Windows Java address space

This article applies to the IBM 32-bit SDK and Runtime Environment for Windows, Java2 Technology Edition. It explains how the process space for Java is divided and explores a way of increasing the available space.

How is the space in a basic Win32 process allocated ?

A 32-bit architecture imposes a fundamental limit on an address range of 4 GB. Hence, on a 32-bit platform any given process can theoretically achieve a maximum size of 4 GB.

Unfortunately, on Win-32, the process space available to an application is only 2 GB. Why is this ?

When the NT kernel was being designed, it was decided to conceptually split the process space into two. One half for the application and one half for use by the OS. At the time 2 GB must have seemed far more space than anyone could possibly use. This split had certain advantages, in that the internal address tables used by NT only needed to be 31 bits wide.

Inside this 2 GB space, the process code and its memory requirements are loaded at the 'bottom' of the space. (For the purpose of this article we will refer to virtual address 0x00000000 as the 'bottom' of the process space and address 0x7FFFFFFF as the 'top' of the space.)

Every application has at least one thread. A thread has an associated stack, which is storage space allocated on its behalf from the process space. In addition, most applications will use dynamic memory space, either explicitly by calls such as 'malloc' and 'free', or implicitly for example by the C++ 'new' operator. Stack and dynamic memory storage is managed by Windows by means of a 'heap'. The heap is usually loaded above the application code and grows upwards.

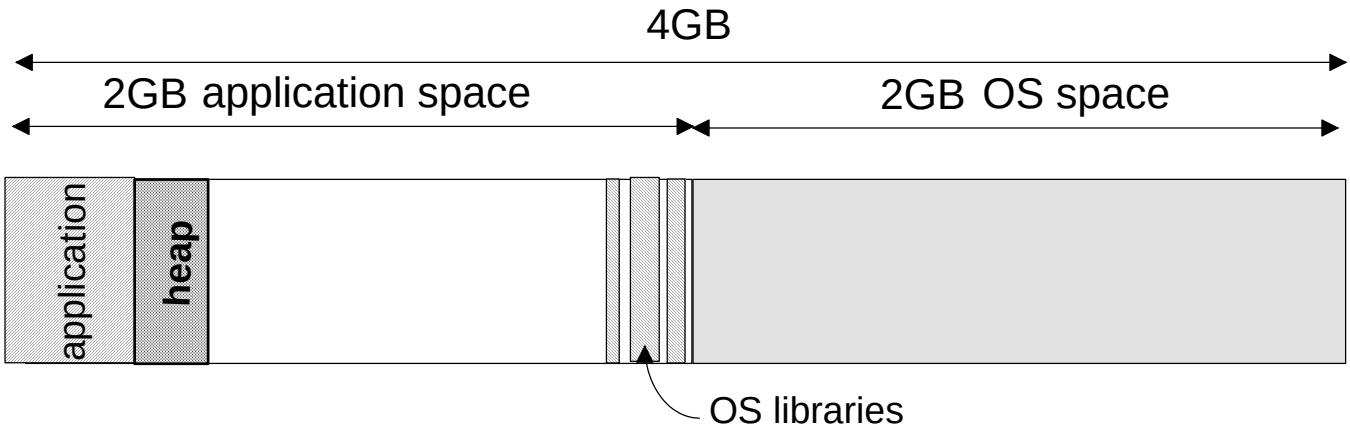
When linking your application, you need to link in certain OS libraries. A Java process loads a large number of Java libraries - the JVM executable is very small, most of the VM comprises Dynamic Link Libraries (DLLs). In addition, most applications link in application libraries. For the purposes of this article, we consider application native code libraries to be part of the application; they are loaded at addresses determined by the application. In the case of the VM, the VM link

libraries are sometimes loaded at the bottom of the process space or sometimes not, depending on whether you are in a WebSphere environment and the version of the JVM you are using. The OS libraries are loaded at the top of the process 2 GB space (address 0x70000000 and upwards). These libraries are important when considering the memory map of a Java process as we shall see.

The 32-bit Windows secret !

It's a little known 'secret', but the vast majority of Windows applications are 31-bit code, not 32-bit ! An application's address range is from 0x00000000 to 0x7FFFFFFF. A default Windows process can never have a 'negative' pointer (where the most significant bit of the pointer is set).

Structure of a standard Win32 process



What are the Win32 java process memory requirements ?

A Java process executable code is the Java Virtual Machine (VM). It has the same requirements as any other Windows process, but in addition needs memory space for

- Application byte codes (Java classes)
- The Java heap
- JIT-compiled code

Java Virtual Machine

The virtual machine (VM) is the mechanism for executing the byte codes in your java application. It is just a C application.

