developerWorks®

Thanks for the memory, Linux

Understanding how the JVM uses native memory on Windows and Linux

Andrew Hall

April 21, 2009

Running out of Java™ heap isn't the only cause of a `java.lang.OutOfMemoryError`. If *native memory* runs out, `OutOfMemoryError`s that your normal debugging techniques won't be able to solve can occur. This article explains what native memory is, how the Java runtime uses it, what running out of it looks like, and how to debug a native `OutOfMemoryError` on Windows® and Linux®. A companion article covers the same topics for AIX® systems.

The Java heap, where every Java object is allocated, is the area of memory you're most intimately connected with when writing Java applications. The JVM was designed to insulate us from the host machine's peculiarities, so it's natural to think about the heap when you think about memory. You've no doubt encountered a Java heap `OutOfMemoryError` — caused by an object leak or by not making the heap big enough to store all your data — and have probably learned a few tricks to debug these scenarios. But as your Java applications handle more data and more concurrent load, you may start to experience `OutOfMemoryError`s that can't be fixed using your normal bag of tricks — scenarios in which the errors are thrown even though the Java heap isn't full. When this happens, you need to understand what is going on inside your Java Runtime Environment (JRE).

Java applications run in the virtualized environment of the Java runtime, but the runtime itself is a native program written in a language (such as C) that consumes native resources, including *native memory*. Native memory is the memory available to the runtime process, as distinguished from the Java heap memory that a Java application uses. Every virtualized resource — including the Java heap and Java threads — must be stored in native memory, along with the data used by the virtual machine as it runs. This means that the limitations on native memory imposed by the host machine's hardware and operating system (OS) affect what you can do with your Java application.

This article is one of two covering the same topic on different platforms. In both, you'll learn what native memory is, how the Java runtime uses it, what running out of it looks like, and how to debug a native `OutOfMemoryError`. This article covers Windows and Linux and does not focus on any particular runtime implementation. The companion article covers AIX and focuses on the IBM® Developer Kit for Java. (The information in that article about the IBM implementation is also true for platforms other than AIX, so if you use the IBM Developer Kit for Java on Linux or the IBM 32-bit Runtime Environment for Windows, you might find that article useful too.)

A recap of native memory

I'll start by explaining the limitations on native memory imposed by the OS and the underlying hardware. If you're familiar with managing dynamic memory in a language such as C, then you may want to skip to the next section.

Hardware limitations

Many of the restrictions that a native process experiences are imposed by the hardware, not the OS. Every computer has a processor and some random-access memory (RAM), also known as physical memory. A processor interprets a stream of data as instructions to execute; it has one or more processing units that perform integer and floating-point arithmetic as well as more advanced computations. A processor has a number of *registers* — very fast memory elements that are used as working storage for the calculations that are performed; the register size determines the largest number that a single calculation can use.

The processor is connected to physical memory by the memory bus. The size of the physical address (the address used by the processor to index physical RAM) limits the amount of memory that can be addressed. For example, a 16-bit physical address can address from 0x0000 to 0xFFFF, which gives 2^16 = 65536 unique memory locations. If each address references a byte of storage, a 16-bit physical address would allow a processor to address 64KB of memory.

Recommended resources

- "Thanks for the memory (for AIX)"
- "IBM Monitoring and Diagnostic Tools for Java - Memory Analyzer"
- "Guided debugging for Java"

Processors are described as being a certain number of bits. This normally refers to the size of the registers, although there are exceptions — such as 390 31-bit — where it refers to the physical address size. For desktop and server platforms, this number is 31, 32, or 64; for embedded devices and microprocessors, it can be as low as 4. The physical address size can be the same as the register width but could be larger or smaller. Most 64-bit processors can run 32-bit programs when running a suitable OS.

Table 1 lists some popular Linux and Windows architectures with their register and physical address sizes:

Table 1. Register and physical address size of some popular processor architectures

Architecture	Register width (bits)	Physical address size (bits)
(Modern) Intel® x86	32	32 36 with Physical Address Extension (Pentium Pro and above)
x86 64	64	Currently 48-bit (scope to increase later)
PPC64	64	50-bit at POWER 5
390 31-bit	32	31
390 64-bit	64	64

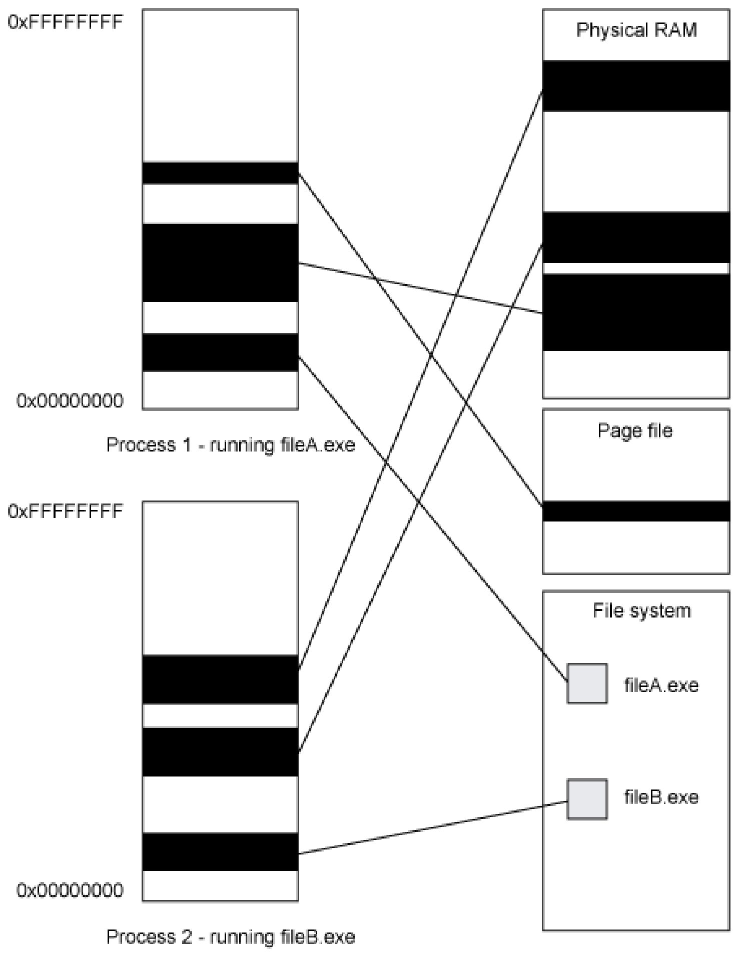
Operating systems and virtual memory

If you were writing applications to run directly on the processor without an OS, you could use all memory that the processor can address (assuming enough physical RAM is connected). But to enjoy features such as multitasking and hardware abstraction, nearly everybody uses an OS of some kind to run their programs.

In multitasking OSs such as Windows and Linux, more than one program uses system resources, including memory. Each program needs to be allocated regions of the physical memory to work in. It's possible to design an OS such that every program works directly with physical memory and is trusted to use only the memory it has been given. Some embedded OSs work like this, but it's not practical in an environment consisting of many programs that are not tested together because any program could corrupt the memory of other programs or the OS itself.

Virtual memory allows multiple processes to share physical memory without being able to corrupt one another's data. In an OS with virtual memory (such as Windows, Linux, and many others), each program has its own virtual address space — a logical region of addresses whose size is dictated by the address size on that system (so 31, 32, or 64 bits for desktop and server platforms). Regions in a process's virtual address space can be mapped to physical memory, to a file, or to any other addressable storage. The OS can move data held in physical memory to and from a swap area (the *page file* on Windows or a *swap partition* on Linux) when it isn't being used, to make the best use of physical memory. When a program tries to access memory using a virtual address, the OS in combination with on-chip hardware maps that virtual address to the physical location. That location could be physical RAM, a file, or the page file/swap partition. If a region of memory has been moved to swap space, then it is loaded back into physical memory before being used. Figure 1 shows how virtual memory works by mapping regions of process address space to shared resources:

Figure 1. Virtual memory mapping process address spaces to physical resources



Each instance of a program runs as a *process*. A process on Linux and Windows is a collection of information about OS-controlled resources (such as file and socket information), typically one virtual address space (more than one on some architectures), and at least one thread of execution.

The virtual address space size can be smaller than the processor's physical address size. Intel x86 32-bit originally had a 32-bit physical address that allowed the processor to address 4GB of storage. Later, a feature called Physical Address Extension (PAE) was added that expanded the physical address size to 36-bit — allowing up to 64GB of RAM to be installed and addressed. PAE allowed OSs to map 32-bit 4GB virtual address spaces onto a large physical address range, but it did not allow each process to have a 64GB virtual address space. This means that if you put more than 4GB of memory into a 32-bit Intel server, you can't map all of it directly into a single process.

