What to do about sun.misc.Unsafe

15 June 2015

Contributors:
Greg Luck, Hazelcast, EC Representative
Chris Engelbert, Hazelcast, EC Representative
Martijn Verburg, LJC, jClarity, Java Champion, EC Representative
Ben Evans, LJC, jClarity, Java Champion, EC Representative
Gil Tene, Azul Systems EC representative
Peter Lawrey, Higher Frequency Trading, Java Champion
Rafael Winterhalter, Bouvet ASA

Status: Draft

Contents

README FIRST

This is a document that is a work in progress. Edit history etc should be taken with a strong grain of salt. A version number can be stamped on this when it's deemed ready to present to the correct OpenJDK mailing list(s).

Summary

1. `sun.misc.Unsafe` is an **unsupported API** which has popular usage in the industry but it is **not an official standard** for Java SE. There are plans for its removal under the modularisation efforts for Java SE 9. This is deemed to be a good thing in general.
2. Many organisations/framework/library/product owners have stated they will be unable to move to Java SE 9 without some sort of replacement for some of the 'safe' `sun.misc.Unsafe` features.
3. The community at large should work on and propose JEPs through OpenJDK to provide the same functionality as the 'safe' parts of of `sun.misc.Unsafe`. It is generally acknowledged that there will not be enough time for this before Java SE 9 is feature complete (~Nov 2015).
4. As the JEPs and other pieces of work that make up Java SE 9 will be feature complete in ~Nov 2015, we would like to propose to allow `sun.misc.Unsafe` to be deprecated and/or be enabled by a `-XX` flag in order to allow new alternatives for the 'safe' parts of `sun.misc.Unsafe` to be proposed through the JEP process.

Current Problems

Widespread Community use of sun.misc.Unsafe - a proprietary API

`sun.misc.Unsafe` has wide traction in common Java frameworks and applications. Most applications, at least indirectly, depend on some library that uses Unsafe to speed up one thing or another.

In fact, even standard libraries such as `java.util.concurrent` depend upon pieces of Unsafe (such as park and CAS operations) for which there is no realistic alternative.

Over time, Unsafe has becoming a “dumping ground” for non-standard, yet necessary, methods for the platform, with useful methods that are relatively safe in experienced hands (such as the CAS operations) being lumped in with low-level methods that are of no real use to library developers.

Pending removal in Java 9 with Modularisation

Access to `sun.misc.Unsafe` is currently slated to be prevented in the upcoming Java 9 release as part of Project Jigsaw. This will break frameworks that do not (or cannot) offer a sufficient fallback and will still harm frameworks that do provide a fallback implementation, as the primary reason for adopting Unsafe in the first place is usually performance.

While additional JNI libraries could provide the same functionality as Unsafe, such libraries would need to provide 32-bit and 64-bit as well as Windows and Linux variations. This is less safe than the Unsafe class in Java 8, or a potential replacement in Java 9, as each framework would have to offer it's own implementation. To achieve comparable performance, more functionality would need to be migrated into C. e.g. an operation to read or write a String in UTF-8 format to/from native memory can be written in Java currently and achieve near C speeds, but without the intrinsics available in Unsafe, such an operation would have to be written in JNI to avoid crossing the JNI barrier too many times.

Missing Cross-Vendor Specification

The current sun.misc.Unsafe class is not specified. Content changes from version to version and vendor to vendor. Cross-JVM implementations need to check for a lot of circumstances to make sure the Unsafe based implementation works on most JVMs.