Java heap

The Java heap is where the VM stores Java objects. It should not be confused with the standard Windows heap space we already alluded to. A Java process therefore has two heaps. To distinguish them we refer to the 'Java heap' and the 'native heap'. The Java heap is a single contiguous chunk of memory. It is a single malloc'd chunk of memory which is never freed and is internally managed by the VM storage component, the Garbage Collector.

The Java heap is allocated as a single chunk of memory whose size is specified by the ***-Xmx*** command line parameter. If you don't specify ***Xmx*** it takes a default value of half of available physical memory up to a maximum of (2 GB - 1) bytes. It is allocated as virtual memory. The amount of physical memory initially committed to the heap is specified by the ***-Xms*** parameter. As with ***Xmx***, this parameter takes a default if not specified.

> ***Tip !*** Specify the max heap size you want with ***-Xmx***. Specify ***Xms*** as a low value. These settings allow the Java process to start with the minimum amount of physical memory and, as the heap grows, the Garbage Collector can optimise it. It is rarely a good idea to specify a large value for ***Xms***.

JIT compiled code

A Java process interpreting Java byte codes cannot approach the performance of compiled code. A VM has a bundled compiler that compiles Java byte codes into true platform code. Because the compiler is invoked at runtime and compiles the code 'just in time' to execute, it is known as a Just-In-Time compiler or more commonly by its initials as a JIT compiler. In fact, we usually just refer to the JIT compiler as 'the JIT' and to JIT-compiled code as 'JIT'd code'.

It is important to note that the JIT does NOT compile Java byte codes the first time those byte codes are executed. If it did this, then the VM would take a very long time to start up, as thousands of methods were

JIT'd. Also, when a large amount of function was loaded there would be a big hit on performance as the JIT laboured to compile all the code. For these reasons, Java byte codes are only JIT'd when they have been executed a certain number of times - the VM imposes a JIT threshold value. On Windows, this threshold varies according to the version of the VM but is of the order of several thousand.

Now, when the JIT runs it generates some executable code. But where does it put it ? Clearly, this must come out of the process space. What this means is that the Java VM will make ever-increasing demands for storage space for JIT'd code as more and more methods hit the JIT threshold. Eventually, all but rarely used methods get compiled and the demands of the JIT for storage level off.

Java threads

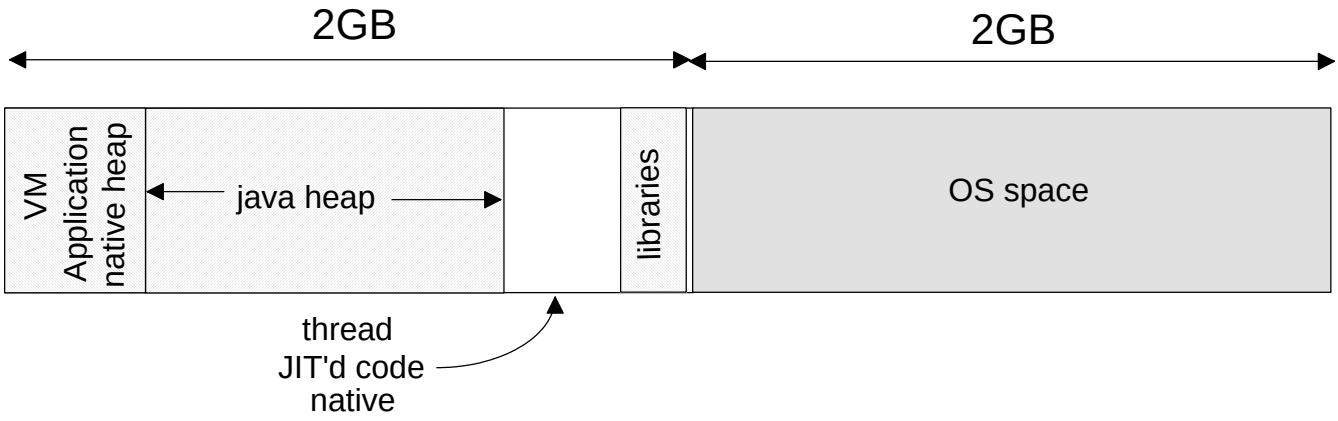
A Java thread is a wrapper for a real native (windows) thread. Recall that all threads take stack space out of the process space. So lots of Java threads take lots of storage in the process 2 GB space. The Windows VM prior to Java 5 allocates an initial 1 MB of stack space to a thread. Java 5 initially allocates 32 K to a thread.