The Address Windowing Extensions feature allows a Windows process to map a portion of its 32-bit address space as a sliding window into a larger area of memory. Linux uses similar technologies based on mapping regions into the virtual address space. This means that although you can't directly reference more than 4GB of memory, you can work with larger regions of memory.

Kernel space and user space

Although each process has its own address space, a program typically can't use all of it. The address space is divided into *user space* and *kernel space*. The kernel is the main OS program and contains the logic for interfacing to the computer hardware, scheduling programs, and providing services such as networking and virtual memory.

As part of the computer boot sequence, the OS kernel runs and initialises the hardware. Once the kernel has configured the hardware and its own internal state, the first user-space process starts. If a user program needs a service from the OS, it can perform an operation — called a *system call* — that jumps into the kernel program, which then performs the request. System calls are generally required for operations such as reading and writing files, networking, and starting new processes.

The kernel requires access to its own memory and the memory of the calling process when executing a system call. Because the processor that is executing the current thread is configured to map virtual addresses using the address-space mapping for the current process, most OSs map a portion of each process address space to a common kernel memory region. The portion of the address space mapped for use by the kernel is called kernel space; the remainder, which can be used by the user application, is called user space.

The balance between kernel and user space varies by OS and even among instances of the same OS running on different hardware architectures. The balance is often configurable and can be adjusted to give more space to user applications or the kernel. Squeezing the kernel area can cause problems such as restricting the number of users that can be logged on simultaneously or the number of processes that can run; smaller user space means that the application programmer has less room to work in.

By default, 32-bit Windows has a 2GB user space and a 2GB kernel space. The balance can be shifted to a 3GB user space and a 1GB kernel space on some versions of Windows by adding the `/3GB` switch to the boot configuration and relinking applications with the `/LARGEADDRESSAWARE` switch. On 32-bit Linux, the default is a 3GB user space and 1GB kernel space. Some Linux distributions provide a *hugemem* kernel that supports a 4GB user space. To achieve this, the kernel is given an address space of its own that is used when a system call is made. The gains in user space are offset by slower system calls because the OS must copy data between address spaces and reset the process address-space mappings each time a system call is made. Figure 2 shows the address-space layout for 32-bit Windows:

Figure 2. Address-space layout for 32-bit Windows

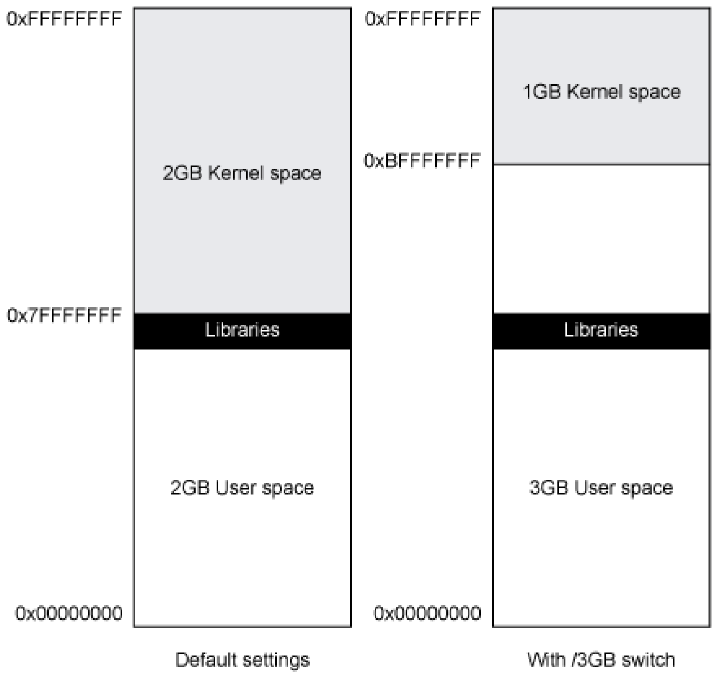
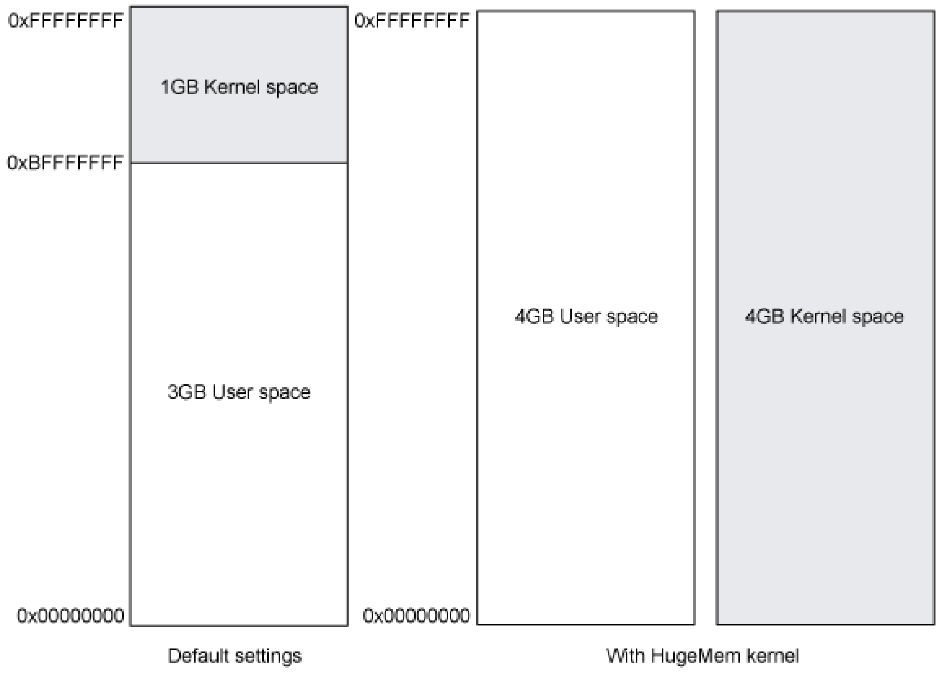


Figure 3 shows the address-space arrangements for 32-bit Linux:

Figure 3. Address-space layout for 32-bit Linux



A separate kernel address space is also used on Linux 390 31-bit, where the smaller 2GB address space makes dividing a single address space undesirable; however, the 390 architecture can work with multiple address spaces simultaneously without compromising performance.

The process address space must contain everything a program requires — including the program itself and the shared libraries (DLLs on Windows, .so files on Linux) that it uses. Not only can shared libraries take up space that a program can't use to store data in, they can also fragment the address space and reduce the amount of memory that can be allocated as a continuous chunk. This is noticeable in programs running on Windows x86 with a 3GB user space. DLLs are built with a preferred loading address: when a DLL is loaded, it's mapped into the address space at a particular location unless that location is already occupied, in which case it's rebased and loaded elsewhere. With the 2GB user space available when Windows NT was originally designed, it made sense for the system libraries to be built to load near the 2GB boundary — thereby leaving most of the user region free for the application to use. When the user region is extended to 3GB, the system shared libraries still load near 2GB — now in the middle of the user space. Although there's a total user space of 3GB, it's impossible to allocate a 3GB block of memory because the shared libraries get in the way.

Using the `/3GB` switch on Windows reduces the kernel space to half what it was originally designed to be. In some scenarios it's possible to exhaust the 1GB kernel space and experience slow I/O or problems creating new user sessions. Although the `/3GB` switch can be extremely valuable for some applications, any environment using it should be thoroughly load tested before being deployed.

A native-memory leak or excessive native memory use will cause different problems depending on whether you exhaust the address space or run out of physical memory. Exhausting the address space normally happens only with 32-bit processes — because the maximum 4GB is easy to allocate. A 64-bit process has a user space of hundreds or thousands of gigabytes, which is hard to fill up even if you try. If you do exhaust the address space of a Java process, then the Java runtime can start to display strange symptoms that I'll describe later in the article. When running on a system with more process address space than physical memory, a memory leak or excessive use of native memory forces the OS to swap out the backing storage for some of the native process's virtual address space. Accessing a memory address that has been swapped is much slower than reading a resident (in physical memory) address because the OS must pull the data from the hard drive. It's possible to allocate enough memory to exhaust all physical memory and all swap memory (paging space); on Linux, this triggers the kernel out-of-memory (OOM) killer, which forcibly kills the most memory-hungry process. On Windows, allocations start failing in the same way as they would if the address space were full.

If you simultaneously try to use more virtual memory than there is physical memory, it will be obvious there's a problem long before the process is killed for being a memory hog. The system will thrash — that is, spend most of its time copying memory back and forth from swap space. When this happens, the performance of the computer and the individual applications will become so poor that the user can't fail to notice there's a problem. When a JVM's Java heap is swapped out, the garbage collector's performance becomes extremely poor, to the extent that the application can appear to hang. If multiple Java runtimes are in use on a single machine at the same time, the physical memory must be sufficient to fit all of the Java heaps.

