JEP 412: Foreign Function & Memory API (Incubator)

Owner	Maurizio Cimadamore
Type	Feature
Scope	JDK
Status	Closed / Delivered
Release	17
Component	core-libs
Discussion	panama dash dev at openjdk dot java dot net
Effort	M
Duration	M
Relates to	JEP 424: Foreign Function & Memory API (Preview) JEP 389: Foreign Linker API (Incubator) JEP 393: Foreign-Memory Access API (Third Incubator) JEP 419: Foreign Function & Memory API (Second Incubator)
Reviewed by	Adam Pocock, Alex Buckley, Paul Sandoz
Created	2021/04/10 21:05
Updated	2022/03/02 17:11
Issue	8265033

Summary

Introduce an API by which Java programs can interoperate with code and data outside of the Java runtime. By efficiently invoking foreign functions (i.e., code outside the JVM), and by safely accessing foreign memory (i.e., memory not managed by the JVM), the API enables Java programs to call native libraries and process native data without the brittleness and danger of JNI.

History

The API proposed in this JEP is the evolution of two incubating APIs: the Foreign-Memory Access API and the Foreign Linker API. The Foreign-Memory Access API was first proposed by JEP 370 and targeted to Java 14 in late 2019 as an incubating API; it was re-incubated by JEP 383 in Java 15 and by JEP 393 in Java 16. The Foreign Linker API was first proposed by JEP 389 and targeted to Java 16 in late 2020, also as an incubating API.

Goals

- *Ease of use* — Replace the Java Native Interface (JNI) with a superior, pure-Java development model.
- *Performance* — Provide performance that is comparable to, if not better than, existing APIs such as JNI and `sun.misc.Unsafe`.
- *Generality* — Provide ways to operate on different kinds of foreign memory (e.g., native memory, persistent memory, and managed heap memory) and, over time, to accommodate other platforms (e.g., 32-bit x86) and foreign functions written in languages other than C (e.g., C++, Fortran).
- *Safety* — Disable unsafe operations by default, allowing them only after explicit opt-in from application developers or end users.

Non-goals

It is not a goal to

- Re-implement JNI on top of this API, or otherwise change JNI in any way;
- Re-implement legacy Java APIs, such as `sun.misc.Unsafe`, on top of this API;
- Provide tooling that mechanically generates Java code from native-code header files; or
- Change how Java applications that interact with native libraries are packaged and deployed (e.g., via multi-platform JAR files).

Motivation

The Java Platform has always offered a rich foundation to library and application developers who wish to reach beyond the JVM and interact with other platforms. Java APIs expose non-Java resources in a convenient and reliable fashion, whether to access remote data (JDBC), invoke web services (HTTP client), serve remote clients (NIO channels), or communicate with local processes (Unix-domain sockets). Unfortunately, Java developers still face significant obstacles in accessing an

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

important kind of non-Java resource: code and data on the same machine as the JVM, but outside the Java runtime.

Foreign memory

Data stored in memory outside the Java runtime is referred to as *off-heap* data. (The *heap* is where Java objects live — *on-heap* data — and where garbage collectors do their work.) Accessing off-heap data is critical for the performance of popular Java libraries such as Tensorflow, Ignite, Lucene, and Netty, primarily because it lets them avoid the cost and unpredictability associated with garbage collection. It also allows data structures to be serialized and deserialized by mapping files into memory via, e.g., `mmap`. However, the Java Platform does not today provide a satisfactory solution for accessing off-heap data.

- The `ByteBuffer` API allows for the creation of *direct* byte buffers that are allocated off-heap, but their maximum size is two gigabytes and they are not deallocated promptly. These and other 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 in areas such as charset encoding/decoding and partial I/O operations. In that context it has not been possible to satisfy the many requests for off-heap enhancements filed over the years (e.g., 4496703, 6558368, 4837564, and 5029431).
- The `sun.misc.Unsafe` API exposes memory access operations for on-heap data that also work for off-heap data. Using `Unsafe` is efficient because its memory access operations are defined as HotSpot JVM intrinsics and optimized by the JIT compiler. However, using `Unsafe` is dangerous because it allows access to any memory location. This means that a Java program can crash the JVM by accessing an already-freed location; for this and other reasons, the use of `Unsafe` has always been strongly discouraged.
- Using JNI to call a native library which then accesses off-heap data is possible, but the performance overhead seldom makes it applicable: Going from Java to native is several orders of magnitude slower than accessing memory because JNI method calls do not benefit from many common JIT optimizations such as inlining.

