Don't forget about memory

How to monitor your Java applications' Windows memory usage

Emma Shepherd
Martin Trotter
Caroline Maynard
Matthew Peters

November 16, 2004

Even though the Java™ runtime handles the bulk of memory management, keeping a vigilant eye on your programs' memory usage is still crucial to optimizing machine performance and spotting memory leaks. Numerous tools are available for monitoring memory usage in Windows. But each has its own advantages and disadvantages and its own particular slant -- often without defining clearly what it measures. The authors clear up a few common misconceptions about memory usage and demystify some of the many available tools, offering guidance on when and how to use them.

One of Java technology's well-known advantages is that the Java programmer -- unlike, say, a C programmer -- doesn't have the daunting responsibility of allocating and freeing required memory. The Java runtime simply manages these tasks for you. Memory is allocated automatically in the heap area for each instantiated object, and the garbage collector periodically reclaims memory occupied by objects that are no longer needed. But you're not completely off the hook. You still need to monitor your program's memory usage, because a Java process's memory is made up of more than objects floating around in the heap. It also consists of the bytecode for the program (instructions the JVM interprets at runtime), JIT code (code that's already been compiled for the target processor), any native code, and some metadata that the JVM uses (exception tables, line number tables and so on). To complicate matters, certain types of memory, such as native libraries, can be shared between processes, so determining a Java application's true footprint can be a difficult task.

Tools for monitoring memory usage under Windows abound, but unfortunately no single tool gives you all the information you need. What's worse, the variety of tools out there don't even share a common vocabulary. But help has arrived. This article introduces some of the most useful freely available tools and provides some tips on how to use them.
Windows memory: A whirlwind tour

Before you can understand the tools we discuss in this article, you need a basic understanding of how Windows manages memory. Windows uses a demand-paged virtual memory system. Read on for a crash course.

The virtual address space

The concept of virtual memory was born in the 1950s as a solution to the complex problem of how to deal with a program that won't all fit into real memory at once. In a virtual-memory system, programs are given access to a larger set of addresses than is physically available, and a dedicated memory manager maps these logical addresses to actual locations, using temporary storage on disc to hold the overflow.

In the modern implementation of virtual memory that Windows uses, virtual storage is organized into equal-sized units known as pages. Each operating-system process is allocated its own virtual address space -- a set of virtual-memory pages that it can read from and write to. Each page can be in one of three states:

- **Free**: The process is not yet using that area of the address space. Any attempt to access that area for reading or writing causes some kind of runtime failure. A Windows dialog box will probably pop up to say that an access violation has occurred. (Your Java program can't make this kind of mistake; only a program written in a language that supports pointers can.)
- **Reserved**: This area of the address space has been reserved for future use by the process but can't be accessed until it has been committed. A lot of the Java heap starts off as reserved.
- **Committed**: Memory that can be accessed by the program and is fully backed, which means that page frames have been allocated for it in the paging file. Committed pages are loaded into main memory only when the process first references them. Hence the name on-demand paging.

Figure 1 illustrates how virtual pages in a process's address space are mapped to physical page frames in memory.
Figure 1. Mapping of virtual pages in a process's address space to physical page frames

If you're running on a 32-bit machine (a normal Intel processor, for example), the total virtual address space for a process is 4GB, because 4GB is the largest number you can address with just 32 bits. Windows doesn't usually let you access all of the memory in this address space; your process gets just under half for its own private use, and Windows uses the rest. The 2GB private area contains most of the memory the JVM needs in order to execute your program: the Java heap, the C heap for the JVM itself, stacks for program threads, memory for holding bytecode and JITted methods, memory that native methods allocate, and more. We'll identify some of these in an address-space map later on in this article.

Programs that want to allocate a large and contiguous area of memory but don’t need it all immediately often use a combination of reserved and committed memory. The JVM allocates the Java heap in this way. The `-mx` parameter to the JVM tells it what the maximum size of the heap should be, but the JVM often doesn't allocate all that memory at the start. It reserves the amount specified by `-mx`, marking the full range of addresses as available to be committed. It then commits just a part of the memory, and it is only for this part that the memory manager needs to allocate pages in real memory and in the paging file to back them up. Later, when the amount of live data grows and the heap needs to be expanded, the JVM can commit a little more, adjacent to the currently committed area. This way the JVM can maintain a single contiguous heap while growing it as needed (see Related topics for an article on how to use the JVM heap parameters).