In summary, a Java process comprises

- Java VM
- Application byte code
- Native heap space
- Java heap space
- JIT'd code space
- Link libraries

Given that everything has to fit into 2 GB the Java heap theoretical maximum is around 1.75 GB, but most applications will get out-of-memory errors at this setting, because of failures to get native memory. By the time all the code is loaded and space reserved for JIT'd code, the practical maximum size of the Java heap is around 1.5 GB.

So, a java process looks like this



Hitting the buffers

As applications grow in size and complexity, the demands on the Java heap grow. Recall that the Java heap must be a contiguous space and that its practical max size is around 1.5 GB. Many applications are now finding themselves approaching this limit. If an application needs, in addition, lots of threads and/or lots of native heap (in native code or used by JNI code) then that 2 GB space starts to get rather constricted.

Can we extend the Win-32 process space ?

Yes we can. But not on all versions of Windows. There are two mechanisms

1. Microsoft Physical Address Extensions (PAE)
2. The /LARGEADDRESSAWARE switch

PAE is a mechanism by which processes can effectively address more than 4 GB. It is supported only on certain platforms and requires use of a formal Microsoft API. The IBM VM does not currently support PAE. Even if it did, application native code would also need to be written to PAE.

The /LARGEADDRESSAWARE switch is often referred to as the /3GB switch. In a nutshell, using this switch redefines the share of the 4 GB Win32 process as 3 GB for the application and 1 GB for the OS.

This seems like a good idea. But it does have some problems.

1. Microsoft only support use of this switch on certain Windows versions (see later). On other platforms, it is unsupported.

2. With the /3GB switch in place, an application's address range is from 0x0000000 to 0xBFFFFFFF. This has some 'gotchas' for applications that play with pointers if there's an implicit assumption the top bit is clear. Microsoft have documented some of these here

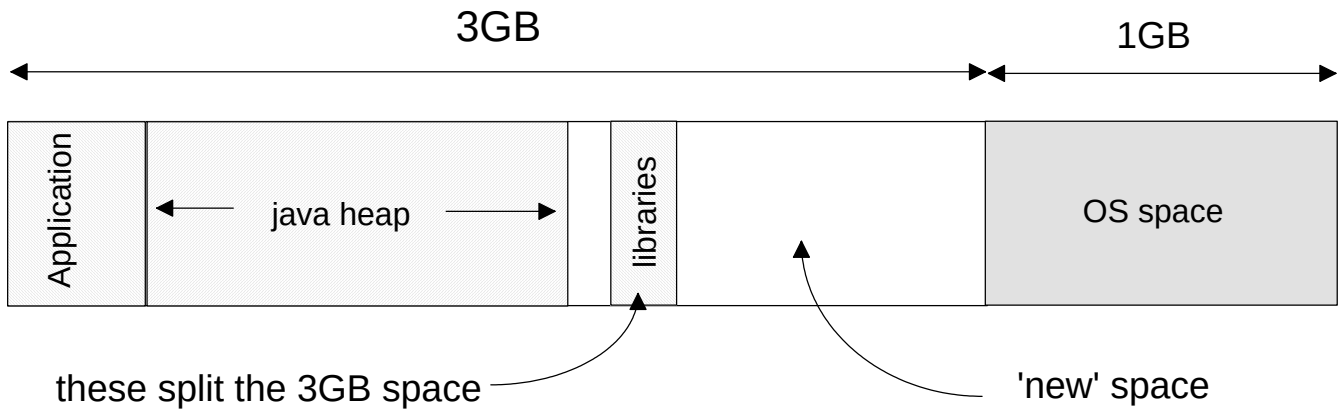
   http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/4gt_ram_tuning.asp

   The Java language has no pointers, so your Java code is quite safe. But if you have any native code in your application that does pointer arithmetic such as

