State of foreign function support

September 2020

- Tweaked references to restricted segments to use new API
- Tweak signature of LibraryLookup::lookup
- Replaced usages of ForeignLinker with CLinker, as per new API

Maurizio Cimadamore

In this document we explore the main concepts behind Panama's foreign function support; as we shall see, the central abstraction in the foreign function support is the so called *foreign linker*, an abstraction that allows clients to construct *native* method handles — that is, method handles whose invocation targets a native function defined in some native library. As we shall see, Panama foreign function support is completely expressed in terms of Java code and no intermediate native code is required.

Native addresses

Before we dive into the specifics of the foreign function support, it would be useful to briefly recap some of the main concepts we have learned when exploring the foreign memory access support. The Foreign Memory Access API allows client to create and manipulate *memory segments*. A memory segment is a view over a memory source (either on- or off-heap) which is spatially bounded, temporally bounded and thread-confined. The guarantees ensure that dereferencing a segment that has been created by Java code is always *safe*, and can never result in a VM crash, or, worse, in silent memory corruption.

Now, in the case of memory segments, the above properties (spatial bounds, temporal bounds and confinement) can be known *in full* when the segment is created. But when we interact with native libraries we will often be receiving *raw* pointers; such pointers have no spatial bounds (does a `char*` in C refer to one `char`, or a `char` array of a given size?), no notion of temporal bounds, nor thread-confinement. Raw addresses in our interop support are modelled using the `MemoryAddress` abstraction.

A memory address is just what the name implies: it encapsulates a memory address (either on- or off-heap). Since, in order to dereference memory using a memory access var handle, we need a segment, it follows that it is *not* possible to directly dereference a memory address — to do that we need a segment first. So clients can proceed in three different ways here.

First, if the memory address is known to belong to a segment the client *already* owns, a *rebase* operation can be performed; in other words, the client can ask the address what is its offset relative to a given segment, and then proceed to dereference the original segment accordingly:

```
MemorySegment segment = MemorySegment.allocateNative(100);
...
MemoryAddress addr = ... //obtain address from native code
int x = MemoryAccess.getIntAtOffset(segment, addr.segmentOffset(segment));
```

Secondly, if the client does *not* have a segment which contains a given memory address, it can create one *unsafely*, using the `MemoryAddress::asSegmentRestricted`; this can also be useful to inject extra knowledge about spatial bounds which might be available in the native library the client is interacting with:

```
MemoryAddress addr = ... //obtain address from native code
MemorySegment segment = addr.asSegmentRestricted(100);
int x = MemoryAccess.getInt(segment);
```

Alternatively, the client can fall back to use the so called *everything* segment - that is, a primordial segment which covers the entire native heap. Since this segment is available as a constant, dereference can happen without the need of creating any additional segment instances:

```
MemoryAddress addr = ... //obtain address from native code
int x = MemoryAccess.getIntAtOffset(MemorySegment.ofNativeRestricted(), addr.toRawLongValue());
```

Of course, since accessing the entire native heap is inherently *unsafe*, accessing the *everything* segment is considered a *restricted* operation which is only allowed after explicit opt in by setting the `foreign.restricted=permit` runtime flag.

`MemoryAddress`, like `MemorySegment` , implements the `Addressable` interface, which is a functional interface whose method projects an entity down to a `MemoryAddress` instance. In the case of `MemoryAddress` such a projection is the identity function; in the case of a memory segment, the projection returns the `MemoryAddres` instance for the segment's base address. This abstraction allows to pass either memory address or memory segments where an address is expected (this is especially useful when generating native bindings).

Symbol lookups

The first ingredient of any foreign function support is a mechanism to lookup symbols in native libraries. In traditional Java/JNI, this is done via the `System::loadLibrary` and `System::load` methods, which internally map into calls to `dlopen`. In Panama, library lookups are modeled more directly, using a class called`LibraryLookup` (similar to a method handle lookup), which provides capabilities to lookup named symbols in a given native library; we can obtain a library lookup in 3 different ways:

- `LibraryLookup::ofDefault` — returns the library lookup which can *see* all the symbols that have been loaded with the VM
- `LibraryLookup::ofPath` — creates a library lookup associated with the library found at the given absolute path
- `LibraryLookup::ofLibrary` — creates a library lookup associated with the library with given name (this might require setting the `java.library.path` variable accordingly)

Once a lookup has been obtained, a client can use it to retrieve handles to library symbols (either global variables or functions) using the `lookup(String)` method, which returns an `Optional<LibraryLookup.Symbol>`. A lookup symbol is just a proxy for a memory address (in fact, it implements `Addressable`) and a name.

For instance, the following code can be used to lookup the `clang_getClangVersion` function provided by the `clang` library:

```
LibraryLookup libclang = LibraryLookup.ofLibrary("clang");
LibraryLookup.Symbol clangVersion = libclang.lookup("clang_getClangVersion").get();
```

One crucial distinction between the library loading support of the Foreign Linker API and of JNI is that JNI libraries are loaded into a class loader. Furthermore, to preserve classloader integrity integrity, the same JNI library cannot be loaded into more than one classloader. The foreign function support described here is more primitive — the Foreign Linker API allows clients to target native libraries directly, without any intervening JNI code. Crucially, Java objects are never passed to and from native code by the Foreign Linker API. Because of this, libraries loaded through the `LibraryLookup` hook are not tied to any class loader and can be (re)loaded as many times as needed.

