a wait() it is put into the wait
   set of its monitor.
2. You can come out of the wait( ) only by notify( ) or notifyAll( ) that are issued by another thread, or
   by letting the clock run out.
One fairly unique aspect of wait(), notify(), and notifyAll() is that these methods are part of the base class **Object**. They all are implemented as final methods in Object, so all classes have them. Also, all three methods can only be called from within a **synchronized** method. The functions of these methods are as follows:

- **wait()** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify() or notifyAll(). Until a notify() or notifyAll() is called the thread is a waiting thread.
- **notify()** wakes up the first thread that called wait() on the same object and the thread becomes an active thread.
- **notifyAll()** wakes up all the threads that called wait() on the same object. The highest priority thread will run first. If a thread cannot acquire the lock of its monitor again, it may issue another wait() or can terminate itself.

The signatures of these methods within Object are as shown here:

```java
final void wait() throws InterruptedException
final void notify()
final void notifyAll()
```

**Note:** final keyword before each of these methods is used to prevent overriding of those methods by subclasses.

We created a basic class calling wait/notify methods to see what Java bytecode gets generated. Here is the source code for our CommTest.java:

```java
public class CommTest {
    public void get() throws InterruptedException {
        synchronized(this) {
            while(true) {
                wait();
            }
        }
    }

    public void put() {
        synchronized(this) {
            notify();
        }
    }
}
```

And here is corresponding Java bytecode:

```java
public class CommTest extends java.lang.Object{
    public CommTest();
    Code:
    0:   aload_0
        1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
        4:   return

    public void get()   throws java.lang.Interrupted Exception;
    Code:
    0:   aload_0
        1:   dup
        2:   astore_1
        3:   monitorenter
```
The **final** keyword can be used to make sure that when an object is constructed, another thread accessing that object doesn't see it in a partially-constructed state. This is because when a constructor exits, the values of final fields are guaranteed to be visible to other threads accessing the constructed object. Also, if a field is final, JVM ensures that, once the object pointer is available to other threads, so are the correct values of that object's final fields. Thus, the objects whose all fields are final or references to immutable objects can be concurrently accessed without synchronization.

**Restrictions and limitations of using final**

1. **The values of the final fields within an object must be set before the constructor exits.**

Sample:
public class MyClass {
private final int myField = 3; // set before constructor exits
public MyClass() {
  // constructor
}
}

Or,
public class MyClass {
private final int myField; // declare
public MyClass() {
  // constructor
  myField = 3; // set
}
}

2. Storing a reference to an object in a final field only makes the reference immutable, not the actual object

private final List<Integer> lst = new ArrayList<Integer>();

One can modify the object with no problem as shown below:
lst.add(5);

However, the following would cause compile time error:
lst = someOtherList;

**Concurrent Libraries**

Concurrent library in Java is comprised from a few different packages representing atomics, locks and concurrent collections.

Atomics are represented by a small toolkit of classes that support lock-free thread-safe programming on single variables. In essence, these classes extend the notion of volatile values, fields and array elements and also provide atomic conditional update operations. These classes employ efficient machine-level atomic instructions that are available on contemporary processors. However, on some platforms, support may require some internal locking. Thus, the methods of atomic classes implementation are not guaranteed to be non-blocking – a thread may block transiently before performing atomic operation.

The most common versions of atomic primitives are implemented in classes AtomicBoolean, AtomicInteger, AtomicLong and AtomicReference. We examined some of the source code for the above implementations and the common pattern for conditional atomic methods is implemented as follows: (taken from AtomicLong.java, getAndIncrement() method)

```java
/**
 * Atomically increments by one the current value.
 * @return the previous value
 */
public final long getAndIncrement() {
  while (true) {
    long current = get();
    long next = current + 1;
```
if (compareAndSet(current, next))
    return current;
}
}

whereas, compareAndSet is:

/**
 * Atomically sets the value to the given updated value
 * if the current value {code ==} the expected value.
 * *
 * @param expect the expected value
 * @param update the new value
 * @return true if successful. False return indicates that
 * the actual value was not equal to the expected value.
 */
public final boolean compareAndSet(long expect, long update) {
    return unsafe.compareAndSwapLong(this, valueOffset, expect, update);
}

and unsafe is a class that invokes native implementation specific to the underlying hardware:

/**
 * Atomically update Java variable to <tt>x</tt> if it is currently
 * holding <tt>expected</tt>.
 * @return <tt>true</tt> if successful
 */
public final native boolean compareAndSwapLong(Object o, long offset, long expected, long x);

The memory effects for accesses and updates of atomics generally follow the rules for volatiles, as stated in The Java Language Specification, Third Edition (17.4 Memory Model):

- get has the memory effects of reading a volatile variable.
- set has the memory effects of writing (assigning) a volatile variable.
- weakCompareAndSet atomically reads and conditionally writes a variable but does not create any
  happens-before orderings, so provides no guarantees with respect to previous or subsequent reads and
  writes of any variables other than the target of the weakCompareAndSet.
- compareAndSet and all other read-and-update operations such as getAndIncrement have the memory
  effects of both reading and writing volatile variables.

