A Comparison of Synchronization Techniques on Thread Performance and Safety in Java

Anderson Huang, UCLA

Abstract

A perpetual battle exists in the field of computer science between the benefits of parallelism and the goal of program predictability. In other words, there is an inverse relationship between performance and correctness. In Java, there is a memory model that dictates how applications should run safely should synchronization techniques be used in a multithreaded program. The dangers of multithreaded are compounded by the effects of out-of-order processing done by compilers, whose first order of business is performance, and caching, a technique with the same as the former. Thus, in many programming languages that include not just Java, many synchronization mechanisms exist to mediate between performance and correctness. Specifically in Java, keywords like synchronized and volatile as well classes designed specifically to address thread-safety are used enforce synchronization. In this paper, we report the findings and results of using different synchronization mechanisms on simple Java module that performs increment and decrement operations on a small array; afterwards, we discuss the advantages and disadvantages of each technique.

1. Introduction

Race conditions are the bane of any multithreaded program. In the following discussion, we explore what occurs when multithreaded programs use no synchronization methods and the converse situation. Specific synchronization techniques, in Java, that will be examined are methods protected by the *synchronized* keyword, locking mechanisms built into the *java.util.concurrent.locks* package, and atomic classes in the *java.util.concurrent.atomic* package.

2. Materials and Methods

All measurements were made on the UCLA SEASNet GNU/Linux servers. Java version 1.8.0_112 was used along with the following metrics about CPU and memory info. To get similar information, output the contents of /proc/cpuinfo and /proc/meminfo.

```
processor       : 29
vendor_id       : GenuineIntel
cpu family      : 6
model           : 62
model name      : Intel(R) Xeon(R) CPU E5-2640 v2
@ 2.00GHz
stepping : 4
microcode       : 0x428
cpu MHz                 : 1842.812
cache size      : 20480 KB
physical id     : 1
siblings  : 16
core id         : 6
cpu cores       : 8
apicid          : 45
initial apicid  : 45
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
MemTotal:      65758692 kB
MemFree:       60096800 kB
MemAvailable:   64032656 kB
Buffers:         197968 kB
Cached:         3290264 kB
SwapCached:            0 kB
Active:         2672808 kB
```

2.1 Methods

To collect metrics, a simple module that operates on a byte-array was used. A variable number of elements exist in the array, and in any given execution of the program a certain number of swaps occur. Swaps are defined as having one element be decremented and a corresponding element that is different be incremented. For an execution to be defined as correct, no elements should end with values less than zero or greater than an arbitrary maximum value, up to 127 and defined by the tester. Secondly, the sum of the elements in the array should be consistent. After all, decrementing and incrementing two elements by 1, respectively, should not change the overall sum.

To test, we used five different versions to model the swapping: Unsynchronized, GetNSet, Synchronized, BetterSafe, and BetterSorry.

The Unsynchronized module has no safeguards on the swap operation, and is fully open to data-race conditions. The GetNSet module utilizes an AtomicIntegerArray to enforce atomic operations; however, synchronization is still not guaranteed. The Synchronized module uses the *synchronized* keyword in Java to segment off a critical section in the code. The BetterSafe module uses another locking mechanism apparent in Java. Finally, the BetterSorry module is an improvement on the GetNSet module.

3. Results

	Module				
Num. of Swaps	Unsync	GetN Set	Sync	Better Safe	Better Sorry
10000	8046	1126 0	9331	10416	9942
100000	Hang	Hang	5881	3733	2998
1000000	Hang	Hang	2904	1126	1492

Figure 1. Average thread time, including overhead, as a function of different swap modules and number of swaps. All times are reported in nanoseconds. Numbers are rounded to the nearest integer for simplicity.

All measurements were made with an execution with eight threads, a five-element array, and a maximum value of ten. The two variables that were monitored were the number of swaps and the type of module running the swap operation.

3.1. Analysis

From Figure 1, it is clear that the BetterSafe module outperforms the Synchronized module, while still remaining 100% reliable. The reason behind this difference is that the *synchronized* keyword in Java can be thought to insert lock and unlock statements in the code. The lock that the synchronized keyword uses is a spin-lock, which “busy-waits”. Whenever a resource is locked and a thread needs it, it will continuously poll until the resource is free. This is very costly in terms of CPU time. Compare this with the BetterSafe module, which uses a reentrant lock. If resources are busy, the thread in question will block until the resource is free, thus saving CPU time.

The BetterSorry is also somewhat faster than the BetterSafe method until input becomes sufficiently large. This performance difference can be attributed to not having to lock when a swap operation occurs, which is a very costly process. However, BetterSorry still improves on reliability by atomically incrementing and decrementing values. Contrast this with Unsynchronized. Consider the following snippet of code.

```
x = x + 1;
value[i] = x;
```

The first operation becomes an load, add and a store. If the entire sequence is not done atomically, then two threads can potentially cause data races. For example, one thread may start by loading the value of x. Another thread may jump in, finish its sequence of operations and increment x, but the first thread will now use an incorrect, stale value for x for its sequence of operations. BetterSorry still suffers from a possible race condition between the check for bounds and the first decrement of one element However, a test case exhibiting this failure is hard to derive because the race condition does not manifest itself in every execution.

Measurements were hard to obtain for Unsynchronized and GetNSet because many configurations hung forever because race conditions pushed elements out of bounds. To summarize, the Synchronized and BetterSafe implementations are data-race free(DRF) because they lock critical sections of code. Unsynchronized is not data-race free because an assignment operation is not atomic under normal circumstances. GetNSet and BetterSorry are similarly not DRF due to similar arguments where parts of the critical section can be executed on simultaneously by multiple threads. A test case both Unsynchronized and GetNSet fail is

```
java UnsafeMemory <module> 8 100000 10 5 6 5 0 3
```

BetterSafe is the best module across the board, but Synchronized provides a good alternative, and BetterSorry is an alternative when correct results aren’t the only motive.

4. Conclusion

Synchronization is a hotly-debated issue and will be for years to come. The ideal module after discussing the tradeoffs in this article is a toss-up between BetterSafe and BetterSorry. If there are less swaps, and performance is of utmost importance, choose BetterSorry. If input is big enough, however, BetterSafe outperforms.