How the Java runtime uses native memory

The Java runtime is an OS process that is subject to the hardware and OS constraints I outlined in the preceding section. Runtime environments provide capabilities that are driven by some unknown user code; that makes it impossible to predict which resources the runtime environment will require in every situation. Every action a Java application takes inside the managed Java environment can potentially affect the resource requirements of the runtime that provides that environment. This section describes how and why Java applications consume native memory.

The Java heap and garbage collection

The Java heap is the area of memory where objects are allocated. Most Java SE implementations have one logical heap, although some specialist Java runtimes such as those implementing the Real Time Specification for Java (RTSJ) have multiple heaps. A single physical heap can be split up into logical sections depending on the garbage collection (GC) algorithm used to manage the heap memory. These sections are typically implemented as contiguous slabs of native memory that are under the control of the Java memory manager (which includes the garbage collector).

The heap's size is controlled from the Java command line using the `-Xmx` and `-Xms` options (`mx` is the maximum size of the heap, `ms` is the initial size). Although the logical heap (the area of memory that is actively used) can grow and shrink according to the number of objects on the heap and the amount of time spent in GC, the amount of native memory used remains constant and is dictated by the `-Xmx` value: the maximum heap size. Most GC algorithms rely on the heap being allocated as a contiguous slab of memory, so it's impossible to allocate more native memory when the heap needs to expand. All heap memory must be reserved up front.

Reserving native memory is not the same as allocating it. When native memory is reserved, it is not backed with physical memory or other storage. Although reserving chunks of the address space will not exhaust physical resources, it does prevent that memory from being used for other purposes. A leak caused by reserving memory that is never used is just as serious as leaking allocated memory.

Some garbage collectors minimise the use of physical memory by decommitting (releasing the backing storage for) parts of the heap as the used area of heap shrinks.

More native memory is required to maintain the state of the memory-management system maintaining the Java heap. Data structures must be allocated to track free storage and record progress when collecting garbage. The exact size and nature of these data structures varies with implementation, but many are proportional to the size of the heap.

The Just-in-time (JIT) compiler

The JIT compiler compiles Java bytecode to optimised native executable code at run time. This vastly improves the run-time speed of Java runtimes and allows Java applications to run at speeds comparable to native code.

Bytecode compilation uses native memory (in the same way that a static compiler such as `gcc` requires memory to run), but both the input (the bytecode) and the output (the executable code) from the JIT must also be stored in native memory. Java applications that contain many JIT-compiled methods use more native memory than smaller applications.

Classes and classloaders

Java applications are composed of classes that define object structure and method logic. They also use classes from the Java runtime class libraries (such as `java.lang.String`) and may use third-party libraries. These classes need to be stored in memory for as long as they are being used.

How classes are stored varies by implementation. The Sun JDK uses the permanent generation (PermGen) heap area. The IBM implementation from Java 5 onward allocates slabs of native memory for each classloader and stores the class data in there. Modern Java runtimes have technologies such as class sharing that may require mapping areas of shared memory into the address space. To understand how these allocation mechanisms affect your Java runtime's native footprint, you need to read the technical documentation for that implementation. However, some universal truths affect all implementations.

At the most basic level, using more classes uses more memory. (This may mean your native memory usage just increases or that you must explicitly resize an area — such as the PermGen or the shared-class cache — to allow all the classes to fit in.) Remember that it's not only your application that needs to fit; frameworks, application servers, third-party libraries, and Java runtimes contain classes that are loaded on demand and occupy space.

Java runtimes can unload classes to reclaim space, but only under strict conditions. It's impossible to unload a single class; classloaders are unloaded instead, taking all the classes they loaded with them. A classloader can be unloaded only if:

- The Java heap contains no references to the `java.lang.ClassLoader` object that represents that classloader.
- The Java heap contains no references to any of the `java.lang.Class` objects that represent classes loaded by that classloader.
- No objects of any class loaded by that classloader are alive (referenced) on the Java heap.

It's worth noting that the three default classloaders that the Java runtime creates for all Java applications — *bootstrap*, *extension*, and *application* — can never meet these criteria; therefore, any system classes (such as `java.lang.String`) or any application classes loaded through the application classloader can't be released at run time.

Even when a classloader is eligible for collection, the runtime collects classloaders only as part of a GC cycle. Some implementations unload classloaders only on some GC cycles.

It's also possible for classes to be generated at run time, without you realising it. Many JEE applications use JavaServer Pages (JSP) technology to produce Web pages. Using JSP generates a class for each .jsp page executed that will last the lifetime of the classloader that loaded them — typically the lifetime of the Web application.

Another common way to generate classes is by using Java reflection. The way reflection works varies among Java implementations, but Sun and IBM implementations both use the method I'll describe now.

When using the `java.lang.reflect` API, the Java runtime must connect the methods of a reflecting object (such as `java.lang.reflect.Field`) to the object or class being reflected on. This can be done by using a Java Native Interface (JNI) accessor that requires very little setup but is slow to use, or by building a class dynamically at run time for each object type you want to reflect on. The latter method is slower to set up but faster to run, making it ideal for applications that reflect on a particular class often.

The Java runtime uses the JNI method the first few times a class is reflected on, but after being used a number of times, the accessor is inflated into a bytecode accessor, which involves building a class and loading it through a new classloader. Doing lots of reflection can cause many accessor classes and classloaders to be created. Holding references to the reflecting objects causes these classes to stay alive and continue occupying space. Because creating the bytecode accessors is quite slow, the Java runtime can cache these accessors for later use. Some applications and frameworks also cache reflection objects, thereby increasing their native footprint.

JNI

JNI allows native code (applications written in native compiled languages such as C and C++) to call Java methods and vice versa. The Java runtime itself relies heavily on JNI code to implement class-library functions such as file and network I/O. A JNI application can increase a Java runtime's native footprint in three ways:

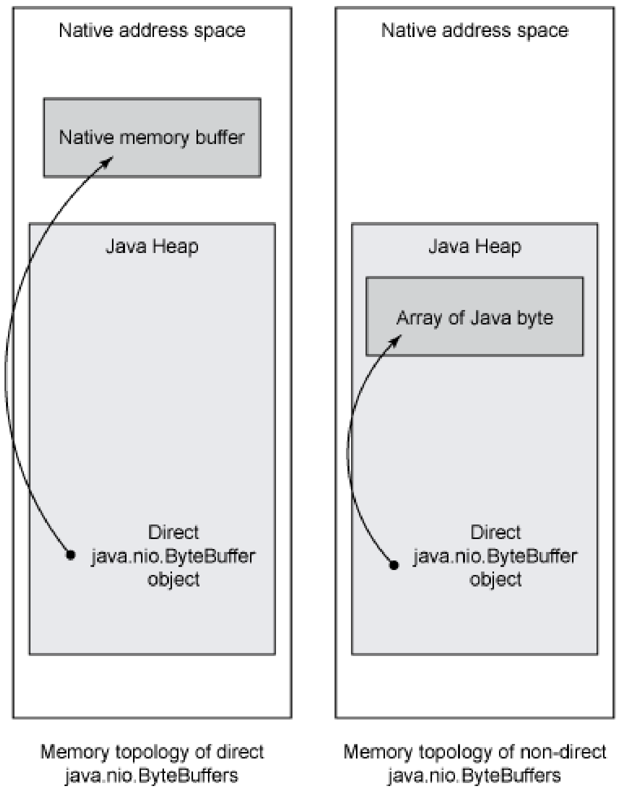
- The native code for a JNI application is compiled into a shared library or executable that's loaded into the process address space. Large native applications can occupy a significant chunk of the process address space simply by being loaded.
- The native code must share the address space with the Java runtime. Any native-memory allocations or memory mappings performed by the native code take memory away from the Java runtime.
- Certain JNI functions can use native memory as part of their normal operation. The `Get`*`Type`*`ArrayElements` and `Get`*`Type`*`ArrayRegion` functions can copy Java heap data into native memory buffers for the native code to work with. Whether a copy is made or not depends on the runtime implementation. (The IBM Developer Kit for Java 5.0 and higher makes a native copy.) Accessing large amounts of Java heap data in this manner can use a correspondingly large amount of native heap.

NIO

The new I/O (NIO) classes added in Java 1.4 introduced a new way of performing I/O based on channels and buffers. As well as I/O buffers backed by memory on the Java heap, NIO added support for *direct* `ByteBuffer`s (allocated using the `java.nio.ByteBuffer.allocateDirect()` method) that are backed by native memory rather than Java heap. Direct `ByteBuffer`s can be passed directly to native OS library functions for performing I/O — making them significantly faster in some scenarios because they can avoid copying data between Java heap and native heap.