In summary, when it comes to accessing off-heap data, Java developers face a dilemma: Should they choose a safe but inefficient path (`ByteBuffer`) or should they abandon safety in favor of performance (`Unsafe`)? What they require is a supported API for accessing off-heap data (i.e., foreign memory) designed from the ground up to be safe and with JIT optimizations in mind.

Foreign functions

JNI has supported the invocation of native code (i.e., foreign functions) since Java 1.1, but it is inadequate for many reasons.

- JNI involves several tedious artifacts: a Java API (`native` methods), a C header file derived from the Java API, and a C implementation that calls the native library of interest. Java developers must work across multiple toolchains to keep platform-dependent artifacts in sync, which is especially burdensome when the native library evolves rapidly.
- JNI can only interoperate with libraries written in languages, typically C and C++, that use the calling convention of the operating system and CPU for which the JVM was built. A `native` method cannot be used to invoke a function written in a language that uses a different convention.
- JNI does not reconcile the Java type system with the C type system. Aggregate data in Java is represented with objects, but aggregate data in C is represented with structs, so any Java object passed to a `native` method must be laboriously unpacked by native code. For example, consider a record class `Person` in Java: Passing a `Person` object to a `native` method will require the native code to use JNI's C API to extract fields (e.g., `firstName` and `lastName`) from the object. As a result, Java developers sometimes flatten their data into a single object (e.g., a byte array or a direct byte buffer) but more often, since passing Java objects via JNI is slow, they use the `Unsafe` API to allocate off-heap memory and pass its address to a `native` method as a `long` — which makes the Java code tragically unsafe!

Over the years, numerous frameworks have emerged to fill the gaps left by JNI, including JNA, JNR and JavaCPP. While these frameworks are often a marked improvement over JNI, the situation is still less than ideal, especially when

compared with languages which offer first-class native interoperation. For example, Python's ctypes package can dynamically wrap functions in native libraries without any glue code. Other languages, such as Rust, provide tools which mechanically derive native wrappers from C/C++ header files.

Ultimately, Java developers should have a supported API that lets them straightforwardly consume any native library deemed useful for a particular task, without the tedious glue and clunk of JNI. An excellent abstraction to build upon is *method handles*, introduced in Java 7 to support fast dynamic languages on the JVM. Exposing native code via method handles would radically simplify the task of writing, building, and distributing Java libraries which depend upon native libraries. Furthermore, an API capable of modeling foreign functions (i.e., native code) and foreign memory (i.e., off-heap data) would provide a solid foundation for third-party native interoperation frameworks.

Description

The Foreign Function & Memory API (FFM API) defines classes and interfaces so that client code in libraries and applications can

- Allocate foreign memory (`MemorySegment`, `MemoryAddress`, and `SegmentAllocator`),
- Manipulate and access structured foreign memory (`MemoryLayout`, `MemoryHandles`, and `MemoryAccess`),
- Manage the lifecycle of foreign resources (`ResourceScope`), and
- Call foreign functions (`SymbolLookup` and `CLinker`).

The FFM API resides in the `jdk.incubator.foreign` package of the `jdk.incubator.foreign` module.

Example

As a brief example of using the FFM API, here is Java code that obtains a method handle for a C library function `radixsort` and then uses it to sort four strings which start life in a Java array (a few details are elided):

```
// 1. Find foreign function on the C library path
MethodHandle radixSort = CLinker.getInstance().downcallHandle(
                            CLinker.systemLookup().lookup("radixsort"), ...);
// 2. Allocate on-heap memory to store four strings
String[] javaStrings   = { "mouse", "cat", "dog", "car" };
// 3. Allocate off-heap memory to store four pointers
MemorySegment offHeap  = MemorySegment.allocateNative(
                            MemoryLayout.ofSequence(javaStrings.length,
                                                    CLinker.C_POINTER), ...);
// 4. Copy the strings from on-heap to off-heap
for (int i = 0; i < javaStrings.length; i++) {
    // Allocate a string off-heap, then store a pointer to it
    MemorySegment cString = CLinker.toCString(javaStrings[i], newImplicitScope());
    MemoryAccess.setAddressAtIndex(offHeap, i, cString.address());
}
// 5. Sort the off-heap data by calling the foreign function
radixSort.invoke(offHeap.address(), javaStrings.length, MemoryAddress.NULL, '\0');
// 6. Copy the (reordered) strings from off-heap to on-heap
for (int i = 0; i < javaStrings.length; i++) {
    MemoryAddress cStringPtr = MemoryAccess.getAddressAtIndex(offHeap, i);
    javaStrings[i] = CLinker.toJavaStringRestricted(cStringPtr);
}
assert Arrays.equals(javaStrings, new String[] {"car", "cat", "dog", "mouse"});  // true
```

This code is far clearer than any solution that uses JNI, since implicit conversions and memory dereferences that would have been hidden behind `native` method calls are now expressed directly in Java. Modern Java idioms can also be used; for example, streams can allow for multiple threads to copy data between on-heap and off-heap memory in parallel.

