Computer Science 131, Homework 3

May 7th, 2018

1 Testing Environment

I performed this assignment on `lnxsrv09`, which has the following output when I ran `java --version`.

```
java 10.0.1 2018-04-17
Java(TM) SE Runtime Environment 18.3
(build 10.0.1+10)
Java HotSpot(TM) 64-Bit Server VM 18.3
(build 10.0.1+10, mixed mode)
```

The file `/proc/cpuinfo` shows that the server has 32 processors with the model shown below.

```
Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
```

The file `/proc/meminfo` shows that the server has 65.7 GB of memory.

2 Testing of Various State Implementations

I ran the `UnsafeMemory` program using an array of about 100 elements and a maximum value of 127 for the given combinations of threads, iterations, and synchronization methods as shown in the table below. The table shows its results in nanoseconds per swap. I did not include `Unsynchronized`, since most of the executions would send it into an infinite loop due to every element in the array being too high or low. It had the worst reliability, since any multi-threading would result in it not working. Because it is so unreliable, it is impossible to assess its performance.

The remaining classes `Synchronized`, `GetNSet`, and `BetterSafe` were completely reliable. They always finished correctly. As the number of threads and iterations increase, it seems that `BetterSafe` has the best performance, followed by `GetNSet`, then `Synchronized`. In test cases with fewer iterations, the overhead of thread creation is too high, which is why every synchronization method takes more time per swap.

Based on the test results shown above, it seems that `BetterSafe` is the best option for GDI, as it scales the best as the number of swaps increases. Since GDI will be processing a lot of data with many threads, it makes the most sense to use the synchronization method that scales the best.

3 Implementation Details

When creating the different synchronization methods, I implemented `Unsynchronized` by simply taking the code from `Synchronized` and removing the synchronized keyword. Then I implemented `GetNSet` with an atomic integer array, as specified. I made sure to check after atomically incrementing and decrementing the values in the array if they were out of bounds. I reversed the increment and decrement in that case, making `GetNSet` reliable.

I looked at each of the suggested packages `java.util.concurrent`, `java.util.concurrent.atomic`, `java.util.concurrent.locks`, and `java.lang.invoke.VarHandle` when trying to implement `BetterSafe`. The package `java.util.concurrent` defines a semaphore, countdown latch, and other concurrent programming structures, but none of the data structures were well suited for atomically incrementing and decrementing two different values in an array. The semaphore or

Threads	Iterations	Null	Synchronized	GetNSet	BetterSafe
2	100000	577.772	532.991	834.128	950.566
2	1000000	114.178	246.946	336.413	538.836
2	10000000	21.7093	411.582	280.731	240.006
4	100000	899.194	2350.86	1147.13	2053.05
4	1000000	374.75	1090.33	583.02	510.414
4	10000000	51.5071	990.217	573.922	313.693
6	100000	1236.72	3551.79	1703.41	3114.63
6	1000000	1240.69	1897.01	960.894	768.654
6	10000000	86.8635	1476.9	766.529	482.331
8	100000	1637.84	5014.69	2137.79	4417.18
8	1000000	1998.76	2385.36	1631.53	1066.93
8	10000000	132.914	1813.62	1133.65	620.82

latch could be used to create a lock, but this would be a spin lock. So I chose not to use this package because it would not scale well, and I could achieve better performance with the other packages.

In `java.util.concurrent.atomic`, I saw that there were some atomic classes that allowed for integer manipulation. The best choice from this package to use for `BetterSafe` was probably an atomic integer array, but I had already used that in `GetNSet`. My implementation of `GetNSet` greedily tries to perform the swap, then checks that the swap is correct. It undoes the swap and reports a failure if it is not correct. As contention increases, using an atomic integer array like this may result in a large number of swaps being incorrect, slowing down execution. So I wanted to see if I could find a better way to implement the `BetterSafe` than this.

While testing, I found that I could simply use a `ReentrantLock` to synchronize my code in `BetterSafe`. This class is a part of `java.util.concurrent.locks`, and it fit my needs very well. It outperformed the synchronized keyword by a significant amount as the number of iterations and threads increased. This is due to it having less overhead than the synchronized keyword. But it results in the same memory consistency effects as the synchronized keyword, ensuring reliability.

The last option for synchronization, `java.lang.invoke.VarHandle` is a class that defines different access modes. For our purposes, we only are concerned with volatile accesses, which can be used to guarantee that different threads agree on the synchronization order. The Opaque and Release/Acquire modes are not strong enough for the synchronization we need. As this is only a single class, this cannot easily be used to implement `BetterSafe`. In the end, `ReentrantLock` was the best option to implement `BetterSafe`, as it resulted in the best performance while being completely reliable.

My implementation of `BetterSafe` is faster than `Synchronized` because the designers of `ReentrantLock` were able to improve the efficiency of locking while still maintaining a consistent memory state. The lock ensures a total ordering between every operation in all the threads. In terms of implementation, the use of `ReentrantLock` does not differ significantly from `Synchronized`. The main idea in both approaches is to surround the critical section of code with a lock so that only one thread can change shared memory at once. It also guarantees that the changes to shared memory in a given thread is visible to other threads.

4 Testing Methodology

While designing my test cases, I had to select a good combination of threads, iterations, and array parameters for each class that I wanted to test. I found that

if I used more than 10 or so threads, the server would sometimes complain that no more threads could be created. This is why I chose to test up to 8 threads. I only tested up until 10 million swaps because if I used any more swaps the test cases would take too long to complete. I also decided to use an array with a lot of entries and a high maximum value because this would mimic the intended application of this program. GDI wants to use this on a large amount of data, so it makes sense to test with a large array.

`BetterSafe` is data race free, as the use of `ReentrantLock` guarantees this. The only class with data races is `Unsynchronized` which is likely to fail in a test case such as

```
java UnsafeMemory Unsynchronized \
8 10000000 127 43 34 56 34
```

Additionally, the class `GetNSet` is data race free, as every access in swap is an atomic operation. So no two threads can access the same memory location at once, though there is still a race condition because of the check that the swap did not change any array element to be out of bounds.