It's easy to become confused about where direct `ByteBuffer` data is being stored. The application still uses an object on the Java heap to orchestrate I/O operations, but the buffer that holds the data is held in native memory — the Java heap object only contains a reference to the native heap buffer. A non-direct `ByteBuffer` holds its data in a `byte[]` array on the Java heap. Figure 4 shows the difference between direct and non-direct `ByteBuffer` objects:

Figure 4. Memory topology for direct and non-direct `java.nio.ByteBuffers`


Native address space
Native memory buffer
Java Heap
Direct java.nio.ByteBuffer object
Memory topology of direct java.nio.ByteBuffers
Native address space
Java Heap
Array of Java byte
Direct java.nio.ByteBuffer object
Memory topology of non-direct java.nio.ByteBuffers

Direct `ByteBuffer` objects clean up their native buffers automatically but can only do so as part of Java heap GC — so they do not automatically respond to pressure on the native heap. GC occurs only when the Java heap becomes so full it can't service a heap-allocation request or if the Java application explicitly requests it (not recommended because it causes performance problems).

The pathological case would be that the native heap becomes full and one or more direct `ByteBuffers` are eligible for GC (and could be freed to make some space on the native heap), but the Java heap is mostly empty so GC doesn't occur.

Threads

Every thread in an application requires memory to store its *stack* (the area of memory used to hold local variables and maintain state when calling functions). Every Java thread requires stack space to run. Depending on implementation, a Java thread can have separate native and Java stacks. In addition to stack space, each thread requires some native memory for thread-local storage and internal data structures.

The stack size varies by Java implementation and by architecture. Some implementations allow you to specify the stack size for Java threads. Values between 256KB and 756KB are typical.

Although the amount of memory used per thread is quite small, for an application with several hundred threads, the total memory use for thread stacks can be large. Running an application with many more threads than available processors to run them is usually inefficient and can result in poor performance as well as increased memory usage.

How can I tell if I'm running out of native memory?

A Java runtime copes quite differently with running out of Java heap compared to running out of native heap, although both conditions can present with similar symptoms. A Java application finds it extremely difficult to function when the Java heap is exhausted — because it's difficult for a Java application to do anything without allocating objects. The poor GC performance and `OutOfMemoryError`s that signify a full Java heap are produced as soon as the Java heap fills up.

In contrast, once a Java runtime has started up and the application is in steady state, it can continue to function with complete native-heap exhaustion. It doesn't necessarily show any odd behaviour, because actions that require a native-memory allocation are much rarer than actions that require Java-heap allocations. Although actions that require native memory vary by JVM implementation, some popular examples are: starting a thread, loading a class, and performing certain kinds of network and file I/O.

Native out-of-memory behaviour is also less consistent than Java heap out-of-memory behaviour, because there's no single point of control for native heap allocations. Whereas all Java heap allocations are under control of the Java memory-management system, any native code — whether it is inside the JVM, the Java class libraries, or application code — can perform a native-memory allocation and have it fail. The code that attempts the allocation can then handle it however its designer wants: it could throw an `OutOfMemoryError` through the JNI interface, print a message on the screen, silently fail and try again later, or do something else.

The lack of predictable behaviour means there's no one simple way to identify native-memory exhaustion. Instead, you need to use data from the OS and from the Java runtime to confirm the diagnosis.

Examples of running out of native memory

To help you see how native-memory exhaustion affects the Java implementation you're using, this article's sample code (see Download) contains some Java programs that trigger native-heap exhaustion in different ways. The examples use a native library written in C to consume all of the native address space and then try to perform some action that uses native memory. The examples are supplied already built, although instructions on compiling them are provided in the README.html file in the sample package's top-level directory.

The `com.ibm.jtc.demos.NativeMemoryGlutton` class provides the `gobbleMemory()` method, which calls `malloc` in a loop until nearly all native memory is exhausted. When it has completed its task, it prints the number of bytes allocated to standard error like this:

```
Allocated 1953546736 bytes of native memory before running out
```

The output for each demo has been captured for a Sun and an IBM Java runtime running on 32-bit Windows. The binaries provided have been tested on:

- Linux x86
- Linux PPC 32
- Linux 390 31
- Windows x86

The following version of the Sun Java runtime was used to capture the output:

```
java version "1.5.0_11"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_11-b03)
Java HotSpot(TM) Client VM (build 1.5.0_11-b03, mixed mode)
```

The version of the IBM Java runtime used was:

```
java version "1.5.0"
Java(TM) 2 Runtime Environment, Standard Edition (build pwi32devifx-20071025 (SR
6b))
IBM J9 VM (build 2.3, J2RE 1.5.0 IBM J9 2.3 Windows XP x86-32 j9vmwi3223-2007100
7 (JIT enabled)
J9VM - 20071004_14218_lHdSMR
JIT  - 20070820_1846ifx1_r8
GC   - 200708_10)
JCL  - 20071025
```

Trying to start a thread when out of native memory

The `com.ibm.jtc.demos.StartingAThreadUnderNativeStarvation` class tries to start a thread when the process address space is exhausted. This is a common way to discover that your Java process is out of memory because many applications start threads throughout their lifetime.

The output from the `StartingAThreadUnderNativeStarvation` demo when run on the IBM Java runtime is:

```
Allocated 1019394912 bytes of native memory before running out
JVMDUMP006I Processing Dump Event "systhrow", detail
"java/lang/OutOfMemoryError" - Please Wait.
JVMDUMP007I JVM Requesting Snap Dump using 'C:\Snap0001.20080323.182114.5172.trc'
JVMDUMP010I Snap Dump written to C:\Snap0001.20080323.182114.5172.trc
JVMDUMP007I JVM Requesting Heap Dump using 'C:\heapdump.20080323.182114.5172.phd'
JVMDUMP010I Heap Dump written to C:\heapdump.20080323.182114.5172.phd
JVMDUMP007I JVM Requesting Java Dump using 'C:\javacore.20080323.182114.5172.txt'
JVMDUMP010I Java Dump written to C:\javacore.20080323.182114.5172.txt
JVMDUMP013I Processed Dump Event "systhrow", detail "java/lang/OutOfMemoryError".
java.lang.OutOfMemoryError: ZIP006:OutOfMemoryError, ENOMEM error in ZipFile.open
   at java.util.zip.ZipFile.open(Native Method)
   at java.util.zip.ZipFile.<init>(ZipFile.java:238)
   at java.util.jar.JarFile.<init>(JarFile.java:169)
   at java.util.jar.JarFile.<init>(JarFile.java:107)
   at com.ibm.oti.vm.AbstractClassLoader.fillCache(AbstractClassLoader.java:69)
   at com.ibm.oti.vm.AbstractClassLoader.getResourceAsStream(AbstractClassLoader.java:113)
   at java.util.ResourceBundle$1.run(ResourceBundle.java:1101)
   at java.security.AccessController.doPrivileged(AccessController.java:197)
   at java.util.ResourceBundle.loadBundle(ResourceBundle.java:1097)
   at java.util.ResourceBundle.findBundle(ResourceBundle.java:942)
   at java.util.ResourceBundle.getBundleImpl(ResourceBundle.java:779)
   at java.util.ResourceBundle.getBundle(ResourceBundle.java:716)
   at com.ibm.oti.vm.MsgHelp.setLocale(MsgHelp.java:103)
   at com.ibm.oti.util.Msg$1.run(Msg.java:44)
   at java.security.AccessController.doPrivileged(AccessController.java:197)
   at com.ibm.oti.util.Msg.<clinit>(Msg.java:41)
   at java.lang.J9VMInternals.initializeImpl(Native Method)
   at java.lang.J9VMInternals.initialize(J9VMInternals.java:194)
   at java.lang.ThreadGroup.uncaughtException(ThreadGroup.java:764)
   at java.lang.ThreadGroup.uncaughtException(ThreadGroup.java:758)
   at java.lang.Thread.uncaughtException(Thread.java:1315)
K0319java.lang.OutOfMemoryError: Failed to fork OS thread
   at java.lang.Thread.startImpl(Native Method)
   at java.lang.Thread.start(Thread.java:979)
   at com.ibm.jtc.demos.StartingAThreadUnderNativeStarvation.main(
StartingAThreadUnderNativeStarvation.java:22)
```

Calling `java.lang.Thread.start()` tries to allocate memory for a new OS thread. This attempt fails and causes an `OutOfMemoryError` to be thrown. The `JVMDUMP` lines notify the user that the Java runtime has produced its standard `OutOfMemoryError` debugging data.

Trying to handle the first `OutOfMemoryError` caused a second — the `:OutOfMemoryError, ENOMEM error in ZipFile.open`. Multiple `OutOfMemoryError`s are common when the native process memory is exhausted. The `Failed to fork OS thread` message is probably the most commonly encountered sign of running out of native memory.