Memory segments

A *memory segment* is an abstraction that models a contiguous region of memory, located either off-heap or on-heap. Memory segments can be

- *Native* segments, allocated from scratch in native memory (e.g., via `malloc`),
- *Mapped* segments, wrapped around a region of mapped native memory (e.g., via `mmap`), or

- *Array* or *buffer* segments, wrapped around memory associated with existing Java arrays or byte buffers, respectively.

All memory segments provide spatial, temporal, and thread-confinement guarantees which are strongly enforced in order to make memory dereference operations safe. For example, the following code allocates 100 bytes off-heap:

```
MemorySegment segment = MemorySegment.allocateNative(100, newImplicitScope());
```

The *spatial bounds* of a segment determine the range of memory addresses associated with the segment. The bounds of the segment in the code above are defined by a *base address* `b`, expressed as a `MemoryAddress` instance, and a size in bytes (100), resulting in a range of addresses from `b` to `b + 99`, inclusive.

The *temporal bounds* of a segment determine the lifetime of the segment, that is, when the segment will be deallocated. A segment's lifetime and thread-confinement state is modeled by a `ResourceScope` abstraction, discussed below. The resource scope in the code above is a new *implicit* scope, which ensures that the memory associated with this segment is freed when the `MemorySegment` object is deemed unreachable by the garbage collector. The implicit scope also ensures that the memory segment is accessible from multiple threads.

In other words, the code above creates a segment whose behavior closely matches that of a `ByteBuffer` allocated with the `allocateDirect` factory. The FFM API also supports deterministic memory release and other thread-confinement options, discussed below.

Dereferencing memory segments

Dereferencing the memory associated with a segment is achieved by obtaining a *var handle*, an abstraction for data access introduced in Java 9. In particular, a segment is dereferenced with a *memory-access var handle*. This kind of var handle uses a pair of access coordinates:

- A coordinate of type `MemorySegment` — the segment whose memory is to be dereferenced, and
- A coordinate of type `long` — the offset, from the segment's base address, at which dereference occurs.

Memory-access var handles are obtained via factory methods in the `MemoryHandles` class. For example, this code obtains a memory-access var handle that can write an `int` value into a native memory segment, and uses it to write 25 four-byte values at consecutive offsets:

```
MemorySegment segment = MemorySegment.allocateNative(100, newImplicitScope());
VarHandle intHandle = MemoryHandles.varHandle(int.class, ByteOrder.nativeOrder());
for (int i = 0; i < 25; i++) {
    intHandle.set(segment, /* offset */ i * 4, /* value to write */ i);
}
```

More advanced access idioms can be expressed by combining memory-access var handles using one or more of the combinator methods provided by the `MemoryHandles` class. With these a client can, e.g., reorder the coordinates of a given memory-access var handle, drop one or more coordinates, and insert new coordinates. This allows the creation of memory access var handles which accept one or more logical indices into a multi-dimensional array backed by a flat off-heap memory region.

To make the FFM API more approachable, the `MemoryAccess` class provides static accessors to dereference memory segments without the need to construct memory-access var handles. For example, there is an accessor to set an `int` value in a segment at a given offset, allowing the code above to be simplified to:

```
MemorySegment segment = MemorySegment.allocateNative(100, newImplicitScope());
for (int i = 0; i < 25; i++) {
    MemoryAccess.setIntAtOffset(segment, i * 4, i);
}
```

Memory layouts

To reduce the need for tedious calculations about memory layout (e.g., `i * 4` in the example above), a `MemoryLayout` can be used to describe the content of a memory segment in a more declarative fashion. For example, the desired layout of the native memory segment in the examples above can be described in the following way:

```
SequenceLayout intArrayLayout
    = MemoryLayout.sequenceLayout(25,
        MemoryLayout.valueLayout(32, ByteOrder.nativeOrder()));
```

This creates a *sequence memory layout* in which a 32-bit *value layout* (a layout describing a single 32-bit value) is repeated 25 times. Given a memory layout, we can avoid calculating offsets in our code and simplify both memory allocation and the creation of memory-access var handles:

```
MemorySegment segment = MemorySegment.allocateNative(intArrayLayout, newImplicitScope());
VarHandle indexedElementHandle =
    intArrayLayout.varHandle(int.class, PathElement.sequenceElement());
for (int i = 0; i < intArrayLayout.elementCount().getAsLong(); i++) {
    indexedElementHandle.set(segment, (long) i, i);
}
```

The `intArrayLayout` object 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 `intArrayLayout` object 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, 25, has been replaced with the sequence layout's element count.