C Linker

At the core of Panama foreign function support we find the `CLinker` abstraction. This abstraction plays a dual role: first, for downcalls, it allows to model native function calls as plain `MethodHandle` calls (see `ForeignLinker::downcallHandle`); second, for upcalls, it allows to convert an existing `MethodHandle` (which might point to some Java method) into a `MemorySegment` which could then be passed to native functions as a function pointer (see `ForeignLinker::upcallStub`):

```
interface CLinker {
   MethodHandle downcallHandle(Addressable func, MethodType type, FunctionDescriptor function);
   MemorySegment upcallStub(MethodHandle target, FunctionDescriptor function);
   ...
   static CLinker getInstance() { ... }
}
```

In the following sections we will dive deeper into how downcall handles and upcall stubs are created; here we want to focus on the similarities between these two routines. First, both take a `FunctionDescriptor` instance — essentially an aggregate of memory layouts which is used to describe the signature of a foreign function in full. Speaking of C, the `CLinker` class defines many layout constants (one for each main C primitive type) — these layouts can be combined using a `FunctionDescriptor` to describe the signature of a C function. For instance, assuming we have a C function taking a `char*` and returning a `long` we can model such a function with the following descriptor:

```
FunctionDescriptor func = FunctionDescriptor.of(CLinker.C_LONG, CLinker.C_POINTER);
```

The layouts used above will be mapped to the right layout according to the platform we are executing on. This also means that these layouts will be platform dependent and that e.g. `C_LONG` will be a 32 bit value layout on Windows, while being a 64-bit value on Linux.

Layouts defined in the `CLinker` class are not only handy, as they already model the C types we want to work on; they also contain hidden pieces of information which the foreign linker support uses in order to compute the calling sequence associated with a given function descriptor. For instance, the two C types `int` and `float` might share a similar memory layout (they both are expressed as 32 bit values), but are typically passed using different machine registers. The layout attributes attached to the C-specific layouts in the `CLinker` class ensures that arguments and return values are interpreted in the correct way.

Another similarity between `downcallHandle` and `upcallStub` is that they both accept (either directly, or indirectly) a `MethodType` instance. The method type describes the Java signatures that clients will be using when interacting with said downcall handles, or upcall stubs. The C linker implementation adds constraints on which layouts can be used with which Java carrier — for instance by enforcing that the size of the Java carrier is equal to that of the corresponding layout, or by making sure that certain layouts are associated with specific carriers. The following table shows the Java carrier vs. layout mappings enforced by the Linux/macOS foreign linker implementation:

C layout	Java carrier
`C_BOOL`	`byte`
`C_CHAR`	`byte`
`C_SHORT`	`short`
`C_INT`	`int`
`C_LONG`	`long`
`C_LONGLONG`	`long`
`C_FLOAT`	`float`
`C_DOUBLE`	`double`
`C_POINTER`	`MemoryAddress`
`GroupLayout`	`MemorySegment`

Aside from the mapping between primitive layout and primitive Java carriers (which might vary across platforms), it is important to note how all pointer layouts must correspond to a `MemoryAddress` carrier, whereas structs (whose layout is defined by a `GroupLayout`) must be associated with a `MemorySegment` carrier.

Downcalls

We will now look at how foreign functions can be called from Java using the foreign linker abstraction. Assume we wanted to call the following function from the standard C library:

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

In order to do that, we have to:

- lookup the `strlen` symbol
- describe the signature of the C function using the layouts in the `CLinker` class
- select a Java signature we want to *overlay* on the native function — this will be the signature that clients of the native method handles will interact with
- create a *downcall* native method handle with the above information, using the standard C foreign linker

Here's an example of how we might want to do that (a full listing of all the examples in this and subsequent sections will be provided in the appendix):

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

Note that, since the function `strlen` is part of the standard C library, which is loaded with the VM, we can just use the default lookup to look it up. The rest is pretty straightforward — the only tricky detail is how to model `size_t`: typically this type has the size of a pointer, so we can use `C_LONG` on Linux, but we'd have to use `C_LONGLONG` on Windows. On the Java side, we model the `size_t` using a `long` and the pointer is modeled using a `MemoryAddress` parameter.

One we have obtained the downcall native method handle, we can just use it as any other method handle:

```
long len = strlen.invokeExact(CLinker.toCString("Hello").address()) // 5
```

Here we are using one of the helper methods in `CLinker` to convert a Java string into an off-heap memory segment which contains a `NULL` terminated C string. We then pass that segment to the method handle and retrieve our result in a Java `long`. Note how all this has been possible *without* any piece of intervening native code — all the interop code can be expressed in (low level) Java.

Now that we have seen the basics of how foreign function calls are supported in Panama, let's add some additional considerations. First, it is important to note that, albeit the interop code is written in Java, the above code can *not* be considered 100% safe. There are many arbitrary decisions to be made when setting up downcall method handles such as the one above, some of which might be obvious to us (e.g. how many parameters does the function take), but which cannot ultimately be verified by the Panama runtime. After all, a symbol in a dynamic library is, mostly a numeric offset and, unless we are using a shared library with debugging information, no type information is attached to a given library symbol. This means that, in this case, the Panama runtime has to *trust* our description of the `strlen` function. For this reason, access to the foreign linker is a restricted operation, which can only be performed if the runtime flag `foreign.restricted=permit` is passed on the command line of the Java launcher [1].

Finally let's talk about the life-cycle of some of the entities involved here; first, as a downcall native handle wraps a lookup symbol, the library from which the symbol has been loaded will stay loaded until there are reachable downcall handles referring to one of its symbols; in the above example, this consideration is less important, given the use of the default lookup object, which can be assumed to stay alive for the entire duration of the application.

Certain functions might return pointers, or structs; it is important to realize that if a function returns a pointer (or a `MemoryAddress`), no life-cycle whatsoever is attached to that pointer. It is then up to the client to e.g. free the memory associated with that pointer, or do nothing (in case the library is responsible for the life-cycle of that pointer). If a library returns a struct by value, things are different, as a *fresh*, confined memory segment is allocated off-heap and returned to the callee. It is the responsibility of the callee to cleanup that struct's segment (using `MemorySegment::close`) [2].