The examples supplied with this article trigger clusters of `OutOfMemoryError`s that are more severe than anything you are likely to see with your own applications. This is partly because virtually all of the native memory has been used up and, unlike in a real application, that memory isn't freed later. In a real application, as `OutOfMemoryError`s are thrown, threads shut down and the native-memory pressure is likely to be relieved for a bit, giving the runtime a chance to handle the error. The trivial nature of the test cases also means that whole sections of the class library (such as the security system) have not been initialised yet — and their initialisation is driven by the runtime trying to handle the out-of-memory condition. In a real application, you may see some of the errors shown here, but it's unlikely you will see them all together.

When the same test case is executed on the Sun Java runtime, the following console output is produced:

```
Allocated 1953546736 bytes of native memory before running out
Exception in thread "main" java.lang.OutOfMemoryError: unable to create new native thread
   at java.lang.Thread.start0(Native Method)
   at java.lang.Thread.start(Thread.java:574)
   at com.ibm.jtc.demos.StartingAThreadUnderNativeStarvation.main(
StartingAThreadUnderNativeStarvation.java:22)
```

Although the stack trace and the error message are slightly different, the behaviour is essentially the same: the native allocation fails and a `java.lang.OutOfMemoryError` is thrown. The only thing that distinguishes the `OutOfMemoryError`s thrown in this scenario from those thrown because of Java-heap exhaustion is the message.

Trying to allocate a direct `ByteBuffer` when out of native memory

The `com.ibm.jtc.demos.DirectByteBufferUnderNativeStarvation` class tries to allocate a direct (that is, natively backed) `java.nio.ByteBuffer` object when the address space is exhausted. When run on the IBM Java runtime, it produces the following output:

```
Allocated 1019481472 bytes of native memory before running out
JVMDUMP006I Processing Dump Event "uncaught", detail
"java/lang/OutOfMemoryError" - Please Wait.
JVMDUMP007I JVM Requesting Snap Dump using 'C:\Snap0001.20080324.100721.4232.trc'
JVMDUMP010I Snap Dump written to C:\Snap0001.20080324.100721.4232.trc
JVMDUMP007I JVM Requesting Heap Dump using 'C:\heapdump.20080324.100721.4232.phd'
JVMDUMP010I Heap Dump written to C:\heapdump.20080324.100721.4232.phd
JVMDUMP007I JVM Requesting Java Dump using 'C:\javacore.20080324.100721.4232.txt'
JVMDUMP010I Java Dump written to C:\javacore.20080324.100721.4232.txt
JVMDUMP013I Processed Dump Event "uncaught", detail "java/lang/OutOfMemoryError".
Exception in thread "main" java.lang.OutOfMemoryError:
Unable to allocate 1048576 bytes of direct memory after 5 retries
   at java.nio.DirectByteBuffer.<init>(DirectByteBuffer.java:167)
   at java.nio.ByteBuffer.allocateDirect(ByteBuffer.java:303)
   at com.ibm.jtc.demos.DirectByteBufferUnderNativeStarvation.main(
   DirectByteBufferUnderNativeStarvation.java:29)
Caused by: java.lang.OutOfMemoryError
   at sun.misc.Unsafe.allocateMemory(Native Method)
   at java.nio.DirectByteBuffer.<init>(DirectByteBuffer.java:154)
   ... 2 more
```

In this scenario, an `OutOfMemoryError` is thrown that triggers the default error documentation. The `OutOfMemoryError` reaches the top of the main thread's stack and is printed on `stderr`.

When run on the Sun Java runtime, this test case produces the following console output:

```
Allocated 1953546760 bytes of native memory before running out
Exception in thread "main" java.lang.OutOfMemoryError
   at sun.misc.Unsafe.allocateMemory(Native Method)
   at java.nio.DirectByteBuffer.<init>(DirectByteBuffer.java:99)
   at java.nio.ByteBuffer.allocateDirect(ByteBuffer.java:288)
   at com.ibm.jtc.demos.DirectByteBufferUnderNativeStarvation.main(
DirectByteBufferUnderNativeStarvation.java:29)
```

Debugging approaches and techniques

The first thing to do when faced with a `java.lang.OutOfMemoryError` or an error message about lack of memory is to determine which kind of memory has been exhausted. The easiest way to do this is to first check if the Java heap is full. If the Java heap did not cause the `OutOfMemory` condition, then you should analyse the native-heap usage.

Checking your vendor's documentation

The guidelines in this article are general debugging principles applied to understanding native out-of-memory scenarios. Your runtime vendor may supply its own debugging instructions that it expects you to follow when working with its support teams. If you are raising a problem ticket with your runtime vendor (including IBM), always check its debugging and diagnostic documentation to see what steps you are expected to take when submitting a problem report.

Checking the Java heap

The method for checking the heap utilisation varies among Java implementations. On IBM implementations of Java 5 and 6, the javacore file produced when the `OutOfMemoryError` is thrown will tell you. The javacore file is usually produced in the working directory of the Java process and has a name of the form javacore.*date*.*time*.*pid*.txt. If you open the file in a text editor, you can find a section that looks like this:

```
0SECTION       MEMINFO subcomponent dump routine
NULL           =================================
1STHEAPFREE    Bytes of Heap Space Free: 416760
1STHEAPALLOC   Bytes of Heap Space Allocated: 1344800
```

This section shows how much Java heap was free when the javacore was produced. Note that the values are in hexadecimal format. If the `OutOfMemoryError` was thrown because an allocation could not be satisfied, then the GC trace section will show this:

```
1STGCHTYPE     GC History
3STHSTTYPE     09:59:01:632262775 GMT j9mm.80 -   J9AllocateObject() returning NULL!
32 bytes requested for object of class 00147F80
```

`J9AllocateObject() returning NULL!` means that the Java heap allocation routine completed unsuccessfully and an `OutOfMemoryError` will be thrown.

It is also possible for an `OutOfMemoryError` to be thrown because the garbage collector is running too frequently (a sign that the heap is full and the Java application will be making little or no progress). In this case, you would expect the Heap Space Free value to be very small, and the GC trace will show one of these messages:

```
1STGCHTYPE     GC History
3STHSTTYPE     09:59:01:632262775 GMT j9mm.83 -     Forcing J9AllocateObject()
to fail due to excessive GC
```

```
1STGCHTYPE     GC History
3STHSTTYPE     09:59:01:632262775 GMT j9mm.84 -     Forcing
J9AllocateIndexableObject() to fail due to excessive GC
```

When the Sun implementation runs out of Java heap memory, it uses the exception message to show that it is Java heap that is exhausted:

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

IBM and Sun implementations both have a verbose GC option that produces trace data showing how full the heap is at every GC cycle. This information can be plotted with a tool such as the IBM Monitoring and Diagnostic Tools for Java - Garbage Collection and Memory Visualizer (GCMV) to show if the Java heap is growing.

Measuring native heap usage

If you have determined that your out-of-memory condition was not caused by Java-heap exhaustion, the next stage is to profile your native-memory usage.

The PerfMon tool supplied with Windows allows you to monitor and record many OS and process metrics, including native-memory use. It allows *counters* to be tracked in real time or stored in a log file to be reviewed offline. Use the Private Bytes counter to show the total address-space usage. If this approaches the limit of the user space (between 2 and 3GB as previously discussed), you should expect to see native out-of-memory conditions.

Linux has no PerfMon equivalent, but you have several alternatives. Command-line tools such as `ps`, `top`, and `pmap` can show an application's native-memory footprint. Although getting an instant snapshot of a process's memory usage is useful, you get a much greater understanding of how native memory is being used by plotting memory use over time. One way to do this is to use GCMV.

