Tricks with Direct Memory Access in Java

Java was initially designed as a safe, managed environment. Nevertheless, Java HotSpot VM contains a “backdoor” that provides a number of low-level operations to manipulate memory and threads directly. This backdoor – sun.misc.Unsafe – is widely used by JDK itself in the packages like java.nio or java.util.concurrent. It is hard to imagine a Java developer who uses this backdoor in any regular development because this API is extremely dangerous, non portable, and volatile. Nevertheless, Unsafe provides an easy way to look into HotSpot JVM internals and do some tricks. Sometimes it is simply funny, sometimes it can be used to study VM internals without C++ code debugging, sometimes it can be leveraged for profiling and development tools.

Obtaining Unsafe

The sun.misc.Unsafe class is so unsafe that JDK developers added special checks to restrict access to it. Its constructor is private and caller of the factory method getUnsafe() should be loaded by Bootloader (i.e. caller should also be a part of JDK):

```
1   public final class Unsafe {
2      ...
3      private Unsafe() {}
4      private static final Unsafe theUnsafe = new Unsafe();
5      ...
6      public static Unsafe getUnsafe() {
7         Class cc = sun.reflect.Reflection.getCallerClass(2);
8         if (cc.getClassLoader() != null)
9             throw new SecurityException("Unsafe");
10        return theUnsafe;
11     }
12     ...
13  }
```

Fortunately there is theUnsafe field that can be used to retrieve Unsafe instance. We can easily write a helper method to do this via reflection:

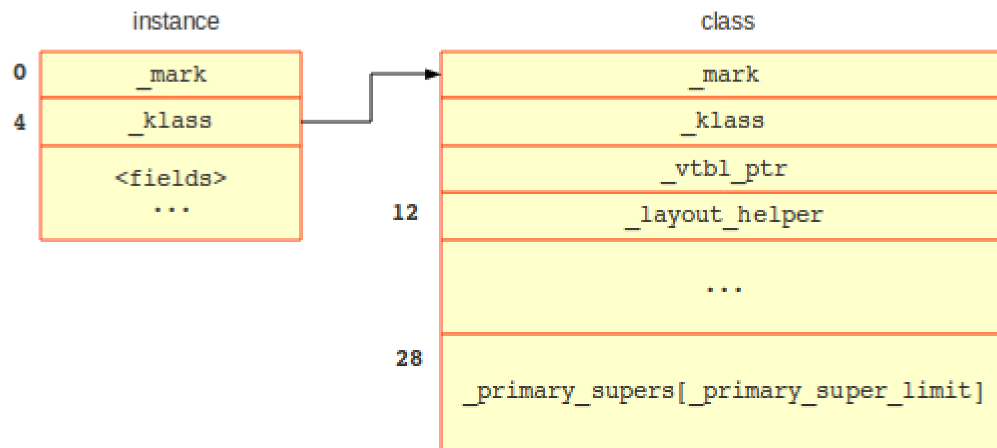
```
1   public static Unsafe getUnsafe() {
2       try {
3               Field f = Unsafe.class.getDeclaredField("theUnsafe");
4               f.setAccessible(true);
5               return (Unsafe)f.get(null);
6       } catch (Exception e) { /* ... */ }
7   }
```

In the next sections we will study several tricks that become possible due to the following methods of Unsafe:

- **long getAddress(long address)** and **void putAddress(long address, long x)** that allows to read and write dwords directly from memory.
- **int getInt(Object o, long offset)** , **void putInt(Object o, long offset, int x)**, and other similar methods that allows to read and write data directly from C structure that represents Java object.
- **long allocateMemory(long bytes)** which can be considered as a wrapper for C's malloc().

sizeof() Function

The first trick we will do is C-like sizeof() function, i.e. function that returns shallow object size in bytes. Inspecting JVM sources of JDK6 and JDK7, in particular src/share/vm/oops/oop.hpp and src/share/vm/oops/klass.hpp, and reading comments in the code, we can notice that size of class instance is stored in _layout_helper which is the fourth field in C structure that represents Java class. Similarly, /src/share/vm/oops/oop.hpp shows that each instance (i.e. object) stores pointer to a class structure in its second field. For 32-bit JVM this means that we can first take class structure address as 4-8 bytes in the object structure and next shift by 3×4=12 bytes inside class structure to capture_layout_helper field which is instance size in bytes. These structures are shown in the picture below:



As so, we can implement sizeof() as follows:

```
1
2
3
4
5
6
7
8
9
public static long sizeOf(Object object) {
  Unsafe unsafe = getUnsafe();
  return  unsafe.getAddress(  normalize(  unsafe.getInt(object,
4L) ) + 12L );
}

public static long normalize(int value) {
  if(value >= 0) return value;
  return (~0L >>> 32) & value;
}
```

