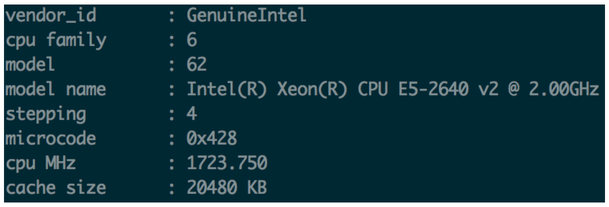
Comparison of Java Thread Synchronization Methods

1. Introduction

During this assignment, multiple Java synchronization methods were used in order to allow for multiple threads to manipulate a single object. With no synchronization, the results became very inaccurate, and race conditions often caused the program to become unresponsive. On the other hand, using Java's "synchronized" keyword allowed for 100% accuracy, but caused the program to run slowly. The goal was to test different methods that would allow for varying levels of speed and reliability.

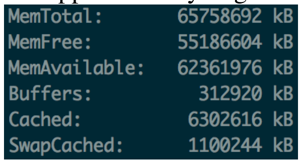
2. Testing Platform

In order to test the results of the program, classes were compiled and run with Java version 1.8.0_112. By viewing the file "/proc/cpuinfo", I found that the system I was using has 32 8-core 2GHz Intel Xeon E5-2640 v2 CPUs.



```
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         : 1723.750
cache size      : 20480 KB
```

By viewing "/proc/meminfo", I found that the system has approximately 65gB of memory.



```
MemTotal:       65758692 kB
MemFree:        55186604 kB
MemAvailable:   62361976 kB
Buffers:          312920 kB
Cached:          6302616 kB
SwapCached:      1100244 kB
```

Although these resources were usually adequate for testing purposes, the results were seen to fluctuate based on the amount of resources that were being used by other users on the system.

In order to test each of the classes that are explained below, the jar file "jmmplus.jar" can be unpacked using the command "jar xf jmmplus.jar". The java files can then be compiled by running "javac *.java", and the main method within the UnsafeMemory class can be invoked by using a command such as
"java UnsafeMemory GetNSet 8 1000 6 5 6 3 0 3".

In this example command, "GetNSet" means that the GetNSetState class will be utilized, "8" is the number of threads that will be used, "1000" is the number of swaps that will be performed, "6" is the maximum value of the array, and the remaining five numbers are the initial values for the array that will be used in the program.

3. NullState

NullState is a provided java class that does nothing for each swap. This is used as a dummy implementation for the purposes of timing the scaffold. After performing many tests by varying each program parameter, it was seen that it consistently performed quickly with 100% reliability, as expected. Since the implementation always prevents conflicting accesses of the critical section, the class is said to be DRF (data-race free).

4. SynchronizedState

Within the provided SynchronizedState class, Java's "synchronized" keyword is used to prevent multiple threads from executing code in the critical section at the same time. This class is DRF since it always prevents conflicting accesses of the critical section, and therefore provides 100% reliability. However, this synchronization is provided at the cost of performance. After running many tests on this class, SynchronizedState was seen to be slower than nearly every other class implementation.

5. UnsynchronizedState

The UnsynchronizedState class was created by simply removing the "synchronized" keyword from the SynchronizedState class. This had the effect of allowing multiple threads to access the critical section at the same time, which predictably lead to a decrease in reliability. In terms of performance, the lack of control over threads often lead to infinite loops due to swaps continuously returning "false". When the program doesn't get stuck, this class usually performs faster than SynchronizedState. However, the constant infinite loops causes the program to become unusable with large numbers of threads and swaps.

Due to the conflicts between threads, the class is not DRF. One particular command that is likely to demonstrate this is "java UnsafeMemory Unsynchronized 32 10000 100 5 6 3 0 3", which usually leads to large numbers of mismatches.

6. GetNSetState

The GetNSetState class uses an AtomicIntegerArray object in order to provided greater performance than SynchronizedState, and better reliability than UnsynchronizedState. By allowing for elements to be manipulated atomically, threads were prevented from accessing or updating elements at the same time. This led to a decent amount of reliability. However, the implementation is not DRF since it is possible for threads to be preempted after getting certain values, and before updating them in the array. One sample command that usually demonstrates this error is “java UnsafeMemory GetNSet 8 1000 6 5 6 3 0 3”.

After running many tests, GetNSetState was found to be consistently faster than SynchronizedState since finely-grained locking was used instead of locking the entire array. However, occasional deadlocks caused extremely poor performance.

7. BetterSafeState

The BetterSafeState class was created to provide better performance than SynchronizedState, while maintaining 100% reliability. In order to accomplish this, the ReentrantLock class was used in order to manually lock and unlock portions of the critical section. The reason for this is that ReentrantLock provides certain features that are more beneficial than the “synchronized” keyword in the case of large numbers of threads.

By locking the entire critical section, the implementation was made to be DRF, and provide 100% reliability. Based on the tests that were run, BetterSafeState was seen to reliably run faster the SynchronizedState in most cases.

8. BetterSorryState

The goal of the BetterSorryState class was to provide better performance than BetterSafeState, and better reliability than UnsynchronizedState. In order to achieve this, an array of AtomicInteger objects was created in order to allow for precise control over the values that were being read or updated. Similarly to in GetNSet, this finely-grained locking caused the implementation to be quite fast when compared to SynchronizedState. However, this also made the implementation not DRF, since it is possible for threads to be preempted after checking the edge cases, but before updating the AtomicIntegers in the array. It was difficult to find a command that would consistently show this error due to the fact that the case was extremely rare.

In terms of performance, BetterSorryState consistently performed faster than BetterSafeState, and usually performed better than GetNSet. This is due to the fact that in BetterSafeState, the class strictly locks the critical section so that only 1 thread can enter at a time, while in BetterSorryState, the class controls access to individual portions of the array.

9. Test Results

Some of the averages of test results that were obtained during the assignment are displayed in the table below. For most of these cases, 10 million swaps were used in order to stimulate the large amounts of computations that would be done on data for GDI.

Model	Average time per transition (ns)			DRF?
	Threads: 32	Threads: 16	Threads: 8	
Null	1575.59	475.081	140.97	Yes
Synchronized	13525.8	6362.09	2106.07	Yes
Unsynchronized	1606.07	787.06	308.034	No
GetNSet	1954.41	1037.63	625.973	No
BetterSafe	5191.71	2282.77	1131.31	Yes
BetterSorry	1698.02	841.602	368.461	No

10. Conclusion

Based on the tests that were run (some of which are displayed in section 9), the BetterSorryState class would likely be the best choice for the stated task for GDI. This is due to the fact that it was seen to be more reliable than the Unsynchronized class, and is deadlock-free. However, if the number of errors becomes unacceptable, the BetterSafeState class would be a good choice since it provides 100% reliability at the cost of performance.