Combine safe publication and effective immutability to improve performance

Reduce synchronization costs in applications with infrequently modified mutable collections

Level: Intermediate

Andrew Citron (citron@us.ibm.com), Senior Programmer, IBM
Christopher R. Seekamp (seekamp@us.ibm.com), Programming Advisor, IBM
Martin Presler-Marshall (mpresler@us.ibm.com), Software Performance Analyst, IBM

02 Oct 2007

The typical way of enabling multiple threads to share access to a mutable collection — synchronizing on access to the collection — can become a performance bottleneck. Learn a technique you can use in Java™ 5.0 and later to minimize this bottleneck for data structures that are read frequently but updated infrequently.

A disadvantage to using data shared among multiple Java threads is that access to the data must be synchronized to avoid an inconsistent view of the contents, which could result in application failures. For example, the `Hashtable` class's `put()` and `get()` methods are synchronized. Synchronization is required so simultaneous `put()` and `get()` methods have sole access to the data when executing; otherwise, application data structures might get corrupted.

The synchronization points around those methods can become bottlenecks when an application's threads access the methods frequently enough that threads end up blocking. Only one thread at a time gets access to the contents. The other threads must wait their turn. Performance and throughput can suffer if threads do queue up when they could otherwise be doing useful work. In cases where performance analysis shows that the synchronized methods are in fact causing queuing points, optimizing the code could be worth the effort.

For data that changes infrequently, a technique called *generational data structures* lets you use the lower overhead of `volatile` to publish mutable data structures safely. When data structures are frequently accessed but infrequently modified, this can be a performance win. For example, you could use an unsynchronized data structure such as a `HashMap`, rather than a synchronized one such as a `Hashtable`. The key to the technique is to:

1. Make a new copy of the data structure when an update is made.
2. Fully populate it.
3. Safely publish the updates to all consumers using a `volatile` reference.

With this technique, `get` and `put` operations never execute at the same time on the same instance of the data structure. It ensures that two threads don't try to update the data structure at the same time and that reading threads always see a consistent, up-to-date version of the data. (The approach works even if data is updated frequently but the performance gains achieved by improved concurrency could be lost. Frequently repopulating the data structure could offset gains that are achieved by avoiding synchronized accessor methods.)

The technique exploits three characteristics of the Java language:

- **Automatic garbage collection.** When the last reference to an object has gone away, the Java runtime can free the object automatically. No action by the application is required other than making sure that no object references remain when the application is done using the object. Earlier generations are automatically freed when the last client is done with it.

- **Atomicity of object references.** A simple assignment statement that gets access to an object cannot be interrupted. This means that as long as the consuming thread can produce correct results with an older (but complete) copy of the object, it is not necessary to synchronize around a single-object assignment statement. However, it is important to note that steps must still be taken on the producer thread to guarantee that the creation of the new object completes prior to performing the assignment. As we explain in this article's Discussion section, synchronization is required on the producer thread to guarantee this completion prior to assignment. However, not having to use synchronization on the consumer threads is what removes an expensive queuing point.

- **The Java memory model.** The Java memory model specifies the semantics of `synchronized` and `volatile`. Those rules define when shared objects and their contents are visible to threads other than the currently executing thread.

Applicability to class pairs

`Hashtable` is one of a number of Java classes that provide access to data shared by multiple threads. `HashMap` is similar in function to `Hashtable` but it is not thread safe. The technique presented here is applicable to other pairs of classes that are similar to each other, except that one class has synchronized accessor methods and the other doesn't. For example, `Vector` has synchronized accessors, and `ArrayList` doesn't. Both provide a similar function and can use the approach described here.

You can take advantage of these Java language characteristics when data contained in a data structure does change by keeping two separate instances of the data structure. Once one is populated, it doesn't change again. It is *effectively immutable*. If the `get` and `put` operations were allowed to execute simultaneously on the same data structure it would be dangerous. The technique we describe here ensures that all `puts` complete before any `get` can execute.

The technique

The sample code in Listing 1 illustrates the technique:

Listing 1. Producer/consumer code that avoids queuing points

```
static volatile Map currentMap = new HashMap(); // this must be volatile to ensure
                                                // consumers will see updated values
static Object lockbox = new Object();

public static void buildNewMap() {              // This is called by the producer
                                                // when the data needs to be updated.

    synchronized (lockbox) {                    // This must be synchronized because
                                                // of the Java memory model.

      Map newMap = new HashMap(currentMap);     // For cases where new data is based on
                                                // the existing values, you can use the
                                                // currentMap as a starting point.
      // add or remove any new or changed items to the newMap
      newMap.put(....);
      newMap.put(....);

      currentMap = newMap;

   }
/* After the above synchronization block, everything that is in the HashMap is
   visible outside this thread.  The updated set of values is available to
   the consumer threads.

   As long as assignment operation can complete without being interrupted
   and is guaranteed to be written to shared memory and the consumer can
   live with the out of date information temporarily, this should work fine. */


}
public static Object getFromCurrentMap(Object key) {  // Called by consumer threads.


    Map m = currentMap;              // No locking around this is required.

    Object result = m.get(key);      // get on a HashMap is not synchronized.

    // Do any additional processing needed using the result.

    return(result);

}
```

Here's what happening in Listing 1:

- A second variable — called `newMap` in Listing 1 — holds the `HashMap` that is being populated with data. This variable, protected by a `synchronized` block, is used by just one thread at a time — a *producer* thread whose job is to:
    - Create a new `HashMap` and store it in the `newMap` variable.
    - Perform a complete set of `put` operations on `newMap` such that all data that is needed by the consumer threads are in `newMap`.
    - When `newMap` is completely populated, assign the value of the `newMap` to `currentMap`.

