Improved JNI Memory Management Using Allocations from the Java Heap

Jeffrey Sorensen and Daniel Bikel

IBM T. J. Watson Research Center
Yorktown Heights, New York
{sorenj,dbikel}@us.ibm.com

1. Introduction

JAVA, as a development platform provides a number of programmer friendly capabilities, including **Platform neutrality**, **Garbage collection**, **Unicode support**, **Regular expressions**, and **Multi-threading support**. The Java Native Interface (JNI) to allow Java to interact with libraries written in other languages. However, JNI introduces platform specific dependencies that mitigate some of Java's benefits. In particular: Java's garbage collection system and heap management has no awareness or control over the use of dynamic memory in the native code. A JNI library can easily cause the Java program to exceed its maximum specified heap.

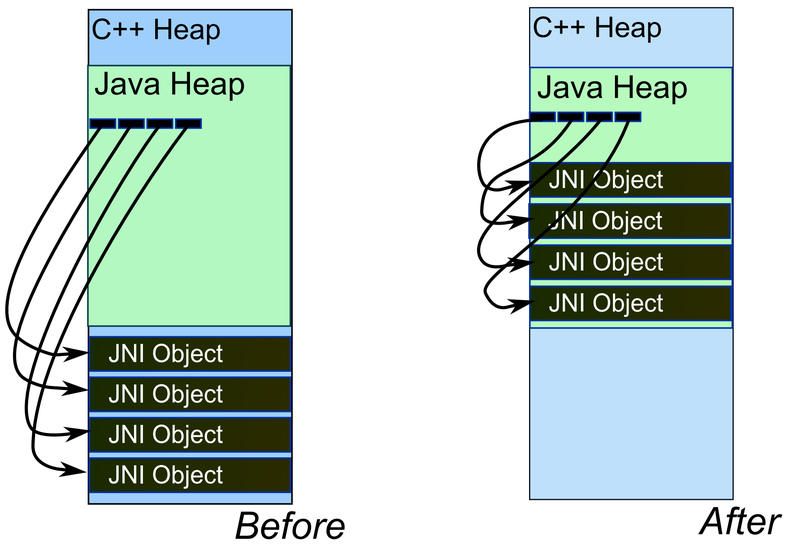
```
#include "jni.h"
extern "C" JNIEXPORT jlong JNICALL
Java_Wasteful__1_1c(JNIEnv*, jobject) {
  return (jlong) new char [1024];
}
extern "C" JNIEXPORT void JNICALL
Java_Wasteful__1_1d(JNIEnv*, jobject, jlong __cptr) {
  delete [] (char *) __cptr;
}
```

```
public class Wasteful {
  protected long cPtr = 0;
  private native long __c();
  private native void __d(long);
  public Wasteful() { cPtr = __c(); }
  protected void finalize() { __d(cPtr); }
  static public void main(String args[]) {
    while (1) { new Wasteful(); }
  }
};
```

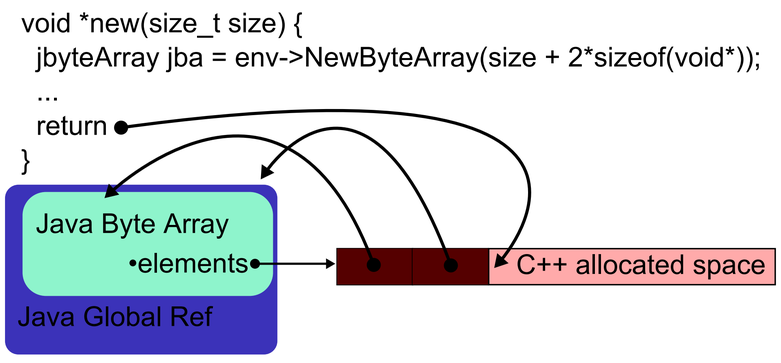
BECAUSE Java only accounts for the storage required for the *handles* to the C++ objects, the memory management system sees little reason to *finalize* objects that are *unreachable*. For Java to effectively manage persistent C++ objects, Java must know (at a minimum) when their aggregate resource consumption necessitates a garbage collection.

2. Java Heap Allocations

OUR solution is to use the language features of C++ to override the default heap allocation methods and instead depend upon the Java virtual machine to allocate storage for C++ objects. The principle advantage of this approach is that Java *will now become aware of each allocation* and will trigger more aggressive garbage collection when these allocations are becoming tight. While Java cannot directly force C++ objects to free up their storage, any unreachable Java objects that maintain handles to C++ objects will be finalized, and in turn, will free up their associated additional heap space.



REQUESTING Java heap storage for C++ objects is most naturally handled via the Java `jbytearray` object, which has methods to access the underlying byte buffer. Because JNI by default creates objects with only *local references*, they are eligible for garbage collection immediately upon return from the JNI calls. To prevent Java from deleting the byte arrays we create *Global References* until the C++ code has de-allocated the block by calling `delete`. We allocate more space than was requested to incorporate a *stash area* to contain the handles (pointers) to the `jbytearray` and the Global Reference.