Uses of Unsafe

- Low, very predictable latencies (low GC overhead)
- Fast de-/serialization
- Thread safe 64-bit sized native memory access (for example offheap)
- Atomic memory operations
- Efficient object / memory layouts
- Fast field / memory access
- Custom memory fences
- Fast interaction with native code
- Multi-operating system replacement for JNI.
- "Type hijacking" of classes for type-safe APIs without calling a constructor.
- Access to array items with volatile semantic

Examples of projects/products using Unsafe

- MapDB
- Netty
- Hazelcast
- Cassandra
- Mockito / EasyMock / JMock / PowerMock
- Scala Specs
- Spock
- Robolectric

- Grails
- Neo4j
- Apache Kafka
- Apache Wink
- Apache Storm
- Apache Continuum
- Zookeeper
- Dropwizard
- Metrics (AOP)
- Kryo
- Byte Buddy
- Hibernate
- Liquibase
- Spring Framework
- Ehcache (sizeof)
- OrientDB
- Chronicles (OpenHFT)
- Apache Hadoop, Apache HBase (hadoop based database)
- GWT
- Disruptor
- jRuby
- Scala
- Akka
- Real Logic Argona
- XRebel
- ...

GAP Analysis of JEPs

Here's an attempt to look at what JEPs would need to be raised to provide safe, support versions of existing 'safe' `sun.misc.Unsafe` features.

Possible Replacements for some aspects (mentioned in the past)

Green = Already available / Will be available in Java 9
Orange = May be available in Java 9
Red = Unlikely to be available in Java 9

Proposal	Expected in Java 9
VarHandle (no JEP)	no

Project Panama (JFFI, JEP 191)	no
Serialization 2.0 (JEP 187)	no (JEP disappeared - wayback machine)
ValueTypes (no JEP)	no
Enhanced Volatiles (JEP 193)	maybe
Arrays 2.0 (no JEP)	no
Variable Object Layout (no JEP)	no
Byte Buffers	available today (but missing 64 bit and atomic operations)
Extending Field / Array reflection access	not yet discussed

What is Needed

A roadmap for the retirement/replacement of Unsafe

Ideally, the retirement of Unsafe should be governed by a JEP within OpenJDK, with a JSR to cover the standardisation of the addition of the "good / safe pieces of Unsafe".

A mapping of Unsafe Features to JEPs / Features

In OpenJDK 7 sun.misc.Unsafe consisted of 105 methods. These subdivide into a few groups of important methods for manipulating various entities. Here are some of the main groupings:

Off-heap memory access is the number one used feature followed by Memory Information.

Green = Full Replacement
Orange = Possible Replacement (partly replacing the functionality)
Red = None

Feature	**sun.misc.Unsafe**	**Usage** Google Search Results[$]	**Java 9 replacement, if any**
Memory Information	addressSize pageSize	17,700 65,800	
Objects	allocateInstance	5,290	Reflection (Field),

	objectFieldOffset	2,820	JEP 193?
Classes	staticFieldOffset defineClass defineAnonymousClass ensureClassInitialized	2,820 11,400 2,350 2,760	
Arrays	arrayBaseOffset arrayIndexScale	1,560 4,960	Reflection (Array), Enhanced Volatiles - JEP 193?
Synchronization	monitorEnter tryMonitorEnter monitorExit park unpark	4,680 2,360 14,700 N/A 13,200	Existing Java syntax and libraries. park / unpark by using LockSupport

"Safe Unsafe" On-heap Object access
 	
Note: All other access operations (e.g. getX/putX with address argument) are currently invalid (as in "will cause random heap corruption") for on-heap object access
	Unordered field access:
	  getX(Object o, ...)
	  putX(Object o, ...)
	Volatile/ordered field access:
	  getXVolatile
	  putXVolatile
	  putOrderedX
	Atomics:
	  compareAndSwapX
	  getAndAddX
	  getAndSetX
	Fences:
	  storeFence
	  readFence
	  fullFence
	copyMemory(Object src, …, Object dst, ...)
	setMemory(Object o, …)
		
		26.300 (object)
		5.420 (object)
		
		3.350 (object)
		3.110 (object)
		4.030 (int)
		
		3.800 (int)
		1.010 (int)
		290 (int)
		
		1.820
		202
		1.900
		19.400
		
		19.900
			Reflection (Field, Array).
			Enhanced Volatiles - JEP 193?
			VarHandle?, Arrays 2.0?, Variable Object Layout,
			A fences API JEP?

