Homework 3. Java shared memory performance races

Background

You're working for a startup company Ginormous Data Inc. (GDI) that specializes in finding patterns in large amounts of data. For example, a big retailer might give GDI all the web visits and purchases made and all the credit reports they've inspected and records of all the phone calls to them, and GDI will then find patterns in the data that suggest which toys will be hot this Christmas season. The programs that GDI writes are mostly written in Java. They aren't perfect; they're just heuristics, and they're operating on incomplete and sometimes-inaccurate information. They do need to be fast, though, as your clients are trying to find patterns faster than their competition can, and are willing to put up with a few errors even if the results aren't perfect, so long as they get good-enough results quickly.

The problem

GDI regularly uses multithreading to speed up its applications, and many of GDI's programs operate on shared-memory representations of the state of a simulation. These states are updated safely, using Java's `synchronized` keyword, and this is known to be a bottleneck in the code. Your boss asks you what will happen if you remove the `synchronized` keyword. You reply, "It'll break the simulations." She responds, "So what? If it's just a small amount of breakage, that might be good enough. Or maybe you can substitute some other synchronization strategy that's less heavyweight, and *that*'ll be good enough." She suggests that you look into this by measuring how often GDI's programs are likely to break if they switch to inadequate-but-faster synchronization methods.

In some sense this assignment is the reverse of what software engineers traditionally do with multithreaded applications. Traditionally, they are worried about race conditions and insert enough synchronization so that the races become impossible. Here, though, you're deliberately trying to add races to the code in order to speed it up, and want to measure whether (and ideally, how badly) things will break if you do.

It should be noted that we are on thin ice here, so thin that some would argue we've gone over the edge. That's OK: we're experimenting! For more about the overall topic, please see: Boehm H-J, Adve SV. You don't know jack about shared variables or memory models. *ACM Queue* 2011 Dec;9(12):40. doi:10.1145/2076796.2088916.

The Java memory model

Java synchronization is based on the Java memory model (JMM), which defines how an application can safely avoid data races when accessing shared memory. The JMM lets Java implementations optimize accesses by allowing more behaviors than the intuitive semantics where there is a global clock and actions by threads interleave in a schedule that assumes sequential consistency. On modern hardware, these intuitive semantics are often incorrect: for example, intraprocessor cache communication might be faster than memory, which means that a cached read can return a new value on one processor before an uncached read returns an old value on another. To allow this kind of optimization, first, the JMM says that two accesses to the same location *conflict* if they come from different threads, at least one is a write, and the location is not declared to be `volatile`; and second, the JMM says that behavior is well-defined to be data-race free (DRF) unless two conflicting accesses occur without synchronization in between.

The details for proving that a program is DRF can be tricky, as is optimizing a Java implementation with data-race freedom in mind. Not only have serious memory-synchronization bugs been found in Java implementations, occasionally bugs have been found in the JMM itself, and sometimes people have even announced bugs only to

find out later that they weren't bugs after all. For more details about this, please see: Lochbihler A. Making the Java memory model safe [PDF]. *ACM TOPLAS* 2013 Dec;35(4):12. doi:10.1145/2518191. You needn't read all this paper, just the first eight pages or so—through the end of §1.1.3.

How to break sequential consistency in Java

It's easy to write programs that break sequential consistency in Java. To model this, you will use a simple prototype that manages a data structure that represents an array of integers. Each integer is in the range [0,*maxval*] where *maxval* is at most 127, so the integer can be represented by the Java type `byte`. A state transition, called a *swap*, consists of subtracting 1 from one of the positive integers in the array, and adding 1 to an integer that is less than *maxval*. The sum of all the integers should therefore remain constant; if it varies, that indicates that one or more transitions weren't done correctly. Also, the values in the array should always be in range. The converse is not true: if the sum remains constant and the values remain in range it's still possible that some state transitions were done incorrectly. Still, these tests are reasonable ways to check for errors in the simulation.

For an example of a simulation, see jmm.jar, a JAR file containing the simplified source code of a simulation. It contains the following interfaces and classes:

`State`
: The API for a simulation state. The only way to change the state is to invoke `swap(i,j)`, where `i` and `j` are indexes into the array. If the `i`th entry in the array is positive and the `j`th entry is less than the maximum value allowed, the swap succeeds, subtracting 1 from the `i`th entry and adding 1 to the `j`th entry, returning `true`. Otherwise the swap fails and does nothing, returning `false`.

`Nullstate`
: An implementation of `State` that does nothing. Swapping has no effect. This is used for timing the scaffolding of the simulation.

`SynchronizedState`
: An implementation of `State` that uses the Synchronized class so that it is safe but slow.