Real memory

Physical storage is also organized into equal-sized units, most commonly known as page frames. An operating-system data structure called the page table maps the virtual pages accessed by applications to real page frames in main memory. The pages that can't fit are kept in temporary
paging files on disc. When a process tries to access a page that's not currently in memory, a page fault occurs that causes the memory manager to retrieve it from the paging file and put it back into main memory -- a task known as paging. The precise algorithm used to decide which page should be swapped out depends on the version of Windows you are using; it's probably a variation of the least-recently-accessed method. It is also important to note that Windows allows page frames to be shared between processes, for example for DLLs, which are often used by several applications at once. Windows does this by mapping multiple virtual pages from different address spaces to the same physical location.

An application is blissfully unaware of all this activity. All it knows about is its own virtual address space. However, an application soon begins to suffer a marked degradation in performance if the set of its pages currently in main memory, known as the resident set, is smaller than the set of pages it actually needs to use, known as the working set. (Unfortunately, as you'll see throughout this article, the tools we'll discuss often use these two terms interchangeably, even though they refer to quite different things.)

**Task Manager and PerfMon**

We'll take a look first at the two most common tools, Task Manager and PerfMon. They're both bundled with Windows, so it'll be easy for you to get started with them.

**Task Manager**

Task Manager is a fairly simple Windows process monitor. You can access it with the familiar Ctrl-Alt-Delete key combination, or by right-clicking on the Taskbar. The Processes tab shows the most detailed information, as illustrated in Figure 2.
Figure 2. The Task Manager Processes tab

The columns that Figure 2 displays have been customized by selecting View > Select Columns. Some of the column headings have fairly cryptic names, but you can find a definition of each one in the Task Manager help. The most relevant counters for a process's memory usage are:

- **Mem Usage**: The online help calls this the working set of the process (although many would call it the resident set) -- the set of pages currently in main memory. However, the catch is that it includes pages that can be shared by other processes, so you must be careful not to double-count. If you're trying to find out the combined memory usage of two processes that use a common DLL, for example, you can't simply add their Mem Usage values.

- **Peak Mem Usage**: The highest value of the Mem Usage field since the process started.

- **Page Faults**: The total number of times since the process started that a page was accessed that wasn't in main memory.

- **VM Size**: The online help describes this as the "total private virtual memory allocated by the process." To be clear, this is the private memory that has been committed by the process. This can be quite different from the size of the total address space if the process reserves memory but doesn't commit it.

Although the Windows documentation refers to Mem Usage as the working set, it might be helpful to understand that in this context it really means what many people call the resident set. You'll find definitions of these terms in the Memory Management Reference's glossary (see Related topics). Working set more commonly means the logical concept of which pages the process would need to have in memory at that point to avoid any paging.
**PerfMon**

Another tool Microsoft ships with Windows is PerfMon, which monitors a wide variety of counters, from print queues to telephony. PerfMon is normally on the system path, so you can start it by entering `perfmon` from a command line. The advantage of this tool is that it displays the counters graphically, so you can easily see how they change over time.

To get going, click on the + button on the toolbar at the top of the PerfMon screen. This brings up a dialog that lets you choose the counters you want to monitor, as in Figure 3a. The counters are grouped into categories known as *performance objects*. The two that are relevant for memory usage are **Memory** and **Process**. You can get a definition of a counter by highlighting it and clicking on the **Explain** button. The explanation then appears in a separate window that pops up below the main dialog box, as shown in Figure 3b.

**Figure 3a. PerfMon counters window**

![Add Counters dialog](image)

**Figure 3b. Explanation**

![Explain Text window](image)

Select the counters you're interested in (using Ctrl to highlight multiple rows) and also the instance you want to monitor -- the Java process for the application you're examining -- and then click on **Add**. The tool immediately begins to display the values of all the counters you've chosen. You can show them as a report, as a graph over time, or as a histogram. Figure 4 shows the histogram display.
Figure 4. PerfMon histogram

If you don't see anything being graphed, you might need to change the scale by right-clicking on the graph area, selecting Properties, and then navigating to the Graph tab. Or to change the scale of a particular counter, go to its Data tab.

