Under Java 9 Version on server 06

Class	Thread	Number of swaps	Max value	Sum of values	Array Size	Average ns/transition	Reliability
Null	4	100000	50	165	5	823.661	100%
Synchronized	4	100000	50	165	5	2377.96	100%
Unsynchronized	4	100000	50	165	5	1301.18	0%
GetNSet	4	100000	50	165	5	5306.26	0%
BetterSafe	4	100000	50	165	5	1914.43	100%

Class	Thread	Number of swaps	Max value	Sum of values	Array Size	Average ns/transition	Reliability
Null	8	100000000	6	22	7	29.8633	100%
Synchronized	8	100000000	6	22	7	2475.22	100%
Unsynchronized	8	100000000	6	22	7	Forever to finish	None
GetNSet	8	100000000	6	22	7	Forever to finish	None
BetterSafe	8	100000000	6	2	7	878.674	100%

Class	Thread	Number of swaps	Max value	Sum of values	Array Size	Average ns/transition	Reliability
Null	16	100000000	6	22	7	48.0247	100%
Synchronized	16	100000000	6	22	7	3997.38	100%
Unsynchronized	16	100000000	6	22	7	Forever to finish	None
GetNSet	16	100000000	6	22	7	Forever to finish	None
BetterSafe	16	100000000	6	2	7	1568.29	100%

Under Java 11 Version on server 06

Class	Thread	Number of swaps	Max value	Sum of values	Array Size	Average ns/transition	Reliability
Null	8	100000000	6	26	7	15.4858	100%

Synchronized	8	100000000	6	26	7	2339.22	100%
Unsynchronized	8	100000000	6	26	7	Forever to finish	None
GetNSet	8	100000000	6	26	7	Forever to finish	None
BetterSafe	8	100000000	6	26	7	695.477	100%

Class	Thread	Number of swaps	Max value	Sum of values	Array Size	Average ns/transition	Reliability
Null	16	100000000	33	80	8	15.4858	100%
Synchronized	16	100000000	33	80	8	3510.56	100%
Unsynchronized	16	100000000	33	80	8	Forever to finish	None
GetNSet	16	100000000	33	80	8	Forever to finish	None
BetterSafe	16	100000000	33	80	8	983.748	100%

Whether BetterSafe is faster than Synchronized and why is it still 100% reliable?

As the data shown above, BetterSafe is faster than Synchronized on Java version 9 and 11, especially when the number of swap transitions are larger; and the number of threads increases. Moreover, BetterSafe still remains 100% reliability because I implement it using ReentrantLock. Like synchronized block, which uses monitor locks that are bound to objects, ReentrantLock also uses these locks but explicitly and with extended capabilities. ReentrantLock is able to maintain total ordering of locked region by simply using the lock() and unlock() operations to guard the critical section. Therefore, ReentrantLock guarantees that only one thread can go into the critical section one at a time, and hence remains 100% reliability. In addition, since BetterSafe needs to handle read, write and atomic updates, I need ReentrantLock to help me handle these memory order modes and still maintain reasonable ordering relations. Also, BetterSafe would be more ideal for GDI since BetterSafe achieves better performance than Synchronized does since GDI particularly specializes in finding patterns in large amounts of data.

	java.util.concurrent	java.util.concurrent.atomic	java.util.concurrent.locks	java.lang.invoke.varHandle
Pros	1. This package helps us manage the details of thread at the low level. 2. It has Queues to prevent race condition. 3. It contains a set of classes that makes it	1. This package contains classes that are easy to implement and use. 2. It supports lock-free thread-safe programming on single variables and hence is faster than other approaches using locks. 3. It has many atomic data	1. This package contains a ReentrantLock class that is easy to use and provides more extensive locking operations. We only need two operation to guard the critical section: lock() and unlock(). This efficiently prevent race condition	1. This package is a strongly typed reference to certain variables and supports various access mode to them (read, write and atomic update, etc). 2. It also supports primitive type like byte, which is exactly what we need in BetterSafe.

	easier to implement multithreading with less effort.	structures where several volatile fields of the same object are independently subject to atomic update.	since only the thread with the lock can enter the critical section while other threads must wait until that thread finishes.	
Cons	Although it has five classes of aid common special-purpose synchronizer, they are mostly not ideal for implementing BetterSafe. Compared to our simple task, these tools are relatively complicated to use.	In BetterSafe, we are dealing with byte data type, while this package only supports AtomicIntegerArray and other types. It's inconvenient to convert each array element from byte to int and then from int to byte. Moreover, its main use is for atomic update while our program requires much more than that: reading, writing. To achieve 100% reliability, we need locking.	However, since other threads must wait until the thread with the lock finishes. It will lead to lower performance and overall throughput.	However, access modes will override any memory ordering effects specified at the declaration site of a variable. For example, a VarHandle accessing a field using the get access mode will access the field as specified by its access mode even if that field is declared volatile. When mixed access is performed, it may produce surprising results.
Conclusion	Therefore, java.util.concurrent.locks is the ideal candidate to use among the above packages since we want to achieve 100% reliability and better performance than Synchronized block simultaneously; its ReentrantLock provides flexible operations for us to guard the critical section.			

Challenge

The problem I had to overcome is that for each iteration, the average ns/transition fluctuates and I couldn't decide which one to use so I do many iterations and calculate their average. Also, on different server, with the same input data, the result of average ns/transition is different, so I stick to the same server and finish these tests.

DRF Analysis

Synchronized is DRF because when one thread enters the critical section, other threads trying to enter the section will be blocked outside. Hence it's impossible to lead to a mismatch.

Unsynchronized is not DRF because it does not have any property or operation to prevent race condition. Therefore, even a small amount of swap transitions would be likely to lead to a mismatch. For example: java UnSafeMemory Unsynchronized 8 1000000 6 5 6 3 0 3

Null is DRF since it does not do anything at all, meaning "no swap, no mismatch."

GetNSet is not DRF since it only has get(read) and set(write) methods that prevent multiple threads trying to read or write the same object but it does not have any protection from the interruption between the get and set accesses. Therefore, if the number of swaps is large enough, the interruption between get and set is likely to occur. For example: java UnSafeMemory GetNSet 8 1000000 6 5 6 3 0 3

BetterSafe is DRF because it guards the critical section whenever a thread is in it already, meaning only one thread can access the critical section one at a time and hence achieves 100% reliability, so a mismatch is impossible.

Testing Platform Spec

The server is running on Intel Xeon 4 Cores CPU E5620 @2.40Ghz and has about 65 GB memory.

References

Doug Lea. Using JDK 9 Memory Order Modes. http://gee.cs.oswego.edu/dl/html/j9mm.html

Java Platform, Standard Edition & Java Development Kit Version 11 API Specification
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/package-summary.html