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, and also by looking into a safe technique that is faster than `synchronized`.

In some sense the first part of 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 `long`s. Each array entry starts at zero. A state transition, called a *swap*, consists of subtracting 1 from one of the entries, and adding 1 to an entry – typically a different entry although the two entries can be the same in which case a swap does nothing. The sum of all the array entries should therefore remain zero; if it becomes nonzero, that indicates that one or more transitions weren't done correctly. The converse is not true: if the sum is zero it's still possible that some state transitions were done incorrectly. Still, this test is a reasonable way 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. This subtracts 1 from the `i`th entry and adds 1 to the `j`th entry.

`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 swap transitions on it.

`UnsafeMemory`
: A test harness, with a `main` method. Invoke it via a shell command like this:

```
time timeout 3600 java UnsafeMemory Synchronized 8 100000000 5
```

Here, the initial `time` tells the shell to give top-level real, user and system time. The `timeout 3600` puts a 3600-second timeout on Java, helpful if your program mistakenly loops forever. `Synchronized` means to test the `SynchronizedState` implementation; `8` means to divide the work into 8 threads of roughly equal size; `100000000` means to do 100 million swap transitions total; and `5` says to use a state array of 5 entries. The output of this command should look something like this:

```
Total time 4.71665 s real, 5.14257 s CPU
Average swap time 377.332 ns real, 51.4257 ns CPU

real    0m4.843s
user    0m5.248s
sys     0m0.060s
```

The output says that from Java's point of view the entire test took 4.71665 s in real time and 5.14257 s of CPU time (the latter can be greater than the former in multithreaded programs). It also says that a single swap by a single thread took an average of 377.332 ns of real time and 51.4257 ns of CPU time (the former should always be greater than the latter). From the Linux kernel's point of view, the entire Java program consumed 4.843 s real time, 5.248 s user CPU time, and 0.060 s system CPU time.

Assignment

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

- Your program should operate under Java 13 or later. There is no need to run on older Java versions.
- Your program should compile cleanly, without any warnings, simply by using the shell command '`javac *.java`'.
- Please keep your implementation as simple and short as possible, for the benefit of the reader.
- Use two of the SEASnet GNU/Linux servers `lnxsrv0[679]` and `lnxsrv10`, with Java version 13.0.2, to do your performance and reliability measurements. On SEASnet if your `PATH` starts with "`/usr/local/cs/bin`:" you will run with version 13.0.2. Choose two servers with differing types of CPUs (look for "model name" in `/proc/cpuinfo`).
- Do not use more than 40 threads at a time, to avoid overloading the servers.
- Gather and report statistics about your two testing platforms, 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, on each of the two servers you chose, along with various values for the size of the state array (use 5, 100, and at least one other value), the number of threads (use 1, 8, 40 and at least one other value) and roughly characterize the performance of the two classes. Use enough swap transitions so that your results are dominated by the actual work instead of by startup overhead. `Null` and `Synchronized` 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 `UnsynchronizedState`, which is implemented just like `SynchronizedState` except that it does not use the keyword `synchronized` in its implementation. Put the implementation into a new file `UnsynchronizedState.java`.
2. Design and implement a new class `AcmeSafeState` of that is safe without using the `synchronized` keyword. Your imlementation should use `java.util.concurrent.atomic.AtomicLongArray`. The goal is to achieve better performance than `SynchronizedState` while retaining safety. Put the implementation into a new file `AcmeSafeState.java`.
3. Integrate all the classes into a single program `UnsafeMemory`, which you should be able to compile with the command '`javac *.java`' and to run using the same sort of shell command as the test harness. Your version of `UnsafeMemory.java` will need to differ slightly from the one distributed as part of this assignment
4. For each class `SynchronizedState`, `UnsynchronizedState`, and `AcmeSafeState`, measure and characterize the class's performance and reliability as described above (that is, use two servers, vary the state array size, and vary the number of threads). Attempt to find areas where these differ.
5. Compare the classes' reliability and performance to each other. For each class, say what kind of benchmarks (if any) the class seems to do best and worst on.

To help think about your `AcmeSafeState` implementation, read Doug Lea's Using JDK 9 Memory Order Nodes, and you can look at the following packages and classes for other facts and 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 five 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.

- Characterize your `AcmeSafeState` implementation's basic idea using terminology taken from Lea's paper. In particular, explain whether and why your `AcmeSafeState` class is DRF.
- Discuss any problems you had to overcome to do your measurements properly.
- Give your measurements and analysis of the measurements.

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.

$Id: hw3.html,v 1.86 2020/01/30 03:36:08 eggert Exp $