Resource scopes

All of the memory segments seen in the previous examples use non-deterministic deallocation: The memory associated with these segments is deallocated by the garbage collector, once the memory segment instance becomes unreachable. We say that such segments are *implicitly deallocated*.

There are cases where the client might want to control when memory deallocation occurs. Suppose, e.g., that a large memory segment is mapped from a file using `MemorySegment::map`. The client might prefer to release (i.e., unmap) the memory associated with the segment as soon as the segment is no longer required rather than wait for the garbage collector to do so, since waiting could adversely affect the application's performance.

Memory segments support deterministic deallocation through *resource scopes*. A resource scope models the lifecycle associated with one or more *resources,* such as memory segments. A newly-created resource scope is in the *alive* state, which means that all the resources it manages can be safely accessed. At the client's request a resource scope can be *closed*, which means that access to the resources managed by the scope is no longer allowed. The `ResourceScope` class implements the `AutoCloseable` interface so that resource scopes work with the try-with-resources statement:

```
try (ResourceScope scope = ResourceScope.newConfinedScope()) {
    MemorySegment s1 = MemorySegment.map(Path.of("someFile"), 0, 100000,
                                         MapMode.READ_WRITE, scope);
    MemorySegment s2 = MemorySegment.allocateNative(100, scope);
    ...
} // both segments released here
```

This code creates a *confined* resource scope and uses it in the creation of two segments: a mapped segment (`s1`) and a native segment (`s2`). The lifecycle of the two segments is tied to the lifetime of the resource scope, so accessing the segments (e.g., dereferencing them with memory-access var handles) after the try-with-resources statement has completed will cause a runtime exception to be thrown.

In addition to managing a memory segment's lifetime, a resource scope also serves as a means to control which threads can access the segment. A confined resource scope restricts access to the thread which created the scope, whereas a *shared* resource scope allows access from any thread.

A resource scope, whether confined or shared, may be associated with a `java.lang.ref.Cleaner` object that takes care of performing implicit deallocation in case the resource scope object becomes unreachable before the `close` method is called by the client.

Some resource scopes, referred to as *implicit* resource scopes, do not support explicit deallocation — calling `close` will fail. Implicit resource scopes always manage their resources using a `Cleaner`. Implicit scopes can be created using the `ResourceScope::newImplicitScope` factory, as shown in earlier examples.

Segment allocators

Memory allocation can often be a bottleneck when clients use off-heap memory. The FFM API includes a `SegmentAllocator` abstraction, which defines useful operations to allocate and initialize memory segments. Segment allocators are obtained via factories in the `SegmentAllocator` interface. For example, the

following code creates an arena-based allocator and uses it to allocate a segment whose content is initialized from a Java `int` array:

```
try (ResourceScope scope = ResourceScope.newConfinedScope()) {
    SegmentAllocator allocator = SegmentAllocator.arenaAllocator(scope);
    for (int i = 0 ; i < 100 ; i++) {
        MemorySegment s = allocator.allocateArray(C_INT, new int[] { 1, 2, 3, 4, 5 });
        ...
    }
    ...
} // all memory allocated is released here
```

This code creates a confined resource scope and then creates an *unbounded arena allocator* associated with that scope. This allocator will allocate slabs of memory, of a specific size, and respond to allocation requests by returning different slices of the pre-allocated slab. If a slab does not have sufficient space to accommodate a new allocation request, a new slab is allocated. If the resource scope associated with the arena allocator is closed, all memory associated with the segments created by the allocator (i.e., in the body of the `for` loop) is deallocated atomically. This idiom combines the advantages of deterministic deallocation, provided by the `ResourceScope` abstraction, with a more flexible and scalable allocation scheme. It can be very useful when writing code which manages a large number of off-heap segments.

Unsafe memory segments

So far, we have seen memory segments, memory addresses, and memory layouts. Dereference operations are only possible on memory segments. Since a memory segment has spatial and temporal bounds, the Java runtime can always ensure that memory associated with a given segment is dereferenced safely. However, there are situations where clients might only have a `MemoryAddress` instance, as is often the case when interacting with native code. Since the Java runtime has no way to know the spatial and temporal bounds associated with a memory address, directly dereferencing memory addresses is forbidden by the FFM API.

To dereference a memory address, a client has two options.