Counters to watch out for
Unfortunately, PerfMon uses different terminology from Task Manager. Table 1 gives you a quick summary of the most useful counters and, if applicable, each one's Task Manager equivalent:

Table 1. Useful PerfMon memory counters

<table>
<thead>
<tr>
<th>Counter name</th>
<th>Category</th>
<th>Description</th>
<th>Task Manager equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Working Set</td>
<td>Process</td>
<td>Resident set -- how many pages are currently in real memory</td>
<td>Mem Usage</td>
</tr>
<tr>
<td>Private Bytes</td>
<td>Process</td>
<td>Total private virtual memory allocated, meaning committed memory</td>
<td>VM Size</td>
</tr>
<tr>
<td>Virtual Bytes</td>
<td>Process</td>
<td>Total size of the virtual address space, including shared pages. Can be much larger than either of the previous two values because it includes reserved memory.</td>
<td>--</td>
</tr>
<tr>
<td>Page Faults / sec</td>
<td>Process</td>
<td>Average number of page faults that Linked to Page Faults, which shows the total</td>
<td></td>
</tr>
</tbody>
</table>
Try an example
You can explore how these quantities appear in Task Manager and in PerfMon by downloading and running a small program we've written in C (see the Download section). The program uses the Windows VirtualAlloc call first to reserve, then to commit, memory. Finally, it starts to touch some of the memory, writing a value into it every 4,096 bytes, to bring pages into the working set. If you run the program and watch it with Task Manager or PerfMon, you'll see the values change.

Useful tools on the Web
Now that you know how much memory your application is using, it's time to dive into the details of the actual memory contents. This section introduces some slightly more sophisticated tools, discusses when it's appropriate to use them, and explains how to interpret their output.

PrcView
PrcView is the first tool we'll show you that lets you inspect the contents of a process's address space (see Related topics). You can use it to do more than look at footprint. It can set priorities and kill processes, and it also exists in a useful command-line version that lists properties of the processes on your machine. But we'll show you how to use it to look at footprint.

When you start PrcView, it shows a Task Manager-like view of the processes in the system. If you scroll to and highlight a Java process, the screen looks like the example in Figure 5.

Figure 5. Initial PrcView screen
Right-clicking on the Java process to bring up a pop-up menu, or choosing Process from the top menu bar, lets you inspect a few things about the process -- which threads it owns and what DLLs it has loaded -- and lets you kill it or set its priority. The option we are interested in is to inspect its memory, which brings up a screen like the one in Figure 6.

**Figure 6. Inspecting a process's memory**

Now you can examine the first few lines of the address-space map that PrcView displays. The first line tells you that from address 0, for a length of 65,536 (64K), the memory is free. Nothing is allocated, nor can the addresses be used. The second line tells you that starting immediately after, at address 0x00010000, is an 8,192-byte stretch (two 4K pages) of committed memory -- memory that can be addressed and that is backed by page frames in the paging file. Then there's another free stretch, then another committed stretch, and so on.

The chances are that none of this area of the address space means anything to you, because it's used by Windows. Microsoft's documentation describing the Windows address space says that these are various areas reserved for MS-DOS compatibility, and that the area for user data and code begins at 4MB (see Related topics).