`SwapTest`
: A `Runnable` class that tests a state implementation by performing a given number of successful swaps on it. It does not count failed swaps.

`UnsafeMemory`
: A test harness, with a `main` method. Invoke it via a shell command like "`java UnsafeMemory Synchronized 8 1000000 6 5 6 3 0 3`". Here, Synchronized means to test the `SynchronizedState` implementation; `8` means to divide the work into 8 threads of roughly equal size; `1000000` means to do a million successful swap transitions total; `6` is *maxval*, an integer in the range [0,127] as described above; and the remaining five numbers are the initial values for the five entries in the state array. The shell command outputs a string like "`Threads average 3318.01 ns/transition`", giving the approximate average number of real-time nanoseconds that it took a thread to do a successful swap, including all the overhead. It also outputs an error diagnostic if a reliability test fails.

Assignment

Build and use a sequential-consistency-violating performance and reliability testing program, along the lines described below.

- Your program should operate under Java 11. There is no need to run on older Java versions.
- Your program should compile cleanly, without any warnings.
- Please keep your implementation as simple and short as possible, for the benefit of the reader.
- Use the SEASnet GNU/Linux servers `lnxsrv0[679]` and `lnxsrv10`, with OpenJDK version 11.0.1 or later, to do your performance and reliability measurements. On SEASnet your `PATH` should be set to a string starting with "`/usr/local/cs/bin:`".

- Do not use more than 40 threads at a time, to avoid overloading the servers.
- Gather and report statistics about your testing platform, so that others can reproduce your results if they have similar hardware. See the output of `java -version`, and see the files `/proc/cpuinfo` and `/proc/meminfo`.
- Run the test harness on the `Null` and Synchronized classes, using various values for the number of threads, number of swap transitions, size of the state array, and sum of values in the state array, and characterize the performance of the two classes. Both classes should have 100% reliability, in the sense that they should pass all the tests (even though the `Null` class does not work); check this.

Do the following tasks and submit work that embodies your results.

1. Implement a new class Unsynchronized, which is implemented just like Synchronized except that it does not use the keyword `synchronized` in its implementation.
2. Implement a new class GetNSet, which is halfway between unsynchronized and synchronized, in that it does not use synchronized code, but instead uses volatile accesses to array elements. Implement it with the `get` and `set` methods of `java.util.concurrent.atomic.AtomicIntegerArray`.
3. Design and implement a new class BetterSafe of your choice, which achieves better performance than Synchronized while retaining 100% reliability.
4. Integrate all the classes into a single program `UnsafeMemory`, which you should be able to compile with the command `javac UnsafeMemory.java` and to run using the same sort of shell command as the test harness.
5. For each class Synchronized, Unsynchronized, GetNSet, and BetterSafe, measure and characterize the class's performance and reliability.
6. Compare the classes' reliability and performance to each other. Does any class seem to be the best choice for GDI's applications?

When considering how to implement BetterSafe, read Doug Lea's Using JDK 9 Memory Order Nodes. Also, you can look at the following packages and classes for implementation ideas:

1. `java.util.concurrent`
2. `java.util.concurrent.atomic`
3. `java.util.concurrent.locks`
4. `java.lang.invoke.VarHandle`

Write a report that contains the following explanations and discussions. Your report should be at least two and at most three pages long, using 10-point font in a two-column format on an 8½"×11" page, as suggested in the USENIX template mentioned in Resources for written reports and oral presentations.

- For each of the four packages and classes listed above, explain the pros and cons of using that package or class to implementing BetterSafe. Explain why you chose the packages and classes that you used.
- Explain why your BetterSafe implementation is faster than Synchronized and why it is still 100% reliable. Characterize your implementation's basic idea using terminology taken from Lea's paper.
- Discuss any problems you had to overcome to do your measurements properly. Explain whether and why the class is DRF; if it is not DRF (due to a bug) give a reliability test (as a shell command "`java UnsafeMemory` *`model`* `...`") that the class is extremely likely to fail on the SEASnet GNU/Linux servers.

Submit

Submit two files:

1. A JAR file `jmmplus.jar` containing your solution. Include your source code (`.java` files) not the `.class` files. It should contain a copy of the files in `jmm.jar`, possibly with modifications (though you should attempt to minimize these modifications). It should also contain the source code to your new classes. Please limit your source-code lines to 80 characters or less.

2. A PDF file `report.pdf` containing your explanations, discussions, and performance and reliability results.

Do not put your name or student-ID into your submissions.

© 2014–2018 Paul Eggert. See copying rules.
$Id: hw3.html,v 1.78 2018/11/29 18:11:03 eggert Exp $