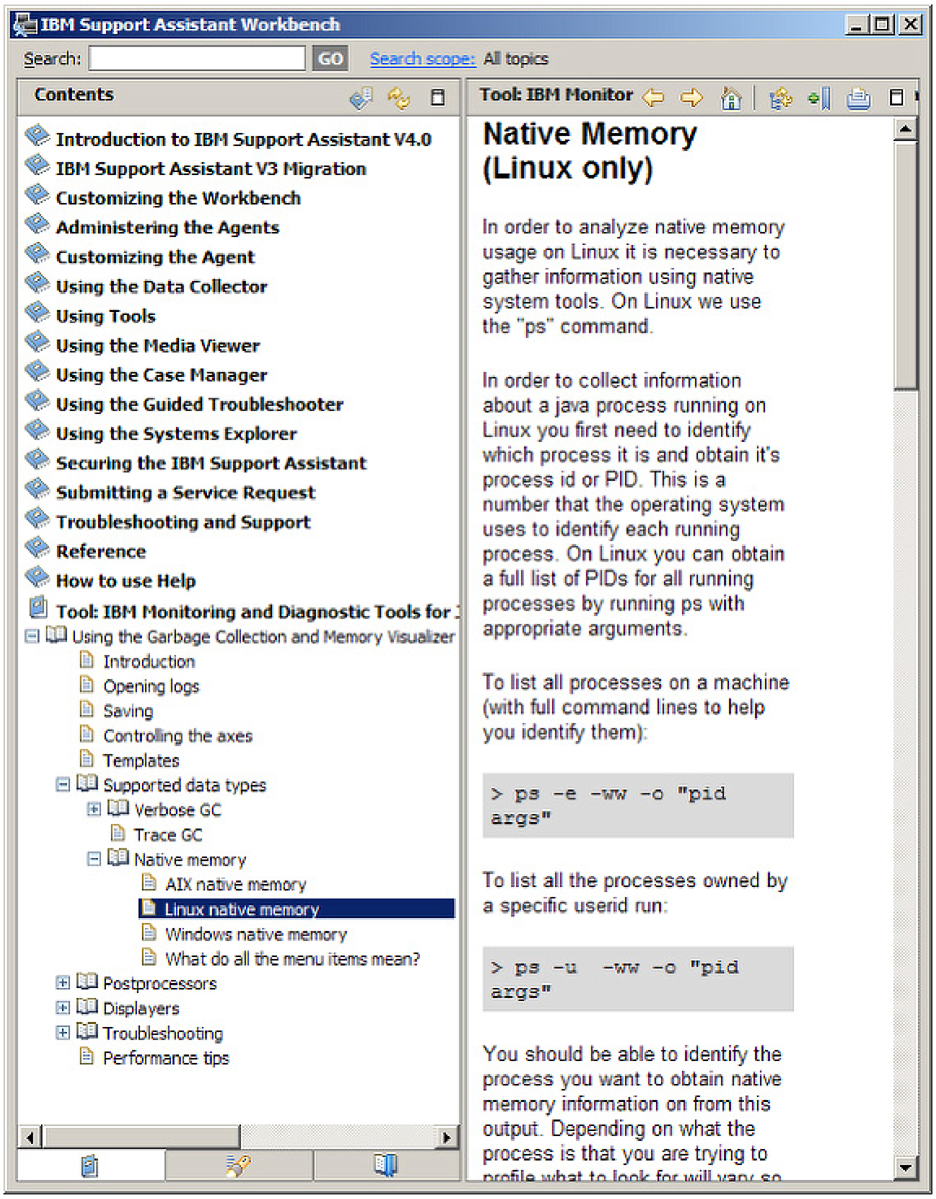
GCMV was originally written to plot verbose GC logs, allowing users to view changes in Java heap usage and GC performance when tuning the garbage collector. GCMV was later extended to allow it to plot other data sources, including Linux and AIX native-memory data. GCMV is shipped as a plug-in for the IBM Support Assistant (ISA).

To plot a Linux native-memory profile with GCMV, you must first collect native-memory data using a script. GCMV's Linux native-memory parser reads output from the Linux `ps` command interleaved with time stamps. A script is provided in the GCMV help documentation that collects data in the correct form. To find the script:

1. Download and install ISA Version 4 (or above) and install the GCMV tool plug-in.
2. Start ISA.
3. Click **Help >> Help Contents** from the menu bar to open the ISA help menu.
4. Find the Linux native-memory instructions in the left-hand pane under **Tool:IBM Monitoring and Diagnostic Tools for Java - Garbage Collection and Memory Visualizer >> Using the Garbage Collection and Memory Visualizer >> Supported Data Types >> Native memory >> Linux native memory**.

Figure 5 shows the location of the script in the ISA help file. If you do not have the GCMV Tool entry in your help file, it is most likely you do not have the GCMV plug-in installed.

Figure 5. Location of Linux native memory data capture script in ISA help dialog



Native Memory (Linux only)

In order to analyze native memory usage on Linux it is necessary to gather information using native system tools. On Linux we use the "ps" command.

In order to collect information about a java process running on Linux you first need to identify which process it is and obtain it's process id or PID. This is a number that the operating system uses to identify each running process. On Linux you can obtain a full list of PIDs for all running processes by running ps with appropriate arguments.

To list all processes on a machine (with full command lines to help you identify them):

```
> ps -e -ww -o "pid
args"
```

To list all the processes owned by a specific userid run:

```
> ps -u  -ww -o "pid
args"
```

You should be able to identify the process you want to obtain native memory information on from this output. Depending on what the process is that you are trying to profile what to look for will vary so

The script provided in the GCMV help uses a `ps` command that only works with recent versions of `ps`. On some older Linux distributions, the command in the help file will not produce the correct information. To check the behaviour on your Linux distribution, try running `ps -o pid,vsz=VSZ,rss=RSS`. If your version of `ps` supports the new command-line argument syntax, the output will look like this:

```
  PID    VSZ   RSS
 5826   3772  1960
 5675   2492   760
```

If your version of `ps` does not support the new syntax, the output will look like this:

```
  PID VSZ,rss=RSS
 5826        3772
 5674        2488
```

If you are running an older version of `ps`, modify the native memory script replacing the line

```
ps -p $PID -o pid,vsz=VSZ,rss=RSS
```

with

```
ps -p $PID -o pid,vsz,rss
```

Copy the script from the help panel into a file (in this example called memscript.sh), find the process id (PID) of the Java process you want to monitor (in this example, 1234), and run:

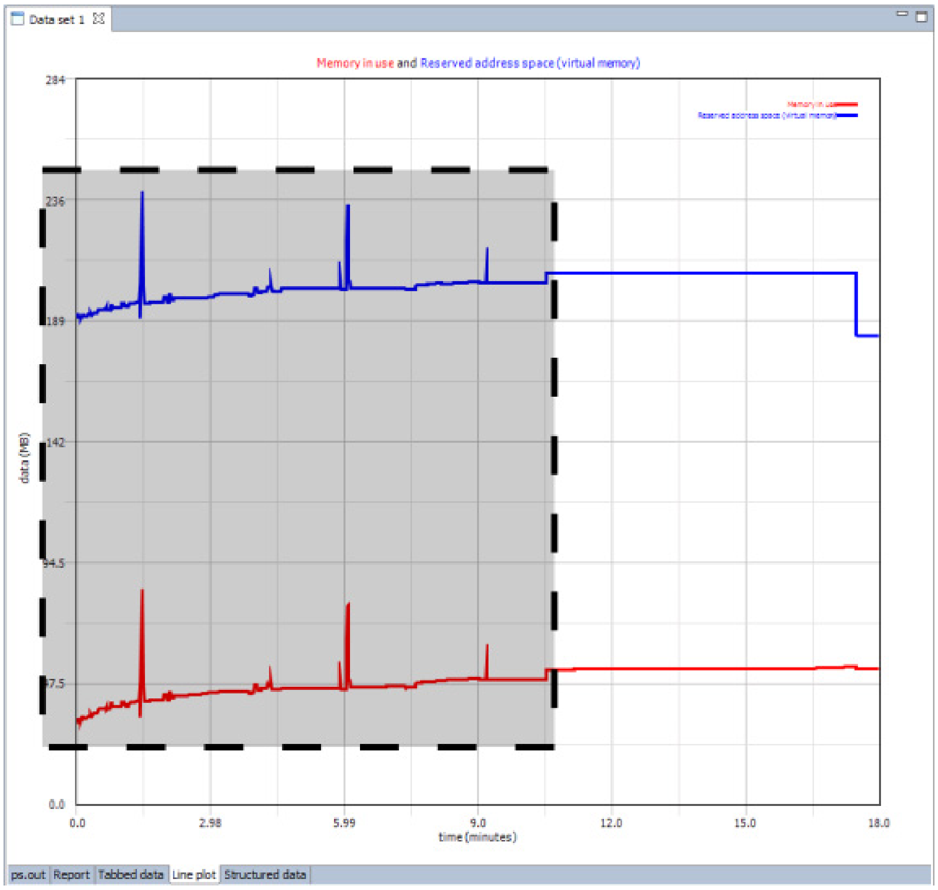
```
./memscript.sh 1234 > ps.out
```

This will write the native-memory log into ps.out. To plot memory usage:

1. In ISA, select **Analyze Problem** from the **Launch Activity** drop-down menu.
2. Select the **Tools** tab near the top of the **Analyze Problem** panel.
3. Select **IBM Monitoring and Diagnostic Tools for Java - Garbage Collection and Memory Visualizer**.
4. Click the **Launch** button near the bottom of the tools panel.
5. Click the Browse button and locate the log file. Click **OK** to launch GCMV.

Once you have the profile of the native-memory use over time, you need to decide whether you are seeing a native-memory leak or are just trying to do too much in the available space. The native-memory footprint for even a well-behaved Java application is not constant from start-up. Several of the Java runtime systems — particularly the JIT compiler and classloaders — initialise over time, which can consume native memory. The memory growth from initialisation will plateau, but if your scenario has an initial native-memory footprint close to the limit of the address space, then this warm-up phase can be enough to cause native out-of-memory. Figure 6 shows an example GCMV native-memory plot from a Java stress test with the warm-up phase highlighted.