Performance-wise, the reader might ask how efficient calling a foreign function using a native method handle is; the answer is *very*. The JVM comes with some special support for native method handles, so that, if a give method handle is invoked many times (e.g, inside an *hot* loop), the JIT compiler might decide to just generate a snippet of assembly code required to call the native function, and execute that directly. In most cases, invoking native function this way is as efficient as doing so through JNI[34].

Upcalls

Sometimes, it is useful to pass Java code as a function pointer to some native function; we can achieve that by using foreign linker support for upcalls. To demonstrate this, let's consider the following function from the C standard library:

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

This is a function that can be used to sort the contents of an array, using a custom comparator function — `compar` — which is passed as a function pointer. To be able to call the `qsort` function from Java we have first to create a downcall native method handle for it:

```
MethodHandle qsort = CLinker.getInstance().downcallHandle(
        LibraryLookup.ofDefault().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).

This time, in order to invoke the `qsort` downcall handle, we need a *function pointer* to be passed as the last parameter; this is where the upcall support in foreign linker comes in handy, as it allows us to create a function pointer out of an existing method handle. First, let's write a function that can compare two int elements (passed as pointers):

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

Here we can see that the function is performing some *unsafe* dereference of the pointer contents, by using the *everything* segment. Now let's create a method handle pointing to the comparator function above:

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

Now that we have a method handle for our Java comparator function, we can create a function pointer, using the foreign linker upcall support — as for downcalls, we have to describe the signature of the foreign function pointer using the layouts in the `CLinker` class:

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

So, we finally have a memory segment — `comparFunc` — containing a stub that can be used to invoke our Java comparator function; this means we now have all we need to call the `qsort` downcall handle:

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

The above code creates an off-heap array, then copies the contents of a Java array on it (we shall see in the next section ways to do that more succinctly), and then pass the array to the `qsort` handle, along with the comparator function we obtained from the foreign linker. As a side-effect, after the call, the contents of the off-heap array will be sorted (as instructed by our comparator function, written in Java). We can than extract a new Java array from the segment, which contains the sorted elements. This is a more advanced example, but one that shows how powerful the native interop support provided by the foreign linker abstraction is, allowing full bidirectional interop support between Java and native.

As before, we conclude with a quick note on life-cycle. First, the life-cycle of the upcall stub is tied to that of the segment returned by the foreign linker. When the segment is closed, the upcall is uninstalled from the VM and will no longer be a valid function pointer. Second, the life-cycle of structs (if any) passed by value to the Java upcall function is independent from that of the upcall [5], so again the user will have to pay attention not to leak memory and to call `MemorySegment::close` on any segments obtained through an upcall.

Native scope

Idiomatic C code implicitly relies on stack allocation to allow for concise variable declarations; consider this example:

```
void foo(int i, int *size);

int size = 5;
foo(42, &size);
```

Here the function `foo` takes an output parameter, a pointer to an `int` variable. Unfortunately (and we have seen part of that pain in the `qsort` example above), implementing this idiom in Panama is not straightforward:

```
MethodHandle foo = ...
MemorySegment size = MemorySegment.allocateNative(C_INT);
MemoryAccess.setInt(size, 5);
foo.invokeExact(42, size);
```

There are a number of issues with the above code snippet:

- compared to the C code, it is very verbose
- allocation is very slow compared to C; allocating the `size` variable now takes a full `malloc`, while in C the variable was simply stack-allocated
- we end up with a standalone segment with its own temporal bounds, which will have to be *closed* later, to avoid leaks

To address these problems, Panama provides a `NativeScope` abstraction, which can be used to group allocation so as to achieve superior allocation performance, but also so that groups of logically-related segments can share the same temporal bounds, and can therefore be closed at once (e.g. by using a single *try-with-resources* statement). With the help of native scopes, the above code can be rewritten as follows:

```
try (NativeScope scope = NativeScope.unboundedScope()) {
   MemorySegment size = scope.allocate(C_INT, 5);
   foo.invokeExact(42, size);
}
```

It's easy to see how this code improves over the former in many ways; first, native scopes have primitives to allocate *and* initialize the contents of a segment; secondly, native scope use more efficient allocation underneath, so that not every allocation request is turned into a `malloc` — in fact, if the size of memory to be used is known before hand, clients can also use the bounded variant of native scope, using the `NativeScope::boundedScope(long size)` factory [6]. Third, a native scope can be used as a single temporal bound for all the segments allocated within it: that is, if the code needs to instantiate other variables, it can keep doing so using the same scope — when the *try-with-resource* statement completes, all resources associated with the scope will be freed.

There are at least two cases where allocation of native resources occurs *outside* a native scope:

- structs segments returned by native calls (or passed to upcalls Java methods)
- upcall stubs segments returned by the foreign linker

In these cases, the API gives us a segment which feature *its own* temporal bounds — this is handy, as we can use the segment to control the lifecycle of the resource allocated by the foreign linker support; but it is also a bit unfortunate: if the surrounding code happens to already have a native scope, these new segments won't be able to interoperate with it, and a separate *try-with-resource* segment should be used.

To alleviate this problem, and allow *all* segments to be managed using the *same* native scope, the native scope API not only supports the ability to allocate *new* segments, but it also allows to take ownership of *existing* ones. To see an example of this, let's go back to our `qsort` example, and see how we can make it better by using native scopes:

```
try (NativeScope scope = NativeScope.unboundedScope()) {
   comparFunc = scope.register(comparFunc);
   MemorySegment array = scope.allocateArray(C_INT, new int[] { 0, 9, 3, 4, 6, 5, 1, 8, 2, 7 });
   qsort.invokeExact(array, 10L, 4L, comparFunc);
   int[] sorted = array.toIntArray(); // [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
}
```

Not only native scope helps in making the allocation of the native array simpler (we no longer need to create an heap segment, and dump its contents onto the off-heap array); but we can also use the native scope to *register* the existing upcall stub segments. When we do that, we obtain a *new* segment, whose temporal bounds are the same as that of the native scope; the old segment will be killed and will no longer be usable. As with all segments returned by native scope, the registered segment we get back will be non-closeable — the only way to close it is to close the native scope it belongs to.

In short, native scopes are a good way to manage the lifecycle of multiple, logically-related segments, and, despite their simplicity, they provide a fairly good usability and performance boost.

Varargs

Some C functions are *variadic* and can take an arbitrary number of arguments. Perhaps the most common example of this is the `printf` function, defined in the C standard library:

```
int printf(const char *format, ...);
```

This function takes a format string, which features zero or more *holes*, and then can take a number of additional arguments that is identical to the number of holes in the format string.

The foreign function support can support variadic calls, but with a caveat: the client must provide a specialized Java signature, and a specialized description of the C signature. For instance, let's say we wanted to model the following C call:

```
printf("%d plus %d equals %d", 2, 2, 4);
```

To do this using the foreign function support provided by Panama we would have to build a *specialized* downcall handle for that call shape [7]:

```
MethodHandle printf = CLinker.getInstance().downcallHandle(
        LibraryLookup.ofDefault().lookup("printf").get(),
        MethodType.methodType(int.class, MemoryAddress.class, int.class, int.class, int.class),
        FunctionDescriptor.of(C_INT, C_POINTER, C_INT, C_INT, C_INT)
);
```

Then we can call the specialized downcall handle as usual:

```
printf.invoke(CLinker.toCString("%d plus %d equals %d").address(), 2, 2, 4); //prints "2 plus 2 equals 4"
```

While this works, it is easy to see how such an approach is not very desirable:

- If the variadic function needs to be called with many different shapes, we have to create many different downcall handles
- while this approach works for downcalls (since the Java code is in charge of determining which and how many arguments should be passed) it fails to scale to upcalls; in that case, the call comes from native code, so we have no way to guarantee that the shape of the upcall stub we have created will match that required by the native function.

To mitigate these issues, the standard C foreign linker comes equipped with support for C variable argument lists — or `va_list`. When a variadic function is called, C code has to unpack the variadic arguments by creating a `va_list` structure, and then accessing the variadic arguments through the `va_list` one by one (using the `va_arg` macro). To facilitate interop between standard variadic functions and `va_list` many C library functions in fact define *two* flavors of the same function, one using standard variadic signature, one using an extra `va_list` parameter. For instance, in the case of `printf` we can find that a `va_list`-accepting function performing the same task is also defined:

```
int vprintf(const char *format, va_list ap);
```

The behavior of this function is the same as before — the only difference is that the ellipsis notation `...` has been replaced with a single `va_list` parameter; in other words, the function is no longer variadic.

It is indeed fairly easy to create a downcall for `vprintf`:

```
MethodHandle vprintf = CLinker.getInstance().downcallHandle(
    LibraryLookup.ofDefault().lookup("vprintf").get(),
    MethodType.methodType(int.class, MemoryAddress.class, VaList.class),
    FunctionDescriptor.of(C_INT, C_POINTER, C_VA_LIST));
```

Here, the notable thing is that `CLinker` comes equipped with the special `C_VA_LIST` layout (the layout of a `va_list` parameter) as well as a `VaList` carrier, which can be used to construct and represent variable argument lists from Java code.

To call the `vprintf` handle we need to create an instance of `VaList` which contains the arguments we want to pass to the `vprintf` function — we can do so, as follows:

```
vprintf.invoke(
    CLinker.toCString("%d plus %d equals %d").address(),
    VaList.make(builder ->
            builder.vargFromInt(C_INT, 2)
                .vargFromInt(C_INT, 2)
                .vargFromInt(C_INT, 4))
); //prints "2 plus 2 equals 4"
```

While the callee has to do more work to call the `vprintf` handle, note that that now we're back in a place where the downcall handle `vprintf` can be shared across multiple callees. A `VaList` object created this way has its own lifetime (`VaList` also supports a `close` operation), so it is up to the callee to make sure that the `VaList` instance is closed (or attached to some existing native scope) so as to avoid leaks.

Another advantage of using `VaList` is that this approach also scales to upcall stubs — it is therefore possible for clients to create upcalls stubs which take a `VaList` and then, from the Java upcall, read the arguments packed inside the `VaList` one by one using the methods provided by the `VaList` AP(e.g. `VaList::vargAsInt(MemoryLayout)`), which mimic the behavior of the C `va_arg` macro.

Appendix: full source code

The full source code containing most of the code shown throughout this document can be seen below:

```
import jdk.incubator.foreign.Addressable;
import jdk.incubator.foreign.CLinker;
import jdk.incubator.foreign.FunctionDescriptor;
import jdk.incubator.foreign.LibraryLookup;
import jdk.incubator.foreign.MemoryAccess;
import jdk.incubator.foreign.MemoryAddress;
import jdk.incubator.foreign.MemorySegment;
import jdk.incubator.foreign.NativeScope;

import java.lang.invoke.MethodHandle;
import java.lang.invoke.MethodHandles;
import java.lang.invoke.MethodType;
import java.util.Arrays;

import static jdk.incubator.foreign.CLinker.*;

public class Examples {

  public static void main(String[] args) throws Throwable {
    strlen();
    qsort();
    printf();
    vprintf();
  }

  public static void strlen() throws Throwable {
    MethodHandle strlen = CLinker.getInstance().downcallHandle(
        LibraryLookup.ofDefault().lookup("strlen").get(),
        MethodType.methodType(long.class, MemoryAddress.class),
        FunctionDescriptor.of(C_LONG, C_POINTER)
    );

    try (MemorySegment hello = CLinker.toCString("Hello")) {
      long len = (long) strlen.invokeExact(hello.address()); // 5
      System.out.println(len);
    }
  }

  static class Qsort {
    static int qsortCompare(MemoryAddress addr1, MemoryAddress addr2) {
      int v1 = MemoryAccess.getIntAtOffset(MemorySegment.ofNativeRestricted(),
addr1.toRawLongValue());
      int v2 = MemoryAccess.getIntAtOffset(MemorySegment.ofNativeRestricted(),
addr2.toRawLongValue());
      return v1 - v2;
    }
  }

  public static void qsort() throws Throwable {
    MethodHandle qsort = CLinker.getInstance().downcallHandle(
        LibraryLookup.ofDefault().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)
    );

