Java Concurrency Experiments Report

Xiangyu Wan
UCLA

1 AcmeSafeState Implementation Explained

The AcmeSafeState class I implemented uses AtomicLongArray class provided in java.util.concurrent.atomic package and comes from modifying SynchronizedState. Changes include declaring what used to be a long[] member as a AtomicLongArray, and changing all array getting and setting statements to corresponding AtomicLongArray methods. These atomic access methods include AtomicLongArray.length(), to get length, AtomicLongArray.get(), to get a certain value by subscript, and AtomicLongArray.getAndIncrement()/getAndDecrement() which takes the element by subscript and increment/decrements it. Then the Synchronized token is taken off from swap() method.

The main difference between AcmeSafeState and SynchronizedState is that, one uses the synchronized functionality provided by JVM, while the other uses Atomic operations. As defined in the Lea paper, a block marked with "synchronized" by default uses builtin locks to enforce execution order, which is the strongest order mode.

On the other hand, AcmeSafeState uses atomic operations, which is implementation of the Volatile mode as mentioned in Lea's paper. In an AtomicLongArray, elements must be updated atomically, as mentioned in its own documentation, meaning that each operation is totally independent from another, they cannot interrupt each other. Therefore, it's volatile mode because atomic operations form a total order: any 2 atomic operation cannot have the same level of precedence.

By taking this approach, array operations are forced to be atomic when they operate on the same element, and this total ordering eliminates potential race conditions as demonstrated by UnsynchronizedState. The resulting AcmeSafeState is therefore a data-race-free class.

Also, according to Lea, Volatile mode is a weaker mode compared with Lock mode. This could explain the overall better performance of AcmeSafeState than SynchronizedState, which will be discussed in later sections.

2 Problems Encountered

Since each test has to be done on both of 2 servers choosing from SEASnet servers, the first problem I encountered was on how to gather system information. Much of the information provided in /proc/cpuinfo and /proc/meminfo are too detailed for this project. Eventually I decided to include number of processors, by counting lines starting with "processor" in /proc/cpuinfo using grep and wc commands, and total memory, by grep the line starting with "MemTotal" in /proc/meminfo, with a shell script. The next problem was limited resource. For some reason, as I later figured out, server 10 only allows me to use 4 processors, so tests running 40 threads are unable to proceed. Moving to server 07 solved it, though I initially thought this was caused by overwhelmed server capacity.

3 Measurements

Each item in the table is total real time, in seconds, reported by each of the 96 test harnesses.

1. On lnxsrv07:

   (a) Synchronized

Threads/Size	5	100	114514
1	2.40645	2.30248	2.44090
8	41.8353	46.7085	59.6889
30	44.2645	50.3885	58.5530
40	58.0985	49.1583	57.5134

   (b) Null

Threads/Size	5	100	114514
1	1.67645	1.48559	1.24514
8	0.456103	0.485669	0.578254
30	0.342519	0.414552	0.504577
40	0.463978	0.528948	0.447990

   (c) Unsynchronized

Threads/Size	5	100	114514
1	1.74454	1.65491	1.64390
8	5.05006!	4.96911!	0.890892!
30	2.93315!	3.39934!	0.731315!
40	2.84386!	3.06318!	0.841629!

(d) AcmeSafe

Threads/Size	5	100	114514
1	2.79151	2.90863	4.08026
8	15.5459	4.11677	1.76787
30	10.8118	5.53087	0.926240
40	8.49417	3.42478	0.835697

2. On lnxsrv09:

   (a) Synchronized

Threads/Size	5	100	114514
1	2.09573	2.06913	2.36986
8	23.4730	20.0931	25.2516
30	23.9477	27.1043	32.8053
40	25.0529	25.5896	32.7916

   (b) Null

Threads/Size	5	100	114514
1	1.39438	1.34573	1.33739
8	0.269665	0.297212	0.297726
30	0.239610	0.301251	0.289336
40	0.485659	0.420304	0.547376

   (c) Unsynchronized

Threads/Size	5	100	114514
1	1.54024	1.51304	1.92184
8	2.47085!	4.25720!	0.848775!
30	2.76206!	3.00428!	0.739111!
40	2.84741!	3.09914!	0.818826!

   (d) AcmeSafe

Threads/Size	5	100	114514
1	2.64326	2.63030	4.00808
8	5.47784	7.53352	1.70335
30	5.45605	4.82547	0.938191
40	9.01959	4.45131	0.832072

There are similarities shared on tests on both servers. First of all, Null always spends the least time.

References