Figure 6. Example Linux native-memory plot from GCMV showing warm-up phase



It's also possible to have a native footprint that varies with workload. If your application creates more threads to handle incoming workload or allocates native-backed storage such as direct `ByteBuffer`s proportionally to how much load is being applied to your system, it's possible that you will run out of native memory under high load.

Running out of native memory because of JVM warm-up-phase native-memory growth, and growth proportional to load, are examples of trying to do too much in the available space. In these scenarios your options are:

- **Reduce your native-memory use.** Reducing your Java heap size is a good place to start.
- **Restrict your native-memory use.** If you have native-memory growth that changes with load, find a way to cap the load or the resources that are allocated because of it.
- **Increase the amount of address space available to you.** You can do this by tuning your OS (increasing your user space with the `/3GB` switch on Windows or a hugemem kernel on Linux, for example), changing platform (Linux typically has more user space than Windows), or moving to a 64-bit OS.

A genuine native-memory leak manifests as a continual growth in native heap that doesn't drop when load is removed or when the garbage collector runs. The rate of memory leak can vary with load, but the total leaked memory will not drop. Memory that is leaked is unlikely to be referenced, so it can be swapped out and stay swapped out.

When faced with a leak, your options are limited. You can increase the amount of user space (so there is more room to leak into) but that will only buy you time before you eventually run out of memory. If you have enough physical memory and address space, you can allow the leak to continue on the basis that you will restart your application before the process address space is exhausted.

What's using my native memory?

Once you have determined you are running out of native memory, the next logical question is: What's using that memory? Answering this question is hard because, by default, Windows and Linux do not store information about which code path is allocated a particular chunk of memory.

Your first step when trying to understand where your native memory has gone is to work out roughly how much native memory will be used based on your Java settings. An accurate value is difficult to work out without in-depth knowledge of the JVM's workings, but you can make a rough estimate based on the following guidelines:

- The Java heap occupies at least the `-Xmx` value.
- Each Java thread requires stack space. Stack size varies among implementations, but with default settings, each thread could occupy up to 756KB of native memory.
- Direct `ByteBuffer`s occupy at least the values supplied to the `allocate()` routine.

If your total is much less than your maximum user space, you are not necessarily safe. Many other components in a Java runtime could allocate enough memory to cause problems; however, if your initial calculations suggest you are close to your maximum user space, it is likely that you will have native-memory issues. If you suspect you have a native-memory leak or you want to understand exactly where your memory is going, several tools can help.

Microsoft provides the UMDH (user-mode dump heap) and LeakDiag tools for debugging native-memory growth on Windows. They both work in a similar way: recording which code path allocated a particular area of memory and providing a way to locate sections of code that allocate memory that isn't freed later. I refer you to the article "Umdhtools.exe: How to use Umdh.exe to find memory leaks on Windows" for instructions on using UMDH. In this article, I'll focus on what the output of UMDH looks like when run against a leaky JNI application.

The samples pack for this article contains a Java application called `LeakyJNIApp`; it runs in a loop calling a JNI method that leaks native memory. The UMDH command takes a snapshot of the current native heap together with native stack traces of the code paths that allocated each region of memory. By taking two snapshots and using the UMDH tool to analyse the difference, you get a report of the heap growth between the two snapshots.

For `LeakyJNIApp`, the difference file contains this information:

```
// _NT_SYMBOL_PATH set by default to C:\WINDOWS\symbols
//
```

```
// Each log entry has the following syntax:
//
// + BYTES_DELTA (NEW_BYTES - OLD_BYTES) NEW_COUNT allocs BackTrace TRACEID
// + COUNT_DELTA (NEW_COUNT - OLD_COUNT) BackTrace TRACEID allocations
//     ... stack trace ...
//
// where:
//
//     BYTES_DELTA - increase in bytes between before and after log
//     NEW_BYTES - bytes in after log
//     OLD_BYTES - bytes in before log
//     COUNT_DELTA - increase in allocations between before and after log
//     NEW_COUNT - number of allocations in after log
//     OLD_COUNT - number of allocations in before log
//     TRACEID - decimal index of the stack trace in the trace database
//         (can be used to search for allocation instances in the original
//         UMDH logs).
//

+  412192 ( 1031943 - 619751)    963 allocs     BackTrace00468

Total increase == 412192
```

The important line is `+ 412192 ( 1031943 - 619751) 963 allocs BackTrace00468`. It shows that one backtrace has made 963 allocations that have not been freed — that have consumed 412192 bytes of memory. By looking in one of the snapshot files, you can associate `BackTrace00468` with the meaningful code path. Searching for `BackTrace00468` in the first snapshot shows:

```
000000AD bytes in 0x1 allocations (@ 0x00000031 + 0x0000001F) by: BackTrace00468
        ntdll!RtlpNtMakeTemporaryKey+000074D0
        ntdll!RtlInitializeSListHead+00010D08
        ntdll!wcsncat+00000224
        leakyjniapp!Java_com_ibm_jtc_demos_LeakyJNIApp_nativeMethod+000000D6
```

This shows the leak coming from the leakyjniapp.dll module in the `Java_com_ibm_jtc_demos_LeakyJNIApp_nativeMethod` function.

At the time of writing, Linux does not have a UMDH or LeakDiag equivalent. But there are still ways to debug native-memory leaks on Linux. The many memory debuggers available on Linux typically fall into one of the following categories:

- **Preprocessor level.** These require a header to be compiled in with the source under test. It's possible to recompile your own JNI libraries with one of these tools to track a native memory leak in your code. Unless you have the source code for the Java runtime itself, this cannot find a leak in the JVM (and even then compiling this kind of tool into a large project like a JVM would almost certainly be difficult and time-consuming). Dmalloc is an example of this kind of tool.
- **Linker level.** These require the binaries under test to be relinked with a debugging library. Again, this is feasible for individual JNI libraries but not recommended for entire Java runtimes because it is unlikely that the runtime vendor would support you running with modified binaries. Ccmalloc is an example of this kind of tool.
- **Runtime-linker level.** These use the `LD_PRELOAD` environment variable to preload a library that replaces the standard memory routines with instrumented versions. They do not require recompilation or relinking of source code, but many of them do not work well with Java

runtimes. A Java runtime is a complicated system that can use memory and threads in unusual ways that can confuse or break this kind of tool. It is worth experimenting with a few to see if they work in your scenario. NJAMD is an example of this kind of tool.
- **Emulator-based.** The Valgrind `memcheck` tool is the only example of this type of memory debugger. It emulates the underlying processor in a similar way to how a Java runtime emulates the JVM. It is possible to run Java under Valgrind, but the heavy performance impact (10 to 30 times slower) means that it would be very hard to run large, complicated Java applications in this manner. Valgrind is currently available on Linux x86, AMD64, PPC 32, and PPC 64. If you use Valgrind, try to narrow down the problem to the smallest test case you can (preferably cutting out the entire Java runtime if possible) before using it.

For simple scenarios that can tolerate the performance overhead, Valgrind `memcheck` is the most simple and user-friendly of the available free tools. It can provide a full stack trace for code paths that are leaking memory in the same way that UMDH can on Windows.

The `LeakyJNIApp` is simple enough to run under Valgrind. The Valgrind `memcheck` tool can print out a summary of leaked memory when the emulated program ends. By default, the `LeakyJNIApp` program runs indefinitely; to make it shut down after a fixed time period, pass the run time in seconds as the only command-line argument.

Some Java runtimes use thread stacks and processor registers in unusual ways; this can confuse some debugging tools, which expect native programs to abide by standard conventions of register use and stack structure. When using Valgrind to debug leaking JNI applications, you may find that many warnings are thrown up about the use of memory, and some thread stacks will look odd; these are due to the way the Java runtime structures its data internally and are nothing to worry about.

To trace the `LeakyJNIApp` with the Valgrind `memcheck` tool, use this command (on a single line):

```
valgrind --trace-children=yes --leak-check=full
java -Djava.library.path=. com.ibm.jtc.demos.LeakyJNIApp 10
```

The `--trace-children=yes` option to Valgrind makes it trace any processes that are started by Java launcher. Some versions of the Java launcher reexecute themselves (they restart themselves from the beginning, again having set environment variables to change behaviour). If you don't specify `--trace-children`, you might not trace the actual Java runtime.

The `--leak-check=full` option requests that full stack traces of leaking areas of code are printed at the end of the run, rather than just summarising the state of the memory.