- If the address is known to fall within a memory segment, the client can perform a *rebase* operation via `MemoryAddress::segmentOffset`. The rebasing operation re-interprets the address's offset relative to the segment's base address to yield a new offset which can be applied to the existing segment — which can then be safely dereferenced.
- Alternatively, if no such segment exists then the client can create one *unsafely*, using the `MemoryAddress::asSegment` factory. This factory effectively attaches fresh spatial and temporal bounds to an otherwise raw memory address, so as to allow dereference operations. The memory segment returned by this factory is *unsafe*: A raw memory address might be associated with a memory region that is 10 bytes long, but the client might accidentally overestimate the size of the region and create an unsafe memory segment that is 100 bytes long. This might result, later, in attempts to dereference memory outside the bounds of the memory region associated with the unsafe segment, which might cause a JVM crash or, worse, result in silent memory corruption. For this reason, creating unsafe segments is regarded as a *restricted operation*, and is disabled by default (see more below).

Looking up foreign functions

The first ingredient of any support for foreign functions is a mechanism to load native libraries. With JNI, this is accomplished with the `System::loadLibrary` and `System::load` methods, which internally map into calls to `dlopen` or its equivalent. Libraries loaded using these methods are always associated with a class loader (namely, the loader of the class which called the `System` method). The association between libraries and class loaders is crucial because it governs the **lifecycle** of loaded libraries: only when a class loader is no longer reachable, can all its libraries be unloaded **safely**.

The FFM API does not provide new methods for loading native libraries. Developers use the `System::loadLibrary` and `System::load` methods to load native libraries that will invoked via the FFM API. The association between libraries and class loaders is preserved, so libraries will be unloaded in the same predictable manner as with JNI.

The FFM API, unlike JNI, provides the capability to find the address of a given symbol in a loaded library. This capability, represented by a `SymbolLookup` object,

is crucial for linking Java code to foreign functions (see below). There are two ways to obtain a `SymbolLookup` object:

- `SymbolLookup::loaderLookup` returns a symbol lookup which sees all the symbols in all the libraries that were loaded by the current class loader.
- `CLinker::systemLookup` returns a platform-specific symbol lookup which sees symbols in the standard C library.

Given a symbol lookup, a client can find a foreign function with the `SymbolLookup::lookup(String)` method. If the named function is present among the symbols seen by the symbol lookup, then the method returns a `MemoryAddress` which points to the entry point of the function. For example, the following code loads the OpenGL library (causing it to be associated with the current class loader) and finds the address of its `glGetString` function:

```
System.loadLibrary("GL");
SymbolLookup loaderLookup  = SymbolLookup.loaderLookup();
MemoryAddress clangVersion = loaderLookup.lookup("glGetString").get();
```

Linking Java code to foreign functions

The `CLinker` interface is the core of how Java code interoperates with native code. While the `CLinker` is focused on providing interoperation between Java and C libraries, the concepts in the interface are general enough to support other, non-Java languages in future. The interface enables both *downcalls* (calls from Java code to native code) and *upcalls* (calls from native code back to Java code).

```
interface CLinker {
    MethodHandle downcallHandle(MemoryAddress func,
                                MethodType type,
                                FunctionDescriptor function);
    MemoryAddress upcallStub(MethodHandle target,
                             FunctionDescriptor function,
                             ResourceScope scope);
}
```

For downcalls, the `downcallHandle` method takes the address of a foreign function — typically, a `MemoryAddress` obtained from a library lookup — and exposes the foreign function as a *downcall method handle*. Later, Java code invokes the downcall method handle by calling its `invokeExact` method, and the foreign function runs. Any arguments passed to the method handle's `invokeExact` method are passed on to the foreign function.

For upcalls, the `upcallStub` method takes a method handle — typically, one which refers to a Java method, rather than a downcall method handle — and converts it to a memory address. Later, the memory address is passed as an argument when Java code invokes a downcall method handle. In effect, the memory address serves as a function pointer. (For more information on upcalls, see below.)

Suppose we wish to downcall from Java to the `strlen` function defined in the standard C library:

```
size_t strlen(const char *s);
```

A downcall method handle that exposes `strlen` can be obtained as follows (the details of `MethodType` and `FunctionDescriptor` will be described shortly):

```
MethodHandle strlen = CLinker.getInstance().downcallHandle(
    CLinker.systemLookup().lookup("strlen").get(),
    MethodType.methodType(long.class, MemoryAddress.class),
    FunctionDescriptor.of(C_LONG, C_POINTER)
);
```

Invoking the downcall method handle will run `strlen` and make its result available in Java. For the argument to `strlen`, we use a helper method to convert a Java string into an off-heap memory segment and pass that segment's address:

```
MemorySegment str = CLinker.toCString("Hello", newImplicitScope());
long len          = strlen.invokeExact(str.address());  // 5
```