Atomic classes are not general purpose replacements for corresponding primitives or wrapper classes
(java.lang.Integer, Long, Double, etc.) Instead, they are designed to be used as building blocks for implementing
non-blocking data structures and related classes.

Package java.concurrent.locks provides framework for locking and conditional waiting. Some of its
implementations include ReadWriteLock, ReentrantLock, Condition, etc.)

Java.util.concurrent package contains utility and collections classes commonly useful in concurrent
programming. This package includes a few small standardized extensible frameworks, as well as some classes
that provide useful functionality and are otherwise tedious or difficult to implement.

ConcurrentHashMap is a good example of how concurrent collections are done in Java. From the name,
it is obvious that the class implements functionality of concurrent hash map, supporting full concurrency of retrievals and adjusted concurrency for map updates. Retrievals do not employ locking and there is no built-in support for locking the whole structure. Nevertheless, this class's operations are considered to be thread-safe. However, retrievals may overlap with updates. In this case, they should reflect result of most recently completed update operations after retrieval completion. Here is the source code for a typical retrieval:

```java
public V get(Object key) {
    int hash = hash(key.hashCode());
    return segmentFor(hash).get(key, hash);
}
```

The above, identifies correct segment to which the key belongs. Then, the actual lookup within the segment is performed:

```java
V get(Object key, int hash) {
    if (count != 0) { // read-volatile
        HashEntry<K,V> e = getFirst(hash);
        while (e != null) {
            if (e.hash == hash && key.equals(e.key)) {
                V v = e.value;
                if (v != null)
                    return v;
            }
            e = e.next;
        }
    }
    return null;
}
```

Update operations are done under lock but instead of having one lock per structure, ConcurrentHashMap has many locks. Specifically, the structure is cleverly divided into multiple segments. The number of segments is adjustable (16 by default) and reflects the desired concurrency level. Then, each segment is protected by its own lock. This way, concurrent updates to the same map structure in different segments do not cause contention. A source code for put operation done on a particular segment shows the following:

```java
public V put(K key, V value) {
    if (value == null)
        throw new NullPointerException();
    int hash = hash(key.hashCode());
    return segmentFor(hash).put(key, hash, value, false);
}
```

```java
V put(K key, int hash, V value, boolean onlyIfAbsent) {
    lock();
    try {
        int c = count;
        if (c++ > threshold) // ensure capacity
            rehash();
        HashEntry<K,V>[] tab = table;
```
Java Memory Model With Regard to Concurrency

Java Language Specification defines that the results of a write by one thread are guaranteed to be visible to a read by another thread only if the write operation happens-before the read operation. The synchronized and volatile constructs can form happens-before relationships. In particular:

- Each action in a thread happens-before every action in that thread that comes later in the program’s order.
- An unlock (synchronized block or method exit) of a monitor happens-before every subsequent lock (synchronized block or method entry) of that same monitor. And because the happens-before relation is transitive, all actions of a thread prior to unlocking happen-before all actions subsequent to any thread locking that monitor.
- A write to a volatile field happens-before every subsequent read of that same field. Writes and reads of volatile fields have similar memory consistency effects as entering and exiting monitors, but do not entail mutual exclusion locking.

The methods of all classes in java.util.concurrent and its sub-packages extend these guarantees to higher-level synchronization. In particular (as it pertains to synchronization and locking):

- Actions in a thread prior to placing an object into any concurrent collection happen-before actions subsequent to the access or removal of that element from the collection in another thread.
- Actions prior to "releasing" synchronizer methods such as Lock.unlock, happen-before actions subsequent to a successful "acquiring" method such as Lock.lock on the same synchronizer object in another thread.
Conclusion

We presented a high-level overview of concurrency primitives in Java programming language. The biggest advantage of having a language level concurrency support is that there is a common platform for developing uniform usage patterns, which help to avoid “reinvent-the-wheel” effect when designing and implementing software for concurrent applications. When portability of “write once run everywhere” promise of Java is added on top of that, we have a perfect combination of portability and power.

The downside of this approach is obviously some overhead that needs to be incurred in order to create a set of “one-size-fit-all” patterns. However, the pros overweight the cons for the majority of the enterprise grade applications and as the language interpreters and compilers are able to generate faster code, the Java with its built-in concurrency support and a vast collection of APIs, has a great future and will see a widespread acceptance in the industry.
References

[1] Doug Lea. *The JSR-133 Cookbook for Compiler Writers*
http://g.oswego.edu/dl/jmm/cookbook.html

[2] Doug Lea. *Synchronization and the Java Memory Model*
http://gee.cs.oswego.edu/dl/cpi/jmm.html

[3] Bill Venners. *Inside the Java Virtual Machine, Thread Synchronization*


http://docs.oracle.com/javase/6/docs/api/overview-summary.html