3. Edge conditions

SIMILARLY supporting C libraries (that use `malloc`/`free`) would pose additional challenges, and would surrender platform independence. This is because Java itself uses these functions, and C, unlike C++, was not designed for overriding of heap allocators. However, using *library interceptors* and other tools this idea could be explored.

BECAUSE at least two Java objects are created with each allocation, it is important that `new` and `delete` not be called for very small objects. Fortunately, pool management is already part of most C++ standard libraries.

THE JVM's use of `dlopen` means that Java heap resources cannot effectively be used to address several C++ allocations including **the executable code segment**, **the initialized data segment**, **the uninitialized data segment**, and **global (extern and static) objects**. Global objects may invoke the `new` operator before the JVM calls the `JNI_OnLoad` method that provides the JNI library with a handle to the Java environment. During this period, our customized `new` method handles allocations from the ordinary C heap using `malloc`, and marks these blocks using a null pointer so they are freed appropriately.

4. SWIG

SWIG (Simplified Wrapper and Interface Generator) is a popular tool that allows one to publish C and C++ libraries through JNI. Due to its ability to inserting custom code, our implementation of `new` and `delete` is entirely compatible with the SWIG environment and can be easily added to a SWIG interface file, automating the use of JNI allocations for all SWIG managed objects. *We propose that the inclusion of this code be a standard (or at least optional) feature of the SWIG tool*, as the current implementations of SWIG generates code where heap management is likely to be problematical.

5. Conclusion

THE difference between the "roll-your-own" style of memory management in C++ and the more abstract and formal heap management within the Java Virtual Machine's garbage collection subsystems has generated endless debate about their relative merits. With JNI programming, developers truly have the worst of both worlds. Our own experience suggests that it is far too easy to create C++ objects that persist and consume large amounts of space. The use of STL smart pointers, such as provided by the Boost `shared_ptr` library, only exacerbates the problems caused by the philosophical disjuncture, as these reference-counting solutions do not handle the general case of arbitrary object graphs.

HOWEVER, with the incorporation of approximately 70 lines of code, a form of détente can be achieved. While C++ may still consume and hold resources for arbitrary amounts of time, by restricting allocation to the Java heap we are reasserting Java's own memory controls. In addition, the consumption of resources will trigger more frequent garbage collection and, when Java objects are proxies for C++ objects, those objects will be freed through finalization.

6. Complete Listing

```
#include <stdexcept>
#include "jni.h"

struct Jalloc {
  jbyteArray jba;
  jobject ref;
};

static JavaVM *cached_jvm = 0;

JNIEXPORT jint JNICALL
JNI_OnLoad(JavaVM *jvm, void *reserved)
{
  cached_jvm = jvm;
  return JNI_VERSION_1_2;
}

static JNIEnv *
JNU_GetEnv()
{
  JNIEnv *env;
  jint rc = cached_jvm->GetEnv((void **)&env, JNI_VERSION_1_2);
  if (rc == JNI_EDETACHED)
    throw std::runtime_error("current thread not attached");
  if (rc == JNI_EVERSION)
    throw std::runtime_error("jni version not supported");
  return env;
}

void *
operator new(size_t t)
{
  if (cached_jvm != 0) {
    JNIEnv *env = JNU_GetEnv();
    jbyteArray jba = env->NewByteArray((int) t + sizeof(Jalloc));
    if (env->ExceptionOccurred()) throw bad_alloc();
    void *jbuffer = static_cast<void *>(env->GetByteArrayElements(jba, 0));
    if (env->ExceptionOccurred()) throw bad_alloc();
    Jalloc *pJalloc = static_cast<Jalloc *>(jbuffer);
    pJalloc->jba = jba;
    pJalloc->ref = env->NewGlobalRef(jba);
    if (env->ExceptionOccurred()) throw bad_alloc();
    return static_cast<void *>(static_cast<char *>(jbuffer) + sizeof(Jalloc));
  }
  else {
    Jalloc *pJalloc = static_cast<Jalloc *>(malloc((int) t + sizeof(Jalloc)));
    if (!pJalloc) throw bad_alloc();
    pJalloc->ref = 0;
    return static_cast<void *>(
        static_cast<char *>(static_cast<void *>(pJalloc)) + sizeof(Jalloc));
  }
}

void operator delete(void *v) {
  if (v != 0) {
    void *buffer = static_cast<void *>(
        static_cast<char *>(v) - sizeof(Jalloc));
    Jalloc *pJalloc = static_cast<Jalloc *>(buffer);
    if (pJalloc->ref) {
      JNIEnv *env = JNU_GetEnv();
      env->DeleteGlobalRef(pJalloc->ref);
      env->ReleaseByteArrayElements(pJalloc->jba,
                                    static_cast<jbyte *>(buffer), 0);
    }
    else {
      free(buffer);
    }
  }
}
```