   ```
   pNext = pStart + pCurrent;
   ```

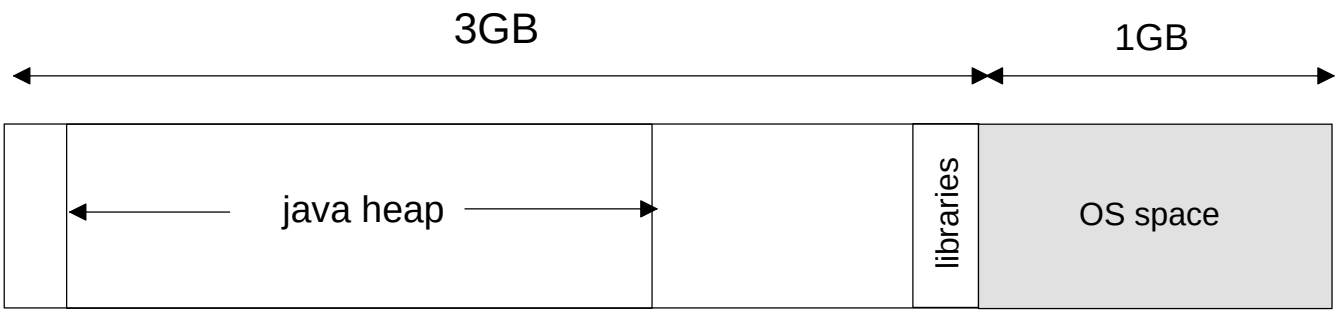
   then, clearly, problems might occur if the pointers are effectively 'signed' quantities.

3. The /3GB switch does not automatically rebase the OS link libraries. Hence you end up with a Win32 Java process looking like this



Because the libraries split the 3 GB space the Java heap cannot grow much, because it requires a contiguous area. Therefore the /3GB switch does NOT provide for a much bigger Java heap.

Those link libraries are loaded at link time and take default base addresses. It is possible to 'rebase' the libraries when a process is built and redefine the library base addresses, so the Java executable could be linked to look like this



Rebasing could extend the java heap up to its theoretical limit. So why don't we do it ?

The design of Windows is such that the OS link libraries are loaded into physical memory and then all processes access that copy provided that the process link libraries are linked at the default base. If they are rebased, then Windows loads separate copies of the OS code for that process. This occupies extra physical memory and increases paging for the process. Performance is severely degraded. And, in any event, the Win32 Garbage Collector design imposes a maximum heap size of (2 GB - 1) bytes. Rebasing would provide only a small heap size advantage. Rebasing is not a worthwhile proposition.

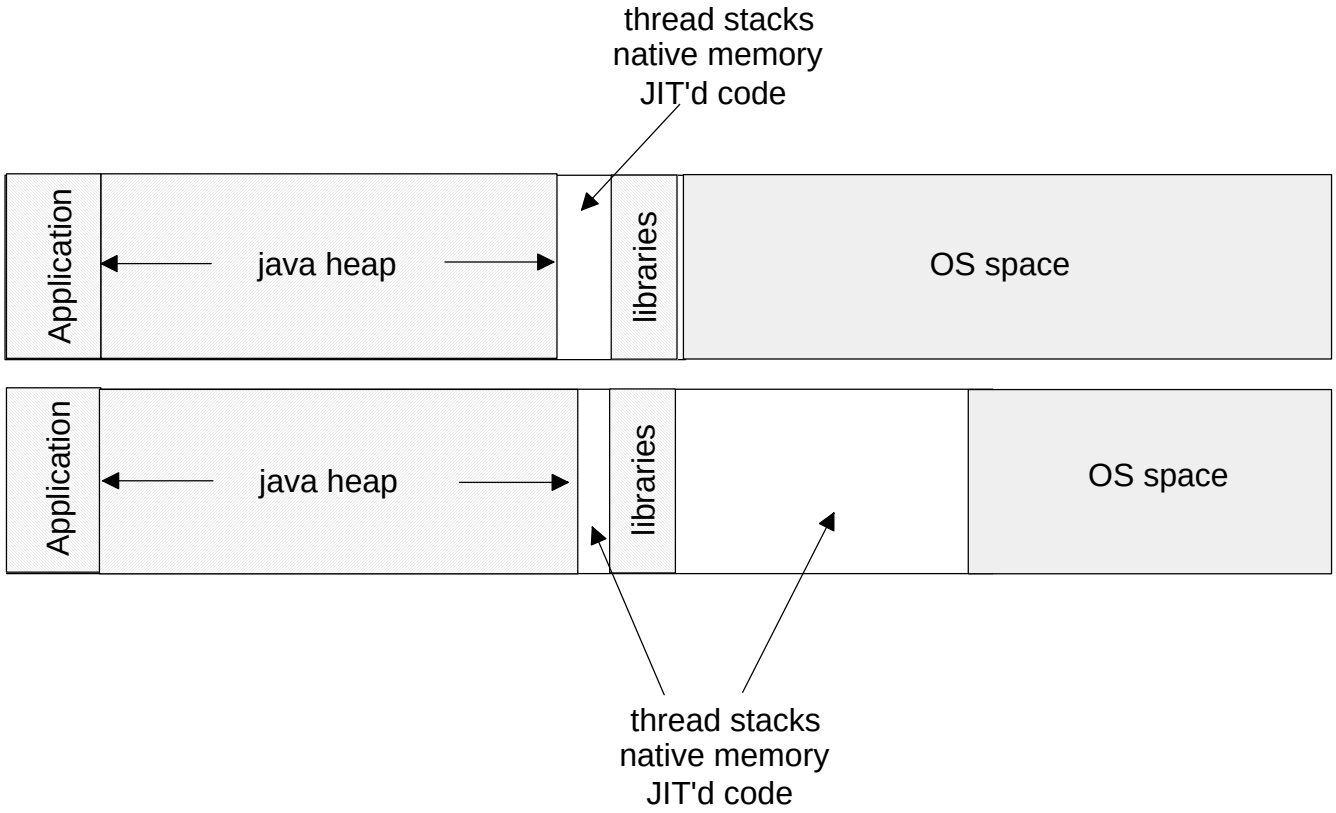
So what does the /3GB switch do for Java then ?