If you scroll down, you eventually come to something in the address space that you can clearly recognize, as shown in Figure 7.
The highlighted line and the one immediately below it in Figure 7 correspond to the Java heap. The Java process we started here was given a 1000MB heap (using \texttt{-mx1000m}) -- extravagantly large for the program in question but made this size so it would show up clearly in the PrcView map. The highlighted line shows the committed part of the heap as only 4MB, starting at address 0x10180000. Immediately after the highlighted line comes a large reserved stretch, which is the remainder of the heap that hasn't yet been committed. During startup, the JVM initially reserved the full 1000MB (making the address range 0x10180000 to 0x4e980000 unavailable) and then committed just what it needed to get started, in this case 4MB. To verify that this value really does correspond to the current heap size, you can invoke the Java program with the \texttt{-verbosegc} JVM option, which prints out detailed information from the garbage collector. From the second line of the second GC in the following \texttt{-verbosegc} output, you can see that the current heap size is approximately 4MB:

```
>java -mx1000m -verbosegc Hello
  [ JVMST080: verbosegc is enabled ]
  [ JVMST082: -verbose:gc output will be written to stderr ]
<GC[0]: Expanded System Heap by 65536 bytes
<GC[1]: GC cycle started Wed Sep 29 16:55:44 2004
<GC[1]: freed 417928 bytes, 72% free (3057160/4192768), in 104 ms>
<GC[1]: mark: 2 ms, sweep: 0 ms, compact: 102 ms>
<GC[1]: refs: soft 0 (age >= 32), weak 0, final 2, phantom 0>
<GC[1]: moved 8265 objects, 642680 bytes, reason=4>
```

The format of the \texttt{-verbosegc} output depends on the JVM implementation you use; see \texttt{Related topics} for an article on the IBM JVMs or consult your own provider's documentation.

In the event that the amount of live data grows and the JVM needs to expand the heap beyond 4MB, it commits a bit more of the reserved area. This means that the new area can start at 0x10580000, contiguous with the heap memory that's already committed.

The three totals in the bottom line of the PrcView screen in Figure 7 give you the total committed memory for your process, totalled based on the seventh column, headed Type. The totals are:

- **Private**: Committed memory backed by the paging file
• **Mapped**: Committed memory mapped directly into the file system
• **Image**: Committed memory belonging to executable code, both the starting executable and DLLs

So far, you've located the heap within the address space, based only on its size. In order to understand better what some of the other areas of memory are, it would be helpful to be able to peer inside the memory. This is what the next tool we'll discuss, TopToBottom, lets you do.

**TopToBottom**

TopToBottom is available free from smidgeonsoft.com (see Related topics). It comes without any documentation but provides a comprehensive set of views into the currently executing processes. You can sort the processes not only by name and process ID, but also by startup time, which can be very useful if you need to understand the sequence in which programs are started on your computer.

Figure 8 shows TopToBottom with the list of processes sorted by creation time (View > Sort > Creation Time).

**Figure 8. TopToBottom processes sorted by creation time**

The StartUp tab displays the process that created our Java process, the time and date at which it was started, and the actual command line used to invoke it, as well as the full path to the executable and the current directory. You can also click on the Environment tab to display the values of all the environment variables that were passed into the process at startup. The Modules tab shows the DLLs in use by our Java process, as in Figure 9.
Again, you can sort the list in a variety of ways. In Figure 9 they’re sorted by initialization order. If you double-click on a row, you'll see detailed information about the DLL: its address and size, the date and time it was written, a list of other DLLs on which it depends, and a list of all the running processes that have loaded the DLL. If you explore the list, you'll see that some DLLs, for example NTDLL.DLL, are required by every running process; some, such as JVM.DLL, are shared among all Java processes; and others might be used by only a single process.

You could try working out the total size of the DLLs the process uses by adding up the individual DLLs' sizes. However, the resulting number could be misleading because it doesn't mean that the process is consuming all that footprint. The true size depends on which parts of the DLL the process is actually using. Those parts contribute to the process's working set. It may seem obvious, but it's also worth noting that the DLLs are read-only and shared. If lots of processes all use a given DLL, only one set of real memory pages holds the DLL data at any one time. Those real pages might then be mapped at a number of different addresses into the processes that are using them. Tools such as Task Manager report the working set as the total of shared and nonshared pages, so it can be quite hard to determine the true effect of DLL use on footprint. The modules information is a useful way to get a "worst case" view of the footprint that's due to DLLs, which you can further refine by more detailed analysis using other tools if necessary.

You're interested in the memory footprint, so click on the Memory tab. Figure 10 shows a small subset of all the memory our Java program uses.
This display is similar to PrcView, but it shows only the committed memory in the virtual address space, not the reserved memory. However, it has a couple of advantages. First, it can characterize the pages in more detail. For example, in Figure 10 it has identified the Thread 3760 stack area specifically, not just as some read/write data. Additional data areas it recognizes include Environment, Process Parameters, Process Heap, Thread Stack, and Thread Environment Block (TEB). Second, you can browse or even search the memory directly from within TopToBottom itself. You can search for a text string, or you can do a hex search for a sequence of up to 16 bytes. You can restrict the hex search to a specified alignment, which is useful when you're searching for a reference to an address.