Method handles work well for exposing foreign functions because the JVM already optimizes the invocation of method handles all the way down to native code. When a method handle refers to a method in a `class` file, invoking the method handle typically causes the target method to be JIT-compiled; subsequently, the JVM interprets the Java bytecode that calls `MethodHandle::invokeExact` by transferring control to the assembly code generated for the target method. Thus, invoking a traditional method handle is already a quasi-foreign invocation; a downcall method handle that targets a function in a C library is just a more-foreign form of method handle. Method handles also enjoy a property called *signature*

polymorphism that allows box-free invocation with primitive arguments. In sum, method handles let the `CLinker` expose foreign functions in a natural, efficient, and extensible manner.

Describing C types in Java

To create a downcall method handle, the FFM API requires the client to provide a two-sided view of the target C function: a high-level signature using *opaque* Java objects (`MemoryAddress`, `MemorySegment`), and a low-level signature using *transparent* Java objects (`MemoryLayout`). Taking each signature in turn:

- The high-level signature, a `MethodType`, serves as the type of the downcall method handle. Every method handle is strongly typed, which means it is stringent about the number and types of the arguments that can be passed to its `invokeExact` method. For example, a method handle created to take one `MemoryAddress` argument cannot be invoked via `invokeExact(<MemoryAddress>, <MemoryAddress>)` or via `invokeExact("Hello")`. Thus, the `MethodType` describes the Java signature which clients must use when invoking the downcall method handle. It is, effectively, the Java view of the C function.
- The low-level signature, a `FunctionDescriptor`, consists of `MemoryLayout` objects. This gives the `CLinker` a precise understanding of the C function's arguments so that it can arrange them properly, as described below. Clients usually have `MemoryLayout` objects on hand in order to dereference data in foreign memory, and such objects can be reused here as foreign function signatures.

As an example, obtaining a downcall method handle for a C function which takes an `int` and returns a `long` would require the following `MethodType` and `FunctionDescriptor` arguments to `downcallHandle`:

```
MethodType mtype         = MethodType.methodType(long.class, int.class);
FunctionDescriptor fdesc = FunctionDescriptor.of(C_LONG, C_INT);
```

(This example targets Linux/x64 and macOS/x64, where the Java types `long` and `int` are associated with the predefined `CLinker` layouts `C_LONG` and `C_INT` respectively. The association of Java types with memory layouts varies by platform; for example, on Windows/x64, a Java `long` is associated with the `C_LONG_LONG` layout.)

As another example, obtaining a downcall method handle for a `void` C function which takes a pointer would require the following `MethodType` and `FunctionDescriptor`:

```
MethodType mtype         = MethodType.methodType(void.class, MemoryAddress.class);
FunctionDescriptor fdesc = FunctionDescriptor.ofVoid(C_POINTER);
```

(All pointer types in C are expressed as `MemoryAddress` objects in Java; the corresponding layout, whose size depends on the current platform, is `C_POINTER`. Clients do not distinguish between, e.g., `int*` and `char**`, because the Java types and memory layouts passed to the `CLinker` jointly contain enough information to pass Java arguments correctly to the C function.)

Finally, unlike JNI, the `CLinker` supports passing structured data to foreign functions. Obtaining a downcall method handle to a `void` C function which takes a struct would require the following `MethodType` and `FunctionDescriptor`:

```
MethodType mtype         = MethodType.methodType(void.class, MemorySegment.class);
MemoryLayout SYSTEMTIME  = MemoryLayout.ofStruct(
  C_SHORT.withName("wYear"),      C_SHORT.withName("wMonth"),
  C_SHORT.withName("wDayOfWeek"), C_SHORT.withName("wDay"),
  C_SHORT.withName("wHour"),      C_SHORT.withName("wMinute"),
  C_SHORT.withName("wSecond"),    C_SHORT.withName("wMilliseconds")
);
FunctionDescriptor fdesc = FunctionDescriptor.ofVoid(SYSTEMTIME);
```

(For the high-level `MethodType` signature, the Java client always uses the opaque type `MemorySegment` where a C function expects a struct passed by value. For the low-level `FunctionDescriptor` signature, the memory layout associated with a C struct type must be a composite layout which defines the sub-layouts for all the fields in the C struct, including padding which might be inserted by a native compiler.)

If a C function returns a by-value struct, as expressed by the low-level signature, then a fresh memory segment must be allocated off-heap and returned to the Java client. To achieve this, the method handle returned by `downcallHandle` requires an

additional `SegmentAllocator` argument, which the FFM API uses to allocate a memory segment to hold the struct returned by the C function.