  The producer thread can be executed periodically, as a result of a timer, or it can be a listener that's awakened when some external data, such as a database, has changed.
- Consumer threads that need to consume the contents of `currentMap` simply access the object and perform `get` operations. Note that the `m = currentMap` assignment is a unit operation and does not need to be synchronized, even though other threads might be accessing the object's value. This is safe because `currentMap` is volatile and is populated inside the producer's synchronized block. That means that the contents of the data structure read through the `currentMap` reference will be at least as up-to-date as the `currentMap` reference itself.

Discussion

Once the `newMap` has been assigned to `currentMap`, the contents never change. Effectively, the `HashMap` is immutable. This allows multiple `get` operations to run in parallel, which can be a major performance boost. According to Brian Goetz in section 3.5.4 of *Java Concurrency in Practice* (see Resources), "safely published effectively immutable objects can be used without additional synchronization." The safe publication is a result of the `volatile` reference.

The only thing that might change while the data is being read is the object reference to the `currentMap` variable. The producer might overwrite the current value with a new value at the same time the consumer threads access the value. Because object references are unit operations in the Java language, the consumer does not need to synchronize when accessing that object. The worst that could happen is the consumer gets a reference to `currentMap`, then the producer overwrites that reference with newer contents. In that case, the consumer thread uses data that is slightly out of date but is still internally consistent. The same result would occur if the consumer thread had executed a second before the producer thread was ready to run. Typically, this should not cause any problems. The key is that `currentMap`'s contents are always fully self-consistent and immutable when they are published.

When this race does occur, the consumer threads could have a reference to the "old" version of the data. The "new" object reference has overwritten the old one, but some consumers still have a reference to the old one. When the last consumer finishes referencing the old object, the object goes out of scope and is eligible for garbage collection. The Java runtime keeps track of when that occurs. The application does not need to free the old object explicitly because it happens automatically.

A new version of `currentMap` might be created periodically based on the application's needs. By following the steps we've outlined above, you can ensure that those updates occur safely and repeatedly.

The `synchronized` block in Listing 1 is required to guarantee that the two producer threads don't race to update `currentMap` at the same time. That could cause data loss, which could lead to consumer threads seeing indeterminate results. The `synchronized` precludes the optimizer from making such decisions, essentially causing the entire map creation to be treated as an atomic operation. The `volatile` keyword guarantees that consumer threads do not continue to see an old value of the `currentMap` variable after it has been modified. Even more important, it guarantees that any values a client reaches by dereferencing through the object reference are at least as up-to-date as the reference itself. An ordinary reference would not provide this ordering guarantee.

The net effect of the use of a `synchronized` block and `volatile` keyword is to ensure that the consumer threads see a consistent view. The producer is aided by the fact that the data structure is not modified after publication. In this case — publishing an effectively immutable object graph — all that is required is to publish the root object reference safely. Note that you could also synchronize the consumer's access to the root reference, but that would be a possible queuing point, which is what this technique is trying to avoid. Brian Goetz refers to this approach as the "cheap read-write lock" trick (see Resources).

Conclusion

This article's technique is applicable to any situation where shared data changes infrequently and is accessed simultaneously by multiple threads of execution. It applies only to situations in which having the absolute latest data is *not* a requirement of the application.

The end result is concurrent access to shared data that can change over time. In environments where high concurrency is required, this technique lets you avoid having unnecessary queuing points within the application.

It's important to note that because of the intricacies of the Java memory model, the technique described here works only in Java 5.0 and later. In earlier Java versions, the client application is at risk of viewing an incompletely populated `HashMap`, or a corrupted, invalid, or inconsistent view of internal data structures of the `HashMap`.

Acknowledgment

The authors would like to thank Brian Goetz for his technical reviews and suggestions to make this article complete, precise, and accurate.

Share this...

Digg this story

Post to del.icio.us

Slashdot it!

Resources

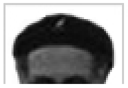
Learn

- "Double-checked locking: Clever, but broken" (Brian Goetz, JavaWorld.com, February 2001): Read about some synchronization gotchas.
- *Java Concurrency in Practice* (Brian, Goetz, Addison-Wesley, May 2006): Chapter 16 of *Java Concurrency in Practice* explains the Java memory model.
- "*Java theory and practice*: Managing volatility" (Brian Goetz, developerWorks, June 2007): Some patterns for using volatile variables correctly.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Hundreds of articles about every aspect of Java programming.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the authors



Andy Citron works in the WebSphere Portal performance group in Research Triangle Park, NC. His 30-year career with IBM has included stints creating products such as the Mwave Multimedia Card and its telephone-answering and call-discrimination subsystem, word processors, operating

systems, and wireless Internet access. In the late 1980s, Andy was lead architect for the SNA communication protocol known as APPC (or LU6.2). His work in the SNA architecture group led to a number of patents in the area of distributed two-phase commit processing.

Chris Seekamp is a programming consultant in the Workplace, Portal, and Collaboration Software division of the IBM Software Group. He has worked on various products, including Lotus Sametime, Lotus Connections, WebSphere Portal, and WebSphere Transcoding Publisher. He has been applying object-oriented design and development techniques for over 15 years, first in C++ and then in the Java language. He also has a strong interest in Linux and open source software.

Martin Presler-Marshall is a senior programmer at IBM's Research Triangle Park, NC site. He has been involved with Web-related software since 1995, starting as a developer on IBM's first HTTP server product. He is currently working as a performance expert in the WPLC performance team in IBM's Lotus division. His main focus areas there are improving the performance of IBM WebSphere Portal and IBM Lotus Quickr. He is co-author of several W3C recommendations and technical reports, such as the P3P specification. He has worked for IBM since 1991. When not working, he enjoys camping, bicycling, woodturning, and the martial art of tae kwon do.