The /3GB switch does enable the Java heap to grow a bit from a practical max of about 1.5 GB up to 1.7 GB or maybe even 1.8 GB. If you are already struggling to fit everything into your Java heap, the /3GB switch is not going to be of much help.

The key advantage lies in the extra space available for threads (stack space), and native code (either for JIT'd code or application requirements).

A big java process

The picture below compares the memory usage of a Java process with a big Java heap, with and without the /3GB switch



Which IBM JVMs support the /3GB switch ?

The /3GB switch can be used on any NT-kernel based version of Windows, - Windows NT, Windows 2000, Windows XP and Windows 2003. However; as stated previously, it is not supported except as defined here

http://support.microsoft.com/kb/291988/

The IBM Windows VM supports the /3GB switch as follows

Java version	/3GB switch ?	Associated Websphere version
1.3.1	No	4.0.3 - 5.0.2
1.4.2	From SR4	5.1.0 - 6.0.x
5.0 and beyond	Yes	6.1 and beyond

How do I get this extra space goodness ?

The extra space available by using the /3GB switch does NOT automatically appear when you start such a Windows process. By default, Windows ignores the /3GB switch. To make Windows check for and effect the /3GB switch, you must edit your BOOT.INI file. The edit has to be done manually on an as-needed basis; the Java install process will not make this change for you. The edit is described here

http://www.microsoft.com/whdc/system/platform/server/PAE/PAEmem.mspx

After BOOT.INI is changed, all processes (not just Java) linked with the /LARGEADDRESSAWARE option will have access to the extra 1 GB of space. Note that this link option merely sets a bit in the internal process header. The Microsoft 'editbin' tool can also be used to flip this bit in a process binary if you do not control access to the source. EDITBIN is a tool provided with Microsoft Visual C toolkits. There is also a freeware licenced version bundled with the MASM32 project

http://www.masm32.com

How can I tell my process is LARGEADDRESSAWARE ?

The counterpart to the EDITBIN tool is the DUMPBIN tool, available from the same sources. If you run DUMPBIN against an executable like this

```
>dumpbin /headers java.exe
```

Then you should see a line something like the one shown below in the process ‘characteristics’

```
Application can handle large (>2GB) addresses
```

If this characteristic is missing, then the process can exist only in the 2:2 GB split configuration. But, as noted, the presence of this characteristic does not, in itself, cause the process 3:1 GB split - BOOT.INI must also be changed.

Summary

- If you are having problems sharing a large Java heap with one or more of the following conditions

    - I need lots of threads
    - I need lots of dynamic memory
    - I have a lot of java code and it takes a lot of JIT'd space

    .. then the /3GB switch is for you !

- To get extra space in your Java process

    1. Install a /3GB version of the Windows VM
    2. Edit your BOOT.INI (then reboot, of course !)

- **But note**

    1. We will support the /3GB switch only on platforms where Microsoft also supports it.
    2. The maximum Java heap size will not increase much when the /3GB switch is used
    3. If you are inclined to try it out for yourself, albeit in an unsupported mode for Java, then you can use the EDITBIN tool as noted above.
    4. Watch out for 'gotchas' with "negative" pointers in your native code.

Phil Vickers Java Technology Centre, Hursley, IBM UK
Amar Devegowda Java Technology Centre, Bangalore, IBM India
Contact: philvickers@uk.ibm.com or adevegow@in.ibm.com
December 2005