Packaging Java arguments for C functions

Interoperation between different languages requires a *calling convention* to specify how code in one language invokes a function in another language, how it passes arguments, and how it receives any results. The `CLinker` implementation has knowledge of several calling conventions out-of-the-box: Linux/x64, Linux/AArch64, macOS/x64, and Windows/x64. Being written in Java, it is far easier to maintain and extend than JNI, whose calling conventions are hardwired into HotSpot's C++ code.

Consider the function descriptor shown above for the `SYSTEMTIME` struct and layout. Given the calling convention of the OS and CPU where the JVM is running, the `CLinker` uses the function descriptor to infer how the struct's fields should be passed to the C function when a downcall method handle is invoked with a `MemorySegment` argument. For one calling convention, the `CLinker` could arrange to decompose the incoming memory segment, pass the first four fields using general CPU registers, and pass the remaining fields on the C stack. For different calling convention, the `CLinker` could arrange for the FFM API to pass the struct indirectly by allocating a region of memory, bulk-copying the contents of the incoming memory segment into that region, and passing a pointer to that memory region to the C function. This lowest-level packaging of arguments happens behind the scenes, without any need for supervision by client code.

Upcalls

Sometimes it is useful to pass Java code as a function pointer to some foreign function. We can achieve that by using the `CLinker` support for upcalls. In this section we build, piece by piece, a more sophisticated example which demonstrates the full power of the `CLinker`, with full bidirectional interoperation of both code and data across the Java/native boundary.

Consider the following function defined in the standard C library:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

To call `qsort` from Java, we first need to create a downcall method handle:

```
MethodHandle qsort = CLinker.getInstance().downcallHandle(
    CLinker.systemLookup().lookup("qsort").get(),
    MethodType.methodType(void.class, MemoryAddress.class, long.class,
                          long.class, MemoryAddress.class),
    FunctionDescriptor.ofVoid(C_POINTER, C_LONG, C_LONG, C_POINTER)
);
```

As before, we use `C_LONG` and `long.class` to map the C `size_t` type, and we use `MemoryAddess.class` both for the first pointer parameter (the array pointer) and the last parameter (the function pointer).

`qsort` sorts the contents of an array using a custom comparator function, `compar`, passed as a function pointer. Therefore, to invoke the downcall method handle, we need a function pointer to pass as the last parameter to the method handle's `invokeExact` method. `CLinker::upcallStub` helps us create function pointers by using existing method handles, as follows.

First, we write a `static` method in Java that compares two `long` values, represented indirectly as `MemoryAddress` objects:

```
class Qsort {
    static int qsortCompare(MemoryAddress addr1, MemoryAddress addr2) {
        return MemoryAccess.getIntAtOffset(MemorySegment.globalNativeSegment(),
                                           addr1.toRawLongValue()) -
               MemoryAccess.getIntAtOffset(MemorySegment.globalNativeSegment(),
                                           addr2.toRawLongValue());
    }
}
```

Second, we create a method handle pointing to the Java comparator method:

```
MethodHandle comparHandle
    = MethodHandles.lookup()
                   .findStatic(Qsort.class, "qsortCompare",
                               MethodType.methodType(int.class,
                                                     MemoryAddress.class,
                                                     MemoryAddress.class));
```

Third, now that we have a method handle for our Java comparator, we can create a function pointer using `CLinker::upcallStub`. Just as for downcalls, we describe the signature of the function pointer using layouts in the `CLinker` class:

```
MemoryAddress comparFunc =
  CLinker.getInstance().upcallStub(comparHandle,
                                   FunctionDescriptor.of(C_INT,
                                                         C_POINTER,
                                                         C_POINTER),
                                   newImplicitScope());
);
```

We finally have a memory address, `comparFunc`, which points to a stub that can be used to invoke our Java comparator function, and so we now have all we need to invoke the `qsort` downcall handle:

```
MemorySegment array = MemorySegment.allocateNative(4 * 10, newImplicitScope());
array.copyFrom(MemorySegment.ofArray(new int[] { 0, 9, 3, 4, 6, 5, 1, 8, 2, 7 }));
qsort.invokeExact(array.address(), 10L, 4L, comparFunc);
int[] sorted = array.toIntArray(); // [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
```

This code creates an off-heap array, copies the contents of a Java array into it, and then passes the array to the `qsort` handle along with the comparator function we obtained from the `CLinker`. After the invocation the contents of the off-heap array will be sorted according to our comparator function, written in Java. We then extract a new Java array from the segment, which contains the sorted elements.

