JEP 370: Foreign-Memory Access API (Incubator)

Owner	Maurizio Cimadamore
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	14
Component	core-libs
Discussion	panama dash dev at openjdk dot java dot net
Relates to	JEP 383: Foreign-Memory Access API (Second Incubator) JEP 393: Foreign-Memory Access API (Third Incubator)
Reviewed by	Brian Goetz, John Rose
Endorsed by	Mark Reinhold
Created	2019/07/09 15:55
Updated	2021/08/28 00:20
Issue	8227446

Summary

Introduce an API to allow Java programs to safely and efficiently access foreign memory outside of the Java heap.

Goals

The foreign memory API should satisfy the following criteria:

- *Generality:* The same API should be able to operate on various kinds of foreign memory (e.g., native memory, persistent memory, managed heap memory, etc.).
- *Safety:* It should not be possible for the API to undermine the safety of the JVM, regardless of the kind of memory being operated upon.
- *Determinism:* Memory deallocation operations should be explicit in the source code.

Success Metrics

The foreign memory API should be a valid alternative to the main avenues by which Java programs access foreign memory today, namely `java.nio.ByteBuffer` and `sun.misc.Unsafe`. The new API should be performance-competitive with these existing APIs.

Motivation

Many existing Java libraries and programs access foreign memory, for example Ignite, mapDB, memcached, and Netty's ByteBuf API. By doing so they can

- Avoid the cost and unpredictability associated with garbage collection (especially when maintaining large caches),
- Share memory across multiple processes, and
- Serialize and deserialize memory content by mapping files into memory (via, e.g., `mmap`).

Yet, the Java API does not provide a satisfactory solution for accessing foreign memory.

The `ByteBuffer` API, introduced in Java 1.4, allows the creation of *direct* byte buffers, which are allocated off-heap, and allows users to manipulate off-heap memory directly from Java. Direct buffers are, however, limited. For instance, it is not possible to create a buffer that is larger than two gigabytes, since the `ByteBuffer` API uses an `int`-based indexing scheme. Moreover, working with direct buffers can be cumbersome since deallocation of the memory associated with direct buffers is left to the garbage collector; that is, only after a direct buffer is deemed unreachable by the garbage collector can the associated memory be released. Over the years many requests for enhancement have been filed in order to overcome these and other limitations (e.g., 4496703, 6558368, 4837564 and

Mission Control
Modules
Multi-Language VM
Nashorn
New I/O
OpenJFX
Panama
Penrose
Port: AArch32
Port: AArch64
Port: BSD
Port: Haiku
Port: Mac OS X
Port: MIPS
Port: Mobile
Port: PowerPC/AIX
Port: RISC-V
Port: s390x
Portola
SCTP
Shenandoah
Skara
Sumatra
ThreeTen
Tiered Attribution
Tsan
Type Annotations
XRender Pipeline
Valhalla
Verona
VisualVM
Wakefield
Zero
ZGC

5029431). Many of these limitations stem from the fact that the `ByteBuffer` API was designed not only for off-heap memory access but also for producer/consumer exchanges of bulk data which are critical to, e.g., charset encoding/decoding and partial I/O operations.

Another common avenue by which developers can access foreign memory from Java code is the `sun.misc.Unsafe` API. `Unsafe` exposes many memory access operations (e.g., `Unsafe::getInt` and `putInt`) which work for on-heap as well as off-heap access thanks to a clever and relatively general addressing model. Using `Unsafe` to access memory is extremely efficient: All memory access operations are defined as JVM intrinsics, so memory access operations are routinely optimized by the JIT. Unfortunately, the `Unsafe` API is, by definition, *unsafe* -- it allows access to any memory location (e.g., `Unsafe::getInt` takes a `long` address). This makes it possible for a Java program to crash the JVM if, e.g., some already-freed memory location is accessed. On top of that, the Unsafe API is not a supported Java API, and its use has always been strongly discouraged.

While using JNI to access memory is also a possibility, the inherent costs associated with this solution make it seldom applicable in practice. The overall development pipeline is complex, since JNI requires the developer to write and maintain snippets of C code. JNI is also inherently slow, since a Java-to-native transition is required for each access.

In summary, when it comes to accessing foreign memory, developers are faced with a dilemma: Should they use a safe but limited (and possibly less efficient) path (e.g., `ByteBuffer`), or should they abandon safety guarantees and embrace the unsupported and dangerous `Unsafe` API?

This JEP introduces a supported, safe, and efficient foreign-memory access API. By providing a targeted solution to the problem of accessing foreign memory, developers will be freed of the limitations and dangers of existing APIs. They will also enjoy improved performance, since the new API will be designed from the ground up with JIT optimizations in mind.

Description

The foreign-memory access API introduces three main abstractions: `MemorySegment`, `MemoryAddress` and `MemoryLayout`.

A `MemorySegment` is used to model a contiguous memory region with given spatial and temporal bounds. A `MemoryAddress` can be thought of as an offset within a segment. Finally, a `MemoryLayout` is a programmatic description of a memory segment's contents.