TopToBottom also has a snapshot facility that dumps all the information it has about the process to the clipboard.

**VADump**

VADump is a convenient command-line tool that's part of the Microsoft ® Platform SDK package (see Related topics). Its purpose is to dump an overview of the virtual address space and resident set for a particular process. The simplest way to use VADump is to enter this command at the command line:

```
vadump  process_id
```

The `process_id` is the number of the process you're interested in. You can invoke VADump with no arguments to display the full usage information. We also recommend that you pipe the output to a file (for example, `vadump 1234 > output.txt`), because VADump produces too much information to fit on the screen.

The output begins by showing an index into the virtual address space for the process:
Address: 00000000 Size: 00010000
  State Free

Address: 00010000 Size: 00002000
  State Committed
  Protect Read/Write
  Type Private

Address: 00012000 Size: 0000E000
  State Free

Address: 00020000 Size: 00001000
  State Committed
  Protect Read/Write
  Type Private

Address: 00021000 Size: 0000F000
  State Free

Address: 00030000 Size: 00010000
  State Committed
  Protect Read/Write
  Type Private

Address: 00040000 Size: 0003B000 RegionSize: 40000
  State Reserved
  Type Private

................................

(We've truncated the output at the dotted line for readability.)

For each block, you can see the following information:

- **Address**: In hexadecimal format, relative to the start of the process's virtual address space
- **Size**: In bytes, also in hexadecimal
- **State**: Free, reserved, or committed
- **Protection status**: Either read-only or read/write
- **Type**: Either private (not accessible by other processes), mapped (directly from the file system), or image (the executable code)

It then lists all of the DLLs in use by the process, with their sizes, followed by a summary of statistics about the working set and page-file usage.

So far, this information is also available from other tools. But you can generate much more revealing output by using VADump's `-o` option. It produces a snapshot of the current working set (the pages actually in main memory at a given point in time). This option is poorly documented, but it can be extremely useful for determining what the most significant components of the resident set are -- and therefore what the most promising candidates for memory optimization are. You can also use it to identify if you have a memory leak, by taking snapshots at regular intervals over a period of time. In this mode, the output begins with a more-detailed dump of the committed pages in the virtual address space, whether they are currently in main memory or not:

```
>vadump -o -p 3904

0x00010000 (0) PRIVATE Base 0x00010000
0x00011000 (0) PRIVATE Base 0x00011000
```
If you scroll down to the end of this long listing you'll come to the more interesting information: a list of the page-table mappings for the process's pages that are currently resident in main memory:

```
0xC0000000 > (0x00000000 : 0x003FFFFF)   132 Resident Pages
  (0x00280000 : 0x00286000) > jsig.dll
  (0x00290000 : 0x00297000) > xhpi.dll
  (0x002A0000 : 0x002AF000) > hpi.dll
  (0x003C0000 : 0x003D8000) > java.dll
  (0x003E0000 : 0x003F7000) > core.dll
  (0x00010000 : 0x00180000) > Process Heap segment 0
  (0x001A0000 : 0x001B0000) > Private Heap 0 segment 0
  (0x00390000 : 0x003A0000) > Unknown Heap 1 segment 0
  (0x003B0000 : 0x003C0000) > Process Heap segment 0
  (0x003D0000 : 0x003E0000) > Private Heap 2 segment 0
  (0x003F0000 : 0x00400000) > Private Heap 3 segment 0
  (0x00400000 : 0x00408000) > Stack for thread 0
0xC0001000 > (0x00400000 : 0x007FFFFF)   13 Resident Pages
  (0x00400000 : 0x00409000) > java.exe
```

Each of these mappings corresponds to a single entry in the page table, and so contributes an additional 4KB to the working set for the process. It can still be quite difficult to work out from these mappings what parts of your application are using the most memory, but luckily the next part of the output is a useful summary:
The two most interesting values are normally Heap, which is the Windows process heap, and Other Data. Memory allocated directly through Windows API calls form part of the process heap, and Other Data includes the Java heap. The Grand Total Working Set correlates to Task Manager's Mem Usage plus the Teb field, which is the memory needed for the processes's Thread Environment Block, an internal Windows structure.