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

    MemorySegment comparFunc = CLinker.getInstance().upcallStub(
        comparHandle,
        FunctionDescriptor.of(C_INT, C_POINTER, C_POINTER)
    );

    try (NativeScope scope = NativeScope.unboundedScope()) {
      comparFunc = scope.register(comparFunc);
      MemorySegment array = scope.allocateArray(C_INT, new int[] { 0, 9, 3, 4, 6, 5, 1, 8, 2, 7 });
      qsort.invokeExact(array.address(), 10L, 4L, comparFunc.address());
      int[] sorted = array.toIntArray(); // [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
      System.out.println(Arrays.toString(sorted));
    }
  }

  public static void printf() throws Throwable {
    MethodHandle printf = CLinker.getInstance().downcallHandle(
        LibraryLookup.ofDefault().lookup("printf").get(),
        MethodType.methodType(int.class, MemoryAddress.class, int.class, int.class, int.class),
        FunctionDescriptor.of(C_INT, C_POINTER, C_INT, C_INT, C_INT)
    );
    try (MemorySegment s = CLinker.toCString("%d plus %d equals %d\n")) {
      printf.invoke(s.address(), 2, 2, 4);
    }
  }
  public static void vprintf() throws Throwable {

    MethodHandle vprintf = CLinker.getInstance().downcallHandle(
        LibraryLookup.ofDefault().lookup("vprintf").get(),
        MethodType.methodType(int.class, MemoryAddress.class, CLinker.VaList.class),
        FunctionDescriptor.of(C_INT, C_POINTER, C_VA_LIST));

    try (NativeScope scope = NativeScope.unboundedScope()) {
      MemorySegment s = CLinker.toCString("%d plus %d equals %d\n", scope);
      CLinker.VaList vlist = CLinker.VaList.make(builder ->
          builder.vargFromInt(C_INT, 2)
              .vargFromInt(C_INT, 2)
              .vargFromInt(C_INT, 4), scope);
      vprintf.invoke(s.address(), vlist);
    }
  }
}
```

---

1 In reality this is not entirely new; even in JNI, when you call a `native` method the VM trusts that the corresponding implementing function in C will feature compatible parameter types and return values; if not a crash might occur. ↩

2 In the fututre we might consider knobs to allow structs returned by value to be allocated on-heap rather than off-heap. If these structs are always passed back and forth in an opaque manner, there could be a significant performance advantage in avoiding an off-heap allocation. ↩

3 At the time of writing, support for native method intrinsics has been disabled by default due to some spurious VM crash being detected when running `jextract` with the intrinsics support enabled. While we work to rectify this situation, the intrinsics support can still be enabled using the `-Djdk.internal.foreign.ProgrammableInvoker.USE_INTRINSICS=true` flag. ↩

4 As an advanced option, Panama allows the user to opt-in to remove Java to native thread transitions; while, in the general case it is unsafe doing so (removing thread transitions could have a negative impact on GC for long running native functions, and could crash the VM if the downcall needs to pop back out in Java, e.g. via an upcall), greater efficiency can be achieved; performance sensitive users should consider this option at least for the functions that are called more frquently, assuming that these functions are *leaf* functions (e.g. do not go back to Java via an upcall) and are relatively short-lived. ↩

5 This might change in the future, as we might want to tie the lifecycle of structs created for an upcall to the lifecycle of the upcall itself so that e.g. any segment that are created ahead of calling a Java upcall, are released immediately after the upcall returns ↩

6 We are currently investigating alternate allocation strategies to make allocation inside native scopes even faster ↩

7 On Windows, layouts for variadic arguments have to be adjusted using the `CLinker.Win64.asVarArg(ValueLayout)`; this is necessay because the Windows AB passes variadic arguments using different rules than the ones used for ordinary arguments. ↩

State of foreign memory support

September 2020

- Updated section on restricted segments reflecting new, simpler API
- Revamped section on confinement to add description of shared segments
- Added section on implicit deallocation

Maurizio Cimadamore

A crucial part of any native interop story lies in the ability of accessing off-heap memory in an efficient fashion. Panama achieves this goal through the so called Foreign Memory Access API. This API has been made available as an incubating API in Java 14 ad 15, and is, to date, the most mature part of the Panama interop story.

Segments

Memory segments are abstractions which can be used to model memory regions, located either on- or off- the Java heap. Segments can be allocated from native memory (e.g. like a `malloc`), or can be wrapped around existing memory sources (e.g. a Java array or a `ByteBuffer`). Memory segments provide *strong* guarantees which make memory dereference operation *safe*. More specifically, each segment provides:

- *spatial bounds* - that is, a segment has a base address and a size, and accessing a segment outside those boundaries is forbidden
- *temporal bounds* - that is, a segment has a *state* - meaning that it can be used and then *closed* when the memory backing the segment is no longer needed (note, this might trigger deallocation of said memory)
- *thread-confinement* - that is, a segment is a view over a memory region that is *owned* by the thread which created it. Attempting to dereference or close a segment outside the confinement thread is forbidden (this is crucial to avoid access vs. close races in multi-threaded scenario)

For instance, the following snippet allocates 100 bytes off-heap:

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

Since segments are `AutoCloseable` they can be used inside a *try-with-resources* statement, which helps ensure that memory will be released when the segment is no longer needed.

Memory segments support *slicing* — that is, given a segment, it is possible to create a new segment whose spatial bounds are stricter than that of the original segment:

```
MemorySegment segment = MemorySement.allocateNative(10);
MemorySegment slice = segment.asSlice(4, 4);
```

The above code creates a slice that starts at offset 4 and has a length of 4 bytes. Slices have the *same* temporal bounds as the parent segment - this means that when the parent segment is closed, all the slices derived from it are also closed. The opposite is also true, closing a slice closes the parent segment (and all the other slices derived from it). If a library wants to share a slice, but prevent a client from closing it (could be useful e.g. when implementing a slab allocator), the library could prevent a client from closing the slice by creating a *non-closeable* view:

```
MemorySegment sharedSlice = slice.withAccessModes(ALL_ACCESS & ~CLOSE);
```

Any attempt to call `close` on `sharedSlice` will be met with an exception. Memory segments support various access modes (including those for read/write access) which can be used to constrain the set of operations available to clients.

Memory access handles

Dereferencing memory associated with segments is made possible by using *memory access var handles*. A memory access var handle is a special kind of var handle which takes a memory segment access coordinate, together with a byte offset — the offset, relative to the segment's base address at which the dereference operation should occur. A memory access var handle can be obtained as follows:

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

To create a dereference handle we have to specify a carrier type — the type we want to use e.g. to extract values from memory, as well as to whether any byte swapping should be applied when contents are read from or stored to memory. Additionally, the user can supply an extra alignment parameter (not shown here) — this can be useful to impose additional constraints on how memory dereferences should occur; for instance, a client might want to prevent access to misaligned 32 bit values.

For instance, to read 10 int values from a segment, we can write the following code:

```
MemorySegment segment = ...
int[] values = new int[10];
for (int i = 0 ; i < values.length ; i++) {
   values[i] = (int)intHandle.get(segment, (long)i * 4);
}
```

Memory access var handles (as any other var handle) are *strongly* typed; and to get maximum efficiency, it is generally necessary to introduce casts to make sure that the access coordinates match the expected types — in this case we have to cast `i * 4` into a long; similarly, since the signature polymorphic method `VarHandle::get` notionally returns `Object` a cast is necessary to force the right return type the var handle operation.

Note that, since the offset of the dereference operation is expressed in bytes, we have to manually compute the starting offset, by multiplying the logical index `i` by `4` — the size (in bytes) of a Java int value; this is not dissimilar to what happens with the `ByteBuffer` absolute get/put methods. We shall see later how memory layouts help us achieving higher level, structured access.

Safety

The memory access API provides basic safety guarantees for memory dereference operations. More specifically, dereferencing memory should either succeed, or result in a runtime exception - but, crucially, should never result in a VM crash, or, more subtly, in memory corruption occurring *outside* the region of memory associated with a memory segment.

To enforce this, strong spatial and temporal checks are enforced upon every access. Consider the following code:

```
MemorySegment segment = MemorySegment.allocateNative(10);
intHandle.get(segment, 8); //throws ISE
```

The above code leads to a runtime exception, as we trying to access memory outside the segment's bounds — the access operation starts at offset 8 (which is within bounds), but ends at offset 11 (which is outside bounds).

Similarly, attempting to access an already closed segment leads to a failure:

```
segment.close();
intHandle.get(segment, 0); //throws ISE
```

This time, the access occurs within the spatial bounds implied by the segment, but when access occurs, the segment has already been *closed*, so the access operation fails. This is crucial to guarantee safety: since memory segments ensure *deterministic deallocation*, the above code might end up attempting to dereference already freed memory.

On top of basic spatial and temporal bound guarantees, memory segments also enforce thread-confinement guarantees, which will be discussed in a later section. Note that, while these checks might seem expensive when considered in isolation, the Foreign Memory Access API is designed and implemented such that the JIT compiler can hoist most, if not all, such checks outside hot loops. Hence, memory access efficiency is not negatively impacted by the safety requirements of the API.

Layouts

Expressing byte offsets (as in the above example) can lead to code that is hard to read, and very fragile — as memory layout invariants are captured, implicitly, in the constants used to scale offsets. To address this issue, we add a *memory layout* API which allows clients to define memory layouts *programmatically*. For instance, the layout of the array used in the above example can be expressed using the following code:

```
MemoryLayout intArray = MemoryLayout.ofSequence(10, MemoryLayout.ofValueBits(32));
```

That is, our layout is a repetition of 10 elements whose size is 32 bit each. The advantage of defining a memory layout upfront, using an API, is that we can then query on the layout — for instance we can compute the offset of the 3rd element of the array:

```
long element3 = intArray.byteOffset(PathElement.sequenceElement(3)); // 12
```

To specify which nested layout element should be used for the offset calculation we use a so called *layout path* - that is, a selection expression that navigates the layout, from the *root* layout, down to the leaf layout we wish to select (in this case the 3rd layout element in the sequence).

Layouts can also be used to obtain memory access var handles; so we can rewrite the above example as follows:

```
MemorySegment segment = ...
int[] values = new int[10];
VarHandle elemHandle = intArray.varHandle(int.class, PathElement.sequenceElement());
for (int i = 0 ; i < values.length ; i++) {
   values[i] = (int)elemHandle.get(segment, (long)i);
}
```

In the above, `elemHandle` is a var handle whose type is `int` , which takes two access coordinates:

1. a `MemorySegment` instance; the segment whose memory should be dereferenced
2. a *logical* index, which is used to select the element of the sequence we want to access

In other words, manual offset computation is no longer needed — offsets and strides can in fact be derived from the layout object.

Memory layouts shine when structured access is needed — consider the following C declaration:

```
typedef struct {
  char kind;
  int value;
} TaggedValues[5];
```

The above C declaration can be modeled using the layout below:

```
SequenceLayout taggedValues = MemoryLayout.ofSequence(5,
  MemoryLayout.ofStruct(
    MemoryLayout.ofValueBits(8, ByteOrder.nativeOrder()).withName("kind"),
    MemoryLayout.ofPaddingBits(24),
    MemoryLayout.ofValueBits(32, ByteOrder.nativeOrder()).withName("value")
  )
).withName("TaggedValues");
```

Here we assume that we need to insert some padding after the `kind` field to honor the alignment requirements of the `value` field [1]. Now, if we had to access the `value` field of all the elements of the array using manual offset computation, the code would quickly become pretty hard to read — on each iteration we would have to remember that the stride of the array is 8 bytes and that the offset of the `value` field relative to the `TaggedValue` struct is 4 bytes. This gives us an access expression like `(i * 8) + 4`, where `i` is the index of the element whose `value` field needs to be accessed.

With memory layouts, we can simply compute, once and for all, the memory access var handle to access the `value` field inside the sequence, as follows:

```
VarHandle valuesHandle = taggedValues.varHandle(int.class,
                                       PathElement.sequenceElement(),
                                       PathElement.groupElement("value"));
```

When using this var handle, no manual offset computation will be required: the resulting `valuesHandle` will feature an additional `long` coordinate which can be used to select the desired `value` field from the sequence.

Var handle combinators

The attentive reader might have noted how rich the var handles returned by the layout API are, compared to the simple memory access var handle we have seen at play here. How do we go from a simple access var handle that takes a byte offset to a var handle that can dereference a complex layout path? The answer is, by using var handle *combinators*. Developers familiar with the method handle API know how simpler method handles can be combined into more complex ones using the various combinator methods in the `MethodHandles` API. These methods allow, for instance, to insert (or bind) arguments into a target method handle, filter return values, permute arguments and much more.

Sadly, none of these features are available when working with var handles. The Foreign Memory Access API rectifies this, by adding a rich set of var handle combinators in the `MemoryHandles` class; with these tools, developers can express var handle transformations such as:

- mapping a var handle carrier type into a different one, using an embedding/projection method handle pairs
- filter one or more var handle access coordinates using unary filters
- permute var handle access coordinates
- bind concrete access coordinates to an existing var handle

Without diving too deep, let's consider how we might want to take a basic memory access handle and turn it into a var handle which dereference a segment at a specific offset (again using the `taggedValues` layout defined previously):

```
VarHandle intHandle = MemoryHandles.varHandle(int.class, ByteOrder.nativeOrder()); // (MS, J) -> I
long offsetOfValue = taggedValues.byteOffset(PathElement.sequenceElement(0),
                                    PathElement.groupElement("value"));
VarHandle valueHandle = MemoryHandles.insertCoordinates(intHandle, 0, offsetOfValue); // (MS) -> I
```

We have been able to derive, from a basic memory access var handle, a new var handle that dereferences a segment at a given fixed offset. It is easy to see how other, richer, var handles obtained using the layout API can be constructed manually using the var handle combinator API.

Segment accessors

Building complex memory access var handles using layout paths and the combinator API is useful, especially for structured access. But in simple cases, creating a `VarHandle` just to be able to read an int value at a given segment offset can be perceived as overkill. For this reason, the foreign memory access API provides ready-made static accessors in the `MemoryAccess` class, which allows to dereference a segment in various ways. For instance, if a client wants to read an int value from a segment, one of the following methods can be used:

- `MemoryAccess::getInt(MemorySegment)` — reads an int value (4 bytes) starting at the segment's base address
- `MemoryAccess::getIntAtOffset(MemorySegment, long)` — reads an int value (4 bytes) starting at the address `A = B + O` where `B` is the segment's base address, and `O` is an offset (in bytes) supplied by the client
- `MemoryAccess::getIntAtIndex(MemorySegment, long)` — reads an int value (4 bytes) starting at the address `A = B + (4 * I)` where `B` is the segment's base address, and `I` is a logical index supplied by the client; this accessor is useful for mimicking array access.

In other words, at least in simple cases, memory dereference operations can be achieved without the need of going through the `VarHandle` API; of course in more complex cases (structured and/or multidimensional access, fenced access) the full power of the `VarHandle` API might still come in handy.

Interoperability

Memory segments are pretty flexible when it comes to interacting with existing memory sources. For instance it is possible to:

- create segment from a Java array
- convert a segment into a Java array
- create a segment from a byte buffer
- convert a segment into a byte buffer

For instance, thanks to bi-directional integration with the byte buffer API, it is possible for users to create a memory segment, and then de-reference it using the byte buffer API, as follows:

```
MemorySegment segment = ...
int[] values = new int[10];
ByteBuffer bb = segment.asByteBuffer();
for (int i = 0 ; i < values.length ; i++) {
   values[i] = bb.getInt();
}
```

The only thing to remember is that, when a byte buffer view is created out of a memory segment, the buffer has the same temporal bound and thread-confinement guarantees as those of the segment it originated from. This means that if the segment is closed, any subsequent attempt to dereference its memory via a (previously obtained) byte buffer view will fail with an exception.

Unsafe segments

It is sometimes necessary to create a segment out of an existing memory source, which might be managed by native code. This is the case, for instance, if we want to create a segment out of memory managed by a custom allocator.

The ByteBuffer API allows such a move, through a JNI method, namely `NewDirectByteBuffer`. This native method can be used to wrap a long address in a fresh byte buffer instance which is then returned to unsuspecting Java code.

Memory segments provide a similar capability - that is, given an address (which might have been obtained through some native calls), it is possible to wrap a segment around it, with given spatial, temporal and confinement bounds; a cleanup action to be executed when the segment is closed might also be specified.

For instance, assuming we have an address pointing at some externally managed memory block, we can construct an *unsafe* segment, as follows:

```
MemoryAddress addr = MemoryAddress.ofLong(someLongAddr);
var unsafeSegment = addr.asSegmentRestricted(10);
```

The above code creates a new confined unsafe segment from a given address; the size of the segment is 10 bytes; the confinement thread is the current thread, and there's no cleanup action associated with the segment (that can be changed as needed by calling `MemorySegment::withCleanupAction`).

Of course, segments created this way are completely *unsafe*. There is no way for the runtime to verify that the provided address indeed points to a valid memory location, or that the size of the memory region pointed to by `addr` is indeed 10 bytes. Similarly, there are no guarantees that the underlying memory region associated with `addr` will not be deallocated *prior* to the call to `MemorySegment::close`.

For these reasons, creating unsafe segments is a *restricted* operation in the Foreign Memory Access API. Restricted operations can only be performed if the running application has set a read-only runtime property — `foreign.restricted=permit`. Any attempt to call restricted operations without said runtime property will fail with a runtime exception.

We plan, in the future, to make access to restricted operations more integrated with the module system; that is, certain modules might *require* restricted native access; when an application which depends on said modules is executed, the user might need to provide *permissions* to said modules to perform restricted native operations, or the runtime will refuse to build the application's module graph.

Confinement

In addition to spatial and temporal bounds, segments also feature thread-confinement. That is, a segment is *owned* by the thread which created it, and no other thread can access the contents on the segment, or perform certain operations (such as `close`) on it. Thread confinement, while restrictive, is crucial to guarantee optimal memory access performance even in a multi-threaded environment.

The Foreign Memory Access API provides several ways to relax the thread confinement barriers. First, threads can cooperatively share segments by performing explicit *handoff* operations, where a thread releases its ownership on a given segment and transfers it onto another thread. Consider the following code:

```
MemorySegment segmentA = MemorySegment.allocateNative(10); // confined by thread A
...
var segmentB = segmentA.withOwnerThread(threadB); // confined by thread B
```

This pattern of access is also known as *serial confinement* and might be useful in producer/consumer use cases where only one thread at a time needs to access a segment. Note that, to make the handoff operation safe, the API *kills* the original segment (as if `close` was called, but without releasing the underlying memory) and returns a *new* segment with the correct owner. The implementation also makes sure that all writes by the first thread are flushed into memory by the time the second thread accesses the segment.

When serial confinement is not enough, clients can optionally remove thread ownership, that is, turn a confined segment into a *shared* one which can be accessed — and closed — concurrently, by multiple threads[2]. As before, sharing a segment kills the original segment and returns a new segment with no owner thread:

```
MemorySegment segmentA = MemorySegment.allocateNative(10); // confined by thread A
...
var sharedSegment = segmentA.withOwnerThread(null); // shared segment
```

A shared segments is especially useful when multiple threads need to operate on the segment's contents in *parallel* (e.g. using a framework such as Fork/Join) — by obtaining a `Spliterator` instance out of a memory segment. For instance to sum all the 32 bit values of a memory segment in parallel, we can use the following code:

```
SequenceLayout seq = MemoryLayout.ofSequence(1_000_000, MemoryLayouts.JAVA_INT);
SequenceLayout seq_bulk = seq.reshape(-1, 100);
VarHandle intHandle = seq.varHandle(int.class, sequenceElement());

int sum = StreamSupport.stream(MemorySegment.spliterator(segment.withOwnerThread(null), seq_bulk),
true)
            .mapToInt(slice -> {
               int res = 0;
               for (int i = 0; i < 100 ; i++) {
                  res += MemoryAccess.getIntAtIndex(slice, i);
               }
               return res;
            }).sum();
```

The `MemorySegment::spliterator` takes a segment, a *sequence* layout and returns a spliterator instance which splits the segment into chunks which corresponds to the elements in the provided sequence layout. Here, we want to sum elements in an array which contains a million of elements; now, doing a parallel sum where each computation processes *exactly* one element would be inefficient, so instead we use the layout AP to derive a *bulk* sequence layout. The bulk layout is a sequence layout which has the same size of the original layouts, but where the elements are arranged into groups of 100 elements — which should make it more amenable to parallel processing.

Once we have the spliterator, we can use it to construct a parallel stream and sum the contents of the segment in parallel. Since the segment operated upon by the spliterator is shared, the segment can be accessed from multiple threads concurrently; the spliterator AP ensures that the access occurs in a regular fashion: a slice is created from the original segment, and given to a thread to perform some computation — thus ensuring that no two threads can ever operate concurrently on the same memory region.

Shared segment can also be useful to perform serial confinement in cases where the thread handing off the segment does not know which other thread will continue the work on the segment, for instance:

```
// thread A
MemorySegment segment = MemorySegment.allocateNative(10); // confined by thread A
//do some work
segment = segment.withOwnerThread(null);

// thread B
segment.withOwnerThread(Thread.currentThread()); // now confined by thread B
// do some more work
```

That is, multiple threads can *race* to acquire a given shared segment — the API ensures that only one of them will succeed in acquiring ownership of the shared segment.

Implicit deallocation

While memory segment feature *deterministic deallocation* they can also be registered against a `Cleaner`, to make sure that the memory resources associated with a segment are released when the GC determines that the segment is no longer *reachable*:

```
MemorySegment segment = MemorySegment.allocateNative(100);
Cleaner cleaner = Cleaner.create();
segment.registerCleaner(cleaner);
// do some work
segment = null; // Cleaner might reclaim the segment memory now
```

Note that registering a segment with a cleaner doesn't prevent clients from calling `MemorySegment::close` explicitly; the API will guarantee that the segment's cleanup action will be called at most once — either explicitly, or implicitly (by a cleaner). Moreover, since an unreachable segment cannot (by definition) be accessed by any thread, the cleaner can always release any memory resources associated with an unreachable segment, regardless of whether it is a confined, or a shared segment.

---

1 In general, deriving a complete layout from a C `struct` declaration is no trivial matter, and it's one of those areas where tooling can help greatly. ↩

2 Shared segments rely on VM thread-local handshakes (JEP 312) to implement lock-free, safe, shared memory access; that is, when it comes to memory access, there should no difference in performance between a shared segment and a confined segment. On the other hand, `MemorySegment::close` might be slower on shared segments than on confined ones. ↩