Memory segments can be created from a variety of sources, such as native memory buffers, Java arrays, and byte buffers (either direct or heap-based). For instance, a native memory segment can be created as follows:

```
try (MemorySegment segment = MemorySegment.allocateNative(100)) {
   ...
}
```

This will create a memory segment that is associated with a native memory buffer whose size is 100 bytes.

Memory segments are *spatially bounded;* that is, they have lower and upper bounds. Any attempt to use the segment to access memory outside of these bounds will result in an exception. As evidenced by the use of the `try`-with-resource construct, segments are also *temporally bounded;* that is, they are created, used, and then closed when no longer in use. Closing a segment is always an explicit operation and can result in additional side effects, such as the deallocation of the memory associated with the segment. Any attempt to access an already-closed memory segment will result in an exception. Together, spatial and temporal safety checks are crucial to guarantee the safety of the memory access API and thus, e.g., the absence of hard JVM crashes.

Dereferencing the memory associated with a segment can be achieved by obtaining a *memory-access var handle*. These special var handles have at least

one mandatory access coordinate, of type `MemoryAddress`, which is the address at which the dereference occurs. They are obtained using factory methods in the `MemoryHandles` class. For instance, to set the elements of a native segment we could use a memory-access var handle as follows:

```
VarHandle intHandle = MemoryHandles.varHandle(int.class,
        ByteOrder.nativeOrder());

try (MemorySegment segment = MemorySegment.allocateNative(100)) {
    MemoryAddress base = segment.baseAddress();
    for (int i = 0; i < 25; i++) {
        intHandle.set(base.addOffset(i * 4), i);
    }
}
```

Memory-access var handles can also acquire one or more extra access coordinates, of type `long`, to support more complex addressing schemes such as multi-dimensional indexed access. Such memory-access var handles are typically obtained by invoking one or more combinator methods, also defined in the `MemoryHandles` class. For instance, a more direct way to set the elements of a native segment is through an indexed memory-access handle, constructed as follows:

```
VarHandle intHandle = MemoryHandles.varHandle(int.class,
        ByteOrder.nativeOrder());
VarHandle intElemHandle = MemoryHandles.withStride(intHandle, 4);

try (MemorySegment segment = MemorySegment.allocateNative(100)) {
    MemoryAddress base = segment.baseAddress();
    for (int i = 0; i < 25; i++) {
        intElemHandle.set(base, (long) i, i);
    }
}
```

This effectively allows rich, multi-dimensional addressing of an otherwise flat memory buffer.

To enhance the expressiveness of the API, and to reduce the need for explicit numeric computations such as those in the above examples, the `MemoryLayout` API can be used to programmatically describe the content of a memory segment. For instance, the layout of the native memory segment used in the above examples can be described in the following way:

```
SequenceLayout intArrayLayout
    = MemoryLayout.ofSequence(25,
        MemoryLayout.ofValueBits(32,
            ByteOrder.nativeOrder()));
```

This creates a *sequence* memory layout in which a given element layout (a 32-bit value) is repeated 25 times. Once we have a memory layout, we can get rid of all the manual numeric computation in our code and also simplify the creation of the required memory access var handles, as shown in the following example:

```
SequenceLayout intArrayLayout
    = MemoryLayout.ofSequence(25,
        MemoryLayout.ofValueBits(32,
            ByteOrder.nativeOrder()));

VarHandle intElemHandle
    = intArrayLayout.varHandle(int.class,
        PathElement.sequenceElement());

try (MemorySegment segment = MemorySegment.allocateNative(intArrayLayout)) {
    MemoryAddress base = segment.baseAddress();
    for (int i = 0; i < intArrayLayout.elementCount().getAsLong(); i++) {
        intElemHandle.set(base, (long) i, i);
```

```
        }
    }
```

In this example, the layout instance drives the creation of the memory-access var handle through the creation of a *layout path*, which is used to select a nested layout from a complex layout expression. The layout instance also drives the allocation of the native-memory segment, which is based upon size and alignment information derived from the layout. The loop constant in the previous examples has been replaced with the sequence layout's element count.

The foreign-memory access API will initially be provided as an incubating module, named `jdk.incubator.foreign`, in a package of the same name.

Alternatives

Keep using existing APIs such as `ByteBuffer` and `Unsafe` or, worse, JNI.

Risks and Assumptions

Creating a foreign-memory access API which is both safe and efficient is a daunting task. Since the spatial and temporal checks described in the previous sections need to be performed upon every access, it is crucial that JITs be able to optimize away these checks by, e.g., hoisting them outside of hot loops. The JIT implementations will likely require some work to ensure that uses of the memory access API are as efficient and as optimizable as uses of existing APIs such as `ByteBuffer` and `Unsafe`.

Dependencies

The API described in this JEP will likely help the development of the native interoperation support that is a goal of Project Panama. This API can also be used to access non-volatile memory, already possible via JEP 352 (Non-Volatile Mapped Byte Buffers), in a more general and efficient way.