Finally, the bottom of the `vADump -o` output is a summary of the relative contributions to the working set from DLLs, heaps, and thread stacks:

```
Module Working Set Contributions in pages
Total   Private Shareable Shared Module
9       2         7         0 java.exe
85      5         0         80 ntdll.dll
43      2         0         41 kernel32.dll
15      2         0         13 ADVAPI32.dll
11      2         0         9 RPCRT4.dll
53      6         0         47 MSVCRT.dll
253     31        222       0 jvm.dll
6       3         3         0 jsig.dll
7       4         3         0 xhpi.dll
15      12        3         0 hpi.dll
12      2         0         10 WINMM.dll
21      2         0         19 USER32.dll
14      2         0         12 GDI32.dll
6       2         0         4 LPK.DLL
10      3         0         7 USP10.dll
24      18        6         0 java.dll
22      16        6         0 core.dll
18      14        4         0 zip.dll
915     869       46        0 jitc.dll

Heap Working Set Contributions
  6 pages from Process Heap (class 0x00000000)
   0x00000000 - 0x000190000 6 pages
  2 pages from Private Heap 0 (class 0x00000100)
   0x00010000 - 0x001A0000 2 pages
  0 pages from UNKNOWN Heap 1 (class 0x00000000)
   0x00000000 - 0x000180000 0 pages
  1 pages from Process Heap (class 0x00000000)
   0x00380000 - 0x003900000 1 pages
  715 pages from Private Heap 2 (class 0x00000100)
   0x000030000 - 0x000400000 15 pages
   0x000080000 - 0x0009A0000 241 pages
   0x004A00000 - 0x004C60000 450 pages
   0x0054E0000 - 0x0058E0000 9 pages
  1 pages from Private Heap 3 (class 0x00000100)
```
You can also use VADump in this mode to get an accurate view of the combined footprint of two or more Java processes (see Tips and tricks, later in this article).

Sysinternals Process Explorer

Yet more useful tools for analyzing memory come from a company called Sysinternals (see Related topics). One is a graphical process explorer, shown in Figure 11, that you can use as an advanced replacement for Task Manager.

Figure 11. Process Explorer process tree

Process Explorer has all the same features as Task Manager. For example, you can get dynamic graphs of total system performance (with View > System Information...), and you can configure the columns in the main process view in a similar way. Under Process > Properties..., Process Explorer offers a lot more information about the process, such as full path and command line, threads, and dynamic graphs of CPU usage and private bytes. Its user interface is superior, as you can see in Figure 11. It can also inspect information on DLLs and handles for a process. You can use Options > Replace Task Manager to execute Process Explorer instead of Task Manager by default.

Don't forget about memory
Sysinternals ListDLLs

You can also download a couple of Sysinternals command-line utilities -- ListDLLs and Handle. They're particularly useful if you want to incorporate some form of memory monitoring into either scripts or programs.