Valgrind prints many warnings and errors while the command runs (most of which are uninteresting in this context), and finally prints a list of leaking call stacks in ascending order of amount of memory leaked. The end of the summary section of the Valgrind output for `LeakyJNIApp` on Linux x86 is:

```
==20494== 8,192 bytes in 8 blocks are possibly lost in loss record 36 of 45
==20494==    at 0x4024AB8: malloc (vg_replace_malloc.c:207)
==20494==    by 0x460E49D: Java_com_ibm_jtc_demos_LeakyJNIApp_nativeMethod
```

```
(in /home/andhall/LeakyJNIApp/libleakyjniapp.so)
==20494==    by 0x535CF56: ???
==20494==    by 0x46423CB: gpProtectedRunCallInMethod
(in /usr/local/ibm-java2-i386-50/jre/bin/libj9vm23.so)
==20494==    by 0x46441CF: signalProtectAndRunGlue
(in /usr/local/ibm-java2-i386-50/jre/bin/libj9vm23.so)
==20494==    by 0x467E0D1: j9sig_protect
(in /usr/local/ibm-java2-i386-50/jre/bin/libj9prt23.so)
==20494==    by 0x46425FD: gpProtectAndRun
(in /usr/local/ibm-java2-i386-50/jre/bin/libj9vm23.so)
==20494==    by 0x4642A33: gpCheckCallin
(in /usr/local/ibm-java2-i386-50/jre/bin/libj9vm23.so)
==20494==    by 0x464184C: callStaticVoidMethod
(in /usr/local/ibm-java2-i386-50/jre/bin/libj9vm23.so)
==20494==    by 0x80499D3: main
(in /usr/local/ibm-java2-i386-50/jre/bin/java)
==20494==
==20494==
==20494== 65,536 (63,488 direct, 2,048 indirect) bytes in 62 blocks are definitely
lost in loss record 42 of 45
==20494==    at 0x4024AB8: malloc (vg_replace_malloc.c:207)
==20494==    by 0x460E49D: Java_com_ibm_jtc_demos_LeakyJNIApp_nativeMethod
(in /home/andhall/LeakyJNIApp/libleakyjniapp.so)
==20494==    by 0x535CF56: ???
==20494==    by 0x46423CB: gpProtectedRunCallInMethod
(in /usr/local/ibm-java2-i386-50/jre/bin/libj9vm23.so)
==20494==    by 0x46441CF: signalProtectAndRunGlue
(in /usr/local/ibm-java2-i386-50/jre/bin/libj9vm23.so)
==20494==    by 0x467E0D1: j9sig_protect
(in /usr/local/ibm-java2-i386-50/jre/bin/libj9prt23.so)
==20494==    by 0x46425FD: gpProtectAndRun
(in /usr/local/ibm-java2-i386-50/jre/bin/libj9vm23.so)
==20494==    by 0x4642A33: gpCheckCallin
(in /usr/local/ibm-java2-i386-50/jre/bin/libj9vm23.so)
==20494==    by 0x464184C: callStaticVoidMethod
(in /usr/local/ibm-java2-i386-50/jre/bin/libj9vm23.so)
==20494==    by 0x80499D3: main
(in /usr/local/ibm-java2-i386-50/jre/bin/java)
==20494==
==20494== LEAK SUMMARY:
==20494==    definitely lost: 63,957 bytes in 69 blocks.
==20494==    indirectly lost: 2,168 bytes in 12 blocks.
==20494==      possibly lost: 8,600 bytes in 11 blocks.
==20494==    still reachable: 5,156,340 bytes in 980 blocks.
==20494==         suppressed: 0 bytes in 0 blocks.
==20494== Reachable blocks (those to which a pointer was found) are not shown.
==20494== To see them, rerun with: --leak-check=full --show-reachable=yes
```

The second line of the stacks shows that the memory was leaked by the `com.ibm.jtc.demos.LeakyJNIApp.nativeMethod()` method.

Several proprietary debugging applications that can debug native-memory leaks are also available. More tools (both open-source and proprietary) are being developed all the time, and it's worth researching the current state of the art.

Currently debugging native-memory leaks on Linux with the freely available tools is more challenging than doing the same on Windows. Whereas UMDH allows native leaks on Windows to be debugged *in situ*, on Linux you will probably need to do some traditional debugging rather than rely on a tool to solve the problem for you. Here are some suggested debugging steps:

- **Extract a test case.** Produce a stand-alone environment that you can reproduce the native leak with. It will make debugging much simpler.
- **Narrow the test case as far as possible.** Try stubbing out functions to identify which code paths are causing the native leak. If you have your own JNI libraries, try stubbing them out entirely one at a time to determine if they are causing the leak.
- **Reduce the Java heap size.** The Java heap is likely to be the largest consumer of virtual address space in the process. By reducing the Java heap, you make more space available for other users of native memory.
- **Correlate the native process size.** Once you have a plot of native-memory use over time, you can compare it to application workload and GC data. If the leak rate is proportional to the level of load, it suggests that the leak is caused by something on the path of each transaction or operation. If the native process size drops significantly when a GC happens, it suggests that you are not seeing a leak — you are seeing a buildup of objects with a native backing (such as direct `ByteBuffer`s). You can reduce the amount of memory held by native-backed objects by reducing the Java heap size (thereby forcing collections to occur more frequently) or by managing them yourself in an object cache rather than relying on the garbage collector to clean up for you.

If you identify a leak or memory growth that you think is coming out of the Java runtime itself, you may want to engage your runtime vendor to debug further.

Removing the limit: Making the change to 64-bit

It is easy to hit native out-of-memory conditions with 32-bit Java runtimes because the address space is relatively small. The 2 to 4GB of user space that 32-bit OSs provide is often less than the amount of physical memory attached to the system, and modern data-intensive applications can easily scale to fill the available space.

If your application cannot be made to fit in a 32-bit address space, you can gain a lot more user space by moving to a 64-bit Java runtime. If you are running a 64-bit OS, then a 64-bit Java runtime will open the door to huge Java heaps and fewer address-space-related headaches. Table 2 lists the user spaces currently available with 64-bit OSs:

Table 2. User space sizes on 64-bit OSs

OS	Default user space size
Windows x86-64	8192GB
Windows Itanium	7152GB
Linux x86-64	500GB
Linux PPC64	1648GB
Linux 390 64	4EB

Moving to 64-bit is not a universal solution to all native-memory problems, however; you still need sufficient physical memory to hold all of your data. If your Java runtime won't fit in physical memory then performance will be intolerably poor because the OS is forced to thrash Java runtime data back and forth from swap space. For the same reason, moving to 64-bit is no permanent solution to a memory leak — you are just providing more space to leak into, which will only buy time between forced restarts.

It's not possible to use 32-bit native code with a 64-bit runtime; any native code (JNI libraries, JVM Tool Interface [JVMTI], JVM Profiling Interface [JVMPI], and JVM Debug Interface [JVMDI] agents) must be recompiled for 64-bit. A 64-bit runtime's performance can also be slower than the corresponding 32-bit runtime on the same hardware. A 64-bit runtime uses 64-bit pointers (native address references), so the same Java object on 64-bit takes up more space than an object containing the same data on 32-bit. Larger objects mean a bigger heap to hold the same amount of data while maintaining similar GC performance, which makes the OS and hardware caches less efficient. Surprisingly, a larger Java heap does not necessarily mean longer GC pause times, because the amount of live data on the heap might not have increased, and some GC algorithms are more effective with larger heaps.

Some modern Java runtimes contain technology to mitigate 64-bit "object bloat" and improve performance. These features work by using shorter references on 64-bit runtimes. This is called *compressed references* on IBM implementations and *compressed oops* on Sun implementations.

A comparative study of Java runtime performance is beyond this article's scope, but if you are considering a move to 64-bit it is worth testing your application early to understand how it performs. Because changing the address size affects the Java heap, you will need to retune your GC settings on the new architecture rather than just port your existing settings.

Conclusion

An understanding of native memory is essential when you design and run large Java applications, but it's often neglected because it is associated with the grubby hardware and OS details that the Java runtime was designed to save us from. The JRE is a native process that must work in the environment defined by these grubby details. To get the best performance from your Java application, you must understand how the application affects the Java runtime's native-memory use.

Running out of native memory can look similar to running out of Java heap, but it requires a different set of tools to debug and solve. The key to fixing native-memory issues is to understand the limits imposed by the hardware and OS that your Java application is running on, and to combine this with knowledge of the OS tools for monitoring native-memory use. By following this approach, you'll be equipped to solve some of the toughest problems your Java application can throw at you.

Downloadable resources

Description	Name	Size
Native memory example code	j-nativememory-linux.zip	115KB

Related topics

- Garbage collection with the IBM Monitoring and Diagnostic Tools for Java - garbage collection and memory visualizer
- Don't forget about memory: How to monitor your Java applications' Windows memory usage
- The Support Authority: Introducing the IBM Guided Activity Assistant
- IBM Support Assistant