Off-heap Memory access	allocateMemory
	freeMemory
	copyMemory
	setMemory
	getAddress
	Unordered field access:
	  getX(long address, …)
	  putX(long address, …)
	Volatile/ordered field access:
	  getXVolatile(0, …)
	  putXVolatile(0, …)
		39,200
		122,000
		19.400
		19.900
		10.600
		
		26.300 (object)
		5.420 (object)
		
		3.350 (object)
		3.110 (object)
			Mapped Byte Buffers (Speed-enhanced mapped buffer? e.g. ones with a constant limit that can allow compilers to avoid range checks in loops)
			Variable Object Layout,
			A fences API JEP?

	putOrderedX(0, …) Atomics: compareAndSwapX(0, ...) getAndAddX(0, …) getAndSetX(0, …) Fences: storeFence readFence fullFence	4.030 (int) 3.800 (int) 1.010 (int) 290 (int) 1.820 202 1.900	

[$] An indicator of popularity

Fields in sun.misc.Unsafe

	INVALID_FIELD_OFFSET	This constant differs from all results that will ever be returned from #staticFieldOffset , #objectFieldOffset , or #arrayBaseOffset .
	ARRAY_BOOLEAN_BASE_OFFSET	The value of {@code arrayBaseOffset(boolean[].class)}
	ARRAY_BYTE_BASE_OFFSET	The value of {@code arrayBaseOffset(byte[].class)}
	ARRAY_SHORT_BASE_OFFSET	The value of {@code arrayBaseOffset(short[].class)}
	ARRAY_CHAR_BASE_OFFSET	The value of {@code arrayBaseOffset(char[].class)}
	ARRAY_INT_BASE_OFFSET	The value of {@code arrayBaseOffset(int[].class)}
	ARRAY_LONG_BASE_OFFSET	The value of {@code arrayBaseOffset(long[].class)}
	ARRAY_FLOAT_BASE_OFFSET	The value of {@code arrayBaseOffset(float[].class)}
	ARRAY_DOUBLE_BASE_OFFSET	The value of {@code arrayBaseOffset(double[].class)}
	ARRAY_OBJECT_BASE_OFFSET	The value of {@code arrayBaseOffset(Object[].class)}
	ARRAY_BOOLEAN_INDEX_SCALE	The value of {@code arrayIndexScale(boolean[].class)}
	ARRAY_BYTE_INDEX_SCALE	The value of {@code arrayIndexScale(byte[].class)}

	ARRAY_SHORT_INDEX_SCALE	The value of {@code arrayIndexScale(short[].class)}
	ARRAY_CHAR_INDEX_SCALE	The value of {@code arrayIndexScale(char[].class)}
	ARRAY_INT_INDEX_SCALE	The value of {@code arrayIndexScale(int[].class)}
	ARRAY_LONG_INDEX_SCALE	The value of {@code arrayIndexScale(long[].class)}
	ARRAY_FLOAT_INDEX_SCALE	The value of {@code arrayIndexScale(float[].class)}
	ARRAY_DOUBLE_INDEX_SCALE	The value of {@code arrayIndexScale(double[].class)}
	ARRAY_OBJECT_INDEX_SCALE	The value of {@code arrayIndexScale(Object[].class)}
	ADDRESS_SIZE	The value of {@code addressSize()}

Proposal - Working Group

In order to propose sane replacements in OpenJDK via the JEP process, there needs to be some serious analysis from the industry at large. Here is an (incomplete) list of people/orgs who will assist in the GAP analysis.

Working Group Members

Geir Magnusson
London JUG - Martijn Verburg & Ben Evans
Hazelcast Inc. - Greg Luck and Chris Engelbert
Credit Suisse
Microdoc - Hendrik
Azul - Gil Tene
Goldman Sachs - John Weir
Fujitsu - Mike De Nicola
SouJava
Eclipse Software Foundation
DataStax - Jonathan Ellis and Ariel Weisberg
Peter Lawrey