ListDLLs lets you look at DLLs, which can make a significant contribution to memory footprint. To start using it, add it to your path and invoke it with the help option to get the usage information. You can invoke it with either the process ID or the name. Here is the list of DLLs our Java program uses:

```
>listdlls -r 3904
```

```
ListDLLs V2.23 - DLL lister for Win9x/NT
Copyright (C) 1997-2000 Mark Russinovich
http://www.sysinternals.com

---------------------------------------------------------------------
java.exe pid: 3904
Command line: java -mx1000m -verbosegc Hello

<table>
<thead>
<tr>
<th>Base</th>
<th>Size</th>
<th>Version</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00400000</td>
<td>0x9000</td>
<td>141.2003.0005.0022</td>
<td>C:\WINDOWS\system32\java.exe</td>
</tr>
<tr>
<td>0x77f50000</td>
<td>0x7800</td>
<td>5.01.2600.1217</td>
<td>C:\WINDOWS\System32\ntdll.dll</td>
</tr>
<tr>
<td>0x77e60000</td>
<td>0xe600</td>
<td>5.01.2600.1106</td>
<td>C:\WINDOWS\system32\kernel32.dll</td>
</tr>
<tr>
<td>0x77dd0000</td>
<td>0xad00</td>
<td>5.01.2600.1106</td>
<td>C:\WINDOWS\system32\ADVAPI32.dll</td>
</tr>
<tr>
<td>0x78000000</td>
<td>0x87000</td>
<td>5.01.2600.1361</td>
<td>C:\WINDOWS\system32\RPCRT4.dll</td>
</tr>
<tr>
<td>0x77c10000</td>
<td>0x53000</td>
<td>7.00.2600.1106</td>
<td>C:\WINDOWS\system32\MSVCRT.dll</td>
</tr>
<tr>
<td>0x10000000</td>
<td>0x17800</td>
<td>141.2004.0003.0001</td>
<td>C:\Java141\jre\bin\jvm.dll</td>
</tr>
<tr>
<td>### Relocated from base of 0x10000000:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00280000</td>
<td>0x6000</td>
<td>141.2004.0003.0001</td>
<td>C:\Java141\jre\bin\jsig.dll</td>
</tr>
<tr>
<td>### Relocated from base of 0x10000000:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00290000</td>
<td>0x7000</td>
<td>141.2004.0003.0001</td>
<td>C:\Java141\jre\bin\xhpi.dll</td>
</tr>
<tr>
<td>### Relocated from base of 0x10000000:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x002a0000</td>
<td>0xf000</td>
<td>141.2004.0003.0001</td>
<td>C:\Java141\jre\bin\hpi.dll</td>
</tr>
<tr>
<td>0x76b40000</td>
<td>0x2c00</td>
<td>5.01.2600.1106</td>
<td>C:\WINDOWS\system32\WINMM.dll</td>
</tr>
<tr>
<td>0x77d40000</td>
<td>0x8c000</td>
<td>5.01.2600.1255</td>
<td>C:\WINDOWS\system32\USER32.dll</td>
</tr>
<tr>
<td>0x7e000000</td>
<td>0x41000</td>
<td>5.01.2600.1346</td>
<td>C:\WINDOWS\system32\GDI32.dll</td>
</tr>
<tr>
<td>0x629c0000</td>
<td>0x8000</td>
<td>5.01.2600.1126</td>
<td>C:\WINDOWS\system32\LPK.DLL</td>
</tr>
<tr>
<td>0x72fa0000</td>
<td>0x5a000</td>
<td>1.400.2600.1106</td>
<td>C:\WINDOWS\system32\USP10.dll</td>
</tr>
<tr>
<td>### Relocated from base of 0x10000000:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x003c0000</td>
<td>0x18000</td>
<td>141.2004.0003.0001</td>
<td>C:\Java141\jre\bin\java.dll</td>
</tr>
<tr>
<td>### Relocated from base of 0x10000000:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x003d0000</td>
<td>0x17000</td>
<td>141.2004.0003.0001</td>
<td>C:\Java141\jre\bin\core.dll</td>
</tr>
<tr>
<td>### Relocated from base of 0x10000000:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x04a40000</td>
<td>0x12000</td>
<td>141.2004.0003.0001</td>
<td>C:\Java141\jre\bin\zip.dll</td>
</tr>
<tr>
<td>### Relocated from base of 0x10000000:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x04df0000</td>
<td>0x3a1000</td>
<td>141.2004.0003.0001</td>
<td>C:\Java141\jre\bin\jitc.dll</td>
</tr>
</tbody>
</table>
```

Alternatively, the `listdlls -r java` command shows all the running Java processes and the DLLs they're using.

Sysinternals Handle

Handle shows the list of handles (to files, sockets, and so on) that a process is using. Unzip the Handle download file and add it to your path to try it out. It produces this output if you try it on our Java program:

```
>handle -p 3904
```
You can see that our process has a handle to the directory on our classpath and to several JAR files. In fact the process has a lot more handles, but by default the utility only shows handles that refer to files. You can display others by using the \-a parameter:

```bash
> handle -a -p 3904
```

You can see that our process has a handle to the directory on our classpath and to several JAR files. In fact the process has a lot more handles, but by default the utility only shows handles that refer to files. You can display others by using the \-a parameter:
If you're interested in memory, handles are important because each one consumes some space. The exact amount depends on the operating-system version and the type of handle. In general, handles should not make a significant contribution to footprint. By simply counting the number of lines this utility outputs, you can quickly see if the number of handles is significant or -- worse still -- growing. Either scenario is a possible cause for concern and suggests some more-detailed investigation.