We need to use normalize() function because addresses between 2^31 and 2^32 will be automatically converted to negative integers, i.e. stored in complement form. Let's test it on 32-bit JVM (JDK 6 or 7):

```
1
2
3
4
5
// sizeOf(new MyStructure()) gives the following results:

class MyStructure { } // 8: 4 (start marker) + 4 (pointer to class)
class MyStructure { int x; } // 16: 4 (start marker) + 4 (pointer to class) + 4 (int) + 4
stuff bytes to align structure to 64-bit blocks
class MyStructure { int x; int y; } // 16: 4 (start marker) + 4 (pointer to class) + 2*4
```

This function will not work for array objects, because _layout_helper field has another meaning in that case. Although it is still possible to generalize sizeOf() to support arrays.

Direct Memory Management

Unsafe allows to allocate and deallocate memory explicitly via allocateMemory and freeMemory methods. Allocated memory is not under GC control and not limited by maximum JVM heap size. In general, such functionality is safely available via NIO's off-heap bufferes. But the interesting thing is that it is possible to map standard Java reference to off-heap memory:

```
     MyStructure structure = new MyStructure(); // create a test object
     structure.x = 777;
1
2    long size = sizeOf(structure);
3    long offheapPointer = getUnsafe().allocateMemory(size);
4    getUnsafe().copyMemory(
5                structure,      // source object
6                0,              // source offset is zero - copy an entire object
7                null,           // destination is specified by absolute address, so destination
8    object is null
9                offheapPointer, // destination address
10               size
11   ); // test object was copied to off-heap
12
13   Pointer p = new Pointer(); // Pointer is just a handler that stores address of some
14   object
15   long                                    pointerOffset                              =
16   getUnsafe().objectFieldOffset(Pointer.class.getDeclaredField("pointer"));
17   getUnsafe().putLong(p, pointerOffset, offheapPointer); // set pointer to off-heap
18   copy of the test object
19
20   structure.x = 222; // rewrite x value in the original object
21   System.out.println(  ((MyStructure)p.pointer).x  ); // prints 777
22
23   ....
24
25   class Pointer {
        Object pointer;
     }
```

So, it is virtually possible to manually allocate and deallocate real objects, not only byte buffers. Of course, it's a big question what may happen with GC after such cheats.

Inheritance from Final Class and void*

Imagine the situation when one has a method that takes a string as an argument, but it is necessary to pass some extra payload. There are at least two standard ways to do it in Java: put payload to thread local or use static field. With Unsafe another two possibilities appears: pass payload address as a string and inherit payload class from String class. The first approach is pretty close to what we see in the previous section – one just need obtain payload address using Pointer and create a new Pointer to payload inside the called method. In other words, any argument that can carrier an address can be used as analog of void* in C. In order to explore the second approach we start with the following code which is compilable, but obviously produces ClassCastException in run time:

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
```

```
Carrier carrier = new Carrier();
carrier.secret = 777;

String message = (String)(Object)carrier; //
ClassCastException
handler( message );

...

void handler(String message) {
   System.out.println( ((Carrier)(Object)message).secret );
}

...

class Carrier {
   int secret;
}
```

To make it work, one need to modify Carrier class to simulate inheritance from String. A list of superclasses is stored in Carrier class structure starting from

position 28, as it shown in the figure. Pointer to object goes first and pointer to Carrier itself goes after it (at position 32) since Carrier is inherited from Object directly. In principle, it is enough to add the following code before the line that casts Carrier to String:

```
1
2
3
long carrierClassAddress = normalize( unsafe.getInt(carrier, 4L) );
long stringClassAddress = normalize( unsafe.getInt("", 4L) );
unsafe.putAddress(carrierClassAddress + 32, stringClassAddress); // insert pointer to Str
list of Carrier's superclasses
```

Now cast works fine. Nevertheless, this transformation is not correct and violates VM contracts. More careful approach should include more steps:

1. Position 32 in Carrier class actually contains a pointer to Carrier class itself, so this pointer should be shifted to position 36, not simply overwritten by the pointer to the String class.
2. Since Carrier is now inherited from String, final markers in String class should be removed.

Conclusion

sun.misc.Unsafe provides almost unlimited capabilities for exploring and modification of VM's runtime data structures. Despite the fact that these capabilities are almost inapplicable in Java development itself, Unsafe is a great tool for anyone who want to study HotSpot VM without C++ code debugging or need to create ad hoc profiling instruments.

Source: http://highlyscalable.wordpress.com/2012/02/02/direct-memory-access-in-java/