Safety

Fundamentally, any interaction between Java code and native code can compromise the integrity of the Java Platform. Linking to a C function in a precompiled library is inherently unreliable because the Java runtime cannot guarantee that the function's signature matches the expectations of the Java code, or even that a symbol in a C library is really a function. Moreover, if a suitable function is linked, actually calling the function can lead to low-level failures, such as segmentation faults, that end up crashing the VM. Such failures cannot be prevented by the Java runtime or caught by Java code.

Native code that uses JNI functions is especially dangerous. Such code can access JDK internals without command-line flags (e.g., `--add-opens`), by using functions such as `getStaticField` and `callVirtualMethod`. It can also change the values of `final` fields long after they are initialized. Allowing native code to bypass the checks applied to Java code undermines every boundary and assumption in the JDK. In other words, JNI is inherently unsafe.

JNI cannot be disabled, so there is no way to ensure that Java code will not call native code which uses dangerous JNI functions. This is a risk to platform integrity that is almost invisible to application developers and end users because 99% of the use of these functions is typically from third, fourth, and fifth-party libraries sandwiched between the application and the JDK.

Most of the FFM API is safe by design. Many scenarios that required the use of JNI and native code in the past can be accomplished by calling methods in the FFM API which cannot compromise the Java Platform. For example, a primary use case for JNI, flexible memory allocation, is supported with a simple method, `MemorySegment::allocateNative`, that involves no native code and always returns memory managed by the Java runtime. Generally speaking, Java code that uses the FFM API cannot crash the JVM.

Part of the FFM API, however, is inherently unsafe. When interacting with the `CLinker`, Java code can request a downcall method handle by specifying parameter types that are incompatible with those of the underlying C function. Invoking the downcall method handle in Java will result in the same kind of outcome — a VM crash, or undefined behavior — that can occur when invoking a `native` method in JNI. The FFM API can also produce unsafe segments, that is, memory segments whose spatial and temporal bounds are user-provided and cannot be verified by the Java runtime (see `MemoryAddress::asSegment`).

The unsafe methods in the FFM API do not pose the same risks as JNI functions; they cannot, e.g., change the values of `final` fields in Java objects. On the other hand, the unsafe methods in the FFM API are easy to call from Java code. For this reason the use of unsafe methods in the FFM API is *restricted:* Access to unsafe methods is disabled by default, so that invoking such methods throws an `IllegalAccessException`. To enable access to unsafe methods for code in some module `M`, specify `java --enable-native-access=M` on the command line. (Specify multiple modules in a comma-separated list; specify `ALL-UNNAMED` to

enable access for all code on the class path.) Most methods of the FFM API are safe, and Java code can use those methods regardless of whether `--enable-native-access` is given.

We do not propose here to restrict any aspect of JNI. It will still be possible to call `native` methods in Java, and for native code to call unsafe JNI functions. However, it is likely that we will restrict JNI in some way in a future release. For example, unsafe JNI functions such as `newDirectByteBuffer` may be disabled by default, just like unsafe methods in the FFM API. More broadly, the JNI mechanism is so irredeemably dangerous that we hope libraries will prefer the pure-Java FFM API for both safe and unsafe operations so that, in time, we can disable all of JNI by default. This aligns with the broader Java roadmap of making the platform safe out-of-the-box, requiring end users to opt in to unsafe activities such as breaking strong encapsulation or linking to unknown code.

We do not propose here to change `sun.misc.Unsafe` in any way. The FFM API's support for off-heap memory is an excellent alternative to the wrappers around `malloc` and `free` in `sun.misc.Unsafe`, namely `allocateMemory`, `setMemory`, `copyMemory`, and `freeMemory`. We hope that libraries and applications that require off-heap storage adopt the FFM API so that, in time, we can deprecate and then eventually remove these `sun.misc.Unsafe` methods.

Alternatives

Keep using `java.nio.ByteBuffer`, `sun.misc.Unsafe`, JNI, and other third-party frameworks.

Risks and Assumptions

Creating an API to access foreign memory in a way that 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 JIT compilers 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 API are as efficient and optimizable as uses of existing APIs such as `ByteBuffer` and `Unsafe`. The JIT implementations will also require work to ensure that uses of the native method handles retrieved from the API are at least as efficient and optimizable as uses of existing JNI native methods.

Dependencies

- The Foreign Function & Memory API can be used to access non-volatile memory, already possible via JEP 352 (Non-Volatile Mapped Byte Buffers), in a more general and efficient way.
- The work described here will likely enable subsequent work to provide a tool, `jextract`, which, starting from the header files for a given native library, mechanically generates the native method handles required to interoperate with that library. This will further reduce the overhead of using native libraries from Java.