**Tips and tricks**

Now that you have a handle (no pun intended) on all the tools we've shown you, here are some ways you can use them individually or together to improve your memory monitoring.

**Finding a process ID**

To find the process ID of an application so you can use it in a command-line tool such as VADump, open the Applications tab in Task Manager and right-click on the process you're interested in. Select **Go To Process**. This takes you to the corresponding ID in the Processes tab.
Identifying a Java process

Have you ever puzzled over a list of processes all named java or javaw, trying to work out which is the one you want to investigate? If the Java process was launched from within an IDE or a script, it can be difficult to determine which JVM is in use and which command-line parameters were sent to the Java process. This information is readily available on the TopToBottom Startup tab. You'll see the full command line used to invoke the JVM and the time the process was started.

Identifying a handle hog

Ever tried to save a file only to be told that it's in use by another process? And even when you close the program you think is responsible, you still get the error message? You can use the SysInternals Process Explorer tool's Handle Search facility to find the culprit. Just open the Search dialog and type in the name of the file. ProcExp looks through all the open handles and identifies the process. Often it turns out to be a small stub process left running by an editor or Web browser after you've closed the user interface.

Investigating how much memory is being shared

You can use VADump with the -o option to get a detailed view of what is in a process's current working set and how much of it is shared. Take a dump of a single Java program running on the system, and then start up another one and take the dump again. If you compare the Code/StaticData values in each of the summaries, you'll see that the "Shareable" bytes have become "Shared," thereby reducing the incremental footprint by a small amount.

Trimming the resident set

Windows implements a policy of "trimming" a process's resident set when it does not appear to be in use. To demonstrate this, open up the Processes tab in Task Manager so that you can see the process of the application you're monitoring, then minimize the application window. Watch what happens to the Mem Usage field!

Determining the minimum amount of memory your application needs

For Windows Server 2003 and Windows NT, Microsoft provides an interesting utility called ClearMem that might be useful if you wish to explore further how applications use memory under Windows (see Related topics). This tool determines the size of real memory, allocates enough memory to consume it, touches the allocated memory as quickly as possible, and then releases it. This presses hard on the memory consumption of other applications, and the net effect of running ClearMem many times is to force the amount of memory an application is using down to its minimum value.

Conclusion

In this article, we've outlined how Windows manages memory and surveyed some of the most useful freely available tools that you can use to monitor your Java programs' memory usage.
You’ll no doubt find and use other tools, be they free downloads from the Web or commercial offerings, but we hope that we have armed you with enough knowledge to fight your way through the contradictory terminology. Often the only way to be confident that you know exactly what is being measured is to perform an experiment, such as the little C program we used to demonstrate the meaning of Task Manager’s VM Size and Mem Usage.

Of course, these tools can only help you to identify the problem. It's then up to you to solve it. Most of the time you'll find that the Java heap is gobbling up the biggest slice of the memory, and you'll need to delve into your code’s details to make sure that object references aren't being held longer than necessary. Many more tools and articles can help you with this effort, and some useful links in the Related topics section should point you in the right direction.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>A C program to demonstrate how Windows uses memory</td>
<td>experiment.c</td>
<td>3KB</td>
</tr>
</tbody>
</table>
Related topics

- Visit the Memory Management Reference for an extensive glossary, FAQ, and articles on memory management.
- Download the latest version of PrcView and learn more about its features.
- You can download TopToBottom and other Windows tools from www.smidgeonsoft.com.
- Process Explore, ListDLLs and Handle are freely available at the SysInternals Web site.
- Learn about ClearMem from the Microsoft Windows Server 2003 documentation site.
- The article "Handling Memory Leaks in Java programs" (developerWorks, February 2001) provides more information on heap-related memory issues.
- The article "Whose object is it, anyway?" (developerWorks, June 2003) discusses the need to track object ownership to avoid memory leaks.