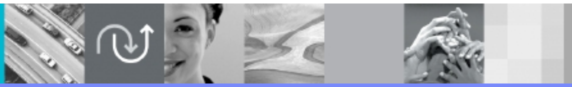
IBM Software Group

Windows® Native Memory Problem Determination Techniques and Tools for WebSphere Application Server

Kevin Grigorenko (kevin.grigorenko@us.ibm.com)
IBM® WAS SWAT Team
22 June 2011



WebSphere® Support Technical Exchange

ON DEMAND BUSINESS

Agenda

- **Overview**
- **Windows Native Memory Layout**
- **Detection**
- **Monitoring**
- **Isolation & Avoidance**
- **Analysis**
  - UMDH
  - LeakDiag, DebugDiag
  - VMMap

Overview

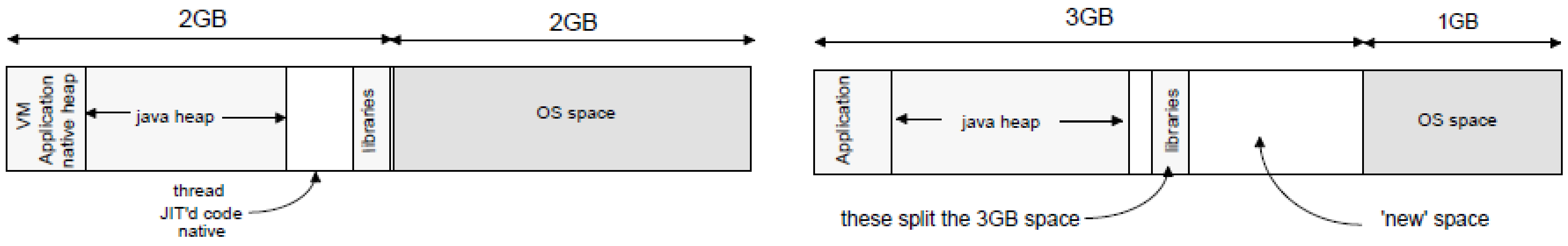
- **Native memory issues are notoriously difficult**
  - May cause: crashes, thrashing, OS instability, high CPU
  - Isolation and/or avoidance is often easier than analysis
  - Operating systems provide analysis tools - difficult to use
- **Common on Windows 32-bit JVMs because the user address space (by default) is limited to 2GB**
- **This presentation covers how to detect, monitor, avoid, isolate, and analyze, in that order.**
- **This presentation focuses on the IBM JVM.**

Native Memory Basics

- **Native memory generally means the virtual native memory of a process or address space. This is limited by the hardware architecture and operating system (OS).**
  - Native memory may be resident, used, unused, reserved, committed, swapped out, paged out, etc. All of these don't matter for this presentation. That's how the OS deals with virtual memory, based on configuration and constraints.
  - Insufficient RAM for the peak, active virtual memory needs causes paging which dramatically impacts performance.
- **32-bit: Max theoretical native memory per process=4GB. 64-bit: 16 million TB (practically less, but essentially no native OOMs)**
  - A 32-bit process running in a 64-bit OS still has a 32-bit virtual address
- **A Java^{TM} process has a native heap and a Java heap. These are both carved out of the native memory.**

Windows Native Memory Layout

- **By default, Windows 32-bit uses a 2GB virtual address space**
  - Rest used by kernel, shared across processes (paged pool, page table, PTEs, drivers, etc.)
  - http://download.boulder.ibm.com/ibmdl/pub/software/dw/jdk/diagnosis/dw3gbswitch3.pdf [MSFT1, 2]



- **In 3GB mode, some libraries are still based at the 2GB boundary, so -Xmx is practically limited to between -Xmx1200m and -Xmx1856m**
  - Library rebasing is possible but then shared libraries loaded privately
- **Java 6 split heap option may be used (forces gencon):**
  - -Xgc:splitheap -Xmx2800m -Xmox1800m

Using 3GB Mode

- **Java is compiled with LARGEADDRESSAWARE**
- **Not risk free!**
  - Third party JNI libraries with pointer arithmetic may have unexpected issues or crashes
  - The kernel itself can run into issues, particularly with exhausted page translation table entries
- **Windows <= 2003: /3GB boot.ini switch, reboot box**
  - http://technet.microsoft.com/en-us/library/bb124810.aspx
- **Windows > 2003: BCDEdit /set increaseuserva 3072, reboot**
  - http://msdn.microsoft.com/en-us/library/ff542202.aspx [Also]
- **Ensure enough physical memory:**
  - http://msdn.microsoft.com/en-us/library/aa366778%28v=vs.85%29.aspx

Detection

- **Best detection is monitoring, covered later; however, some signs of a native OutOfMemoryError (NOOM):**
  - An OutOfMemoryError is generated with details about not being able to launch threads (below example from Javacore, may show in SystemOut.log)
    - 1TISIGINFO   Dump Event "systhrow" (00040000) Detail "java/lang/OutOfMemoryError" "Failed to create a thread: retVal -1" received
  - An OutOfMemoryError is generated but there is sufficient Java heap space (consult verbosegc or the “Bytes of Heap Space Free” section in the Javacore).
    - Not 100% since this can also be Java heap fragmentation, the heap is not fully expanded, or there was a massive Java allocation (always check requested alloc sizes before OOM).

Detection (Continued)

- **An OutOfMemoryError is thrown and the "Current Thread" is in a native method, e.g.:**

```
3XMTHREADINFO3         Java callstack:
4XESTACKTRACE            at java/lang/Thread.startImpl(Native Method)
4XESTACKTRACE            at java/lang/Thread.start(Thread.java:887(Compiled Code))
```

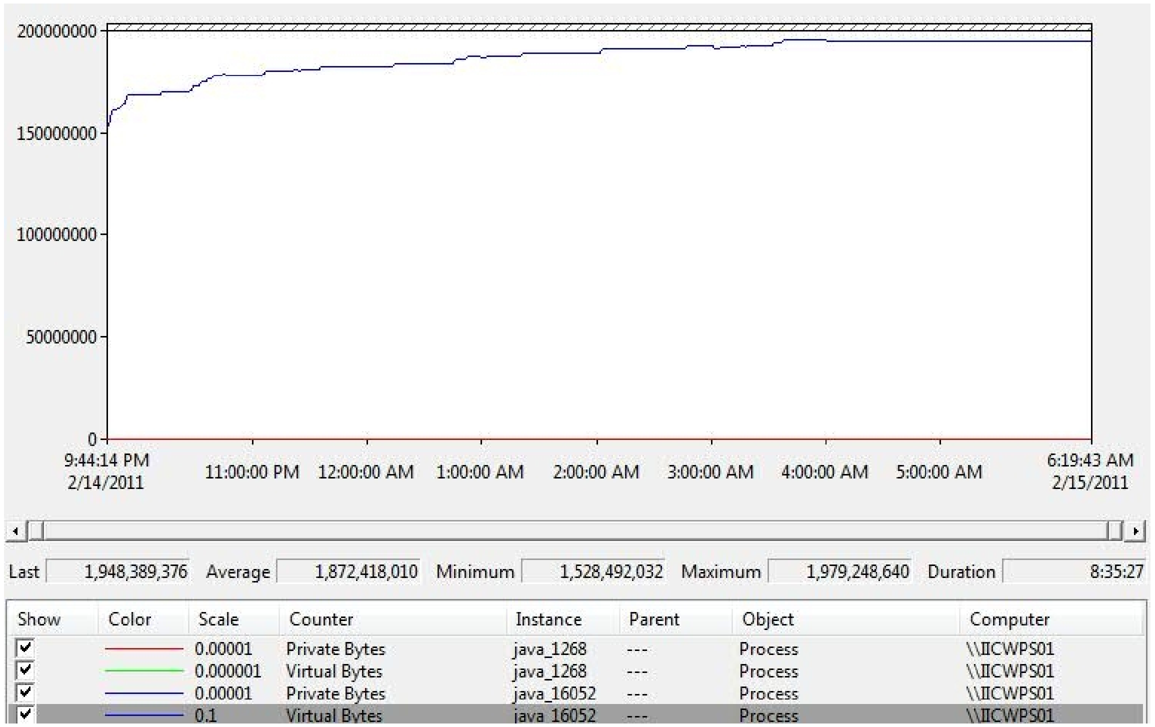
- **The JVM crashes and its virtual memory usage is near its limit (windbg → !address -summary)**
- **With verbosegc enabled, the Javacore has a GC flight recorder section, which may show:**
  - J9AllocateIndexableObject() returning NULL!

Monitoring

- **Use Windows Perfmon. By default, counters do not show the PID, so use the PID format:**
  - http://support.microsoft.com/kb/281884
  - No restart of machine/Java required, just restart perfmon
- **For each process, use the Virtual Bytes and Private Bytes counters (Performance Object → Process)**
  - http://publib.boulder.ibm.com/infocenter/javasdk/tools/topic/com.ibm.java.doc.igaa/_1vg000121410cbe-1195c23a635-8000_1008.html
- **Also gather system-wide counters**
  - CPU, Memory, Disk, etc. (recent Windows versions' perfmon have this predefined under System Performance → Performance Counter)

Monitoring

- **Load CSV or BLG from perfmon (View Log Data → Log Files)**
- **Right click counter to change scale**
  - Example on right shows leak in virtual
  - Change Y-axis
- **Or write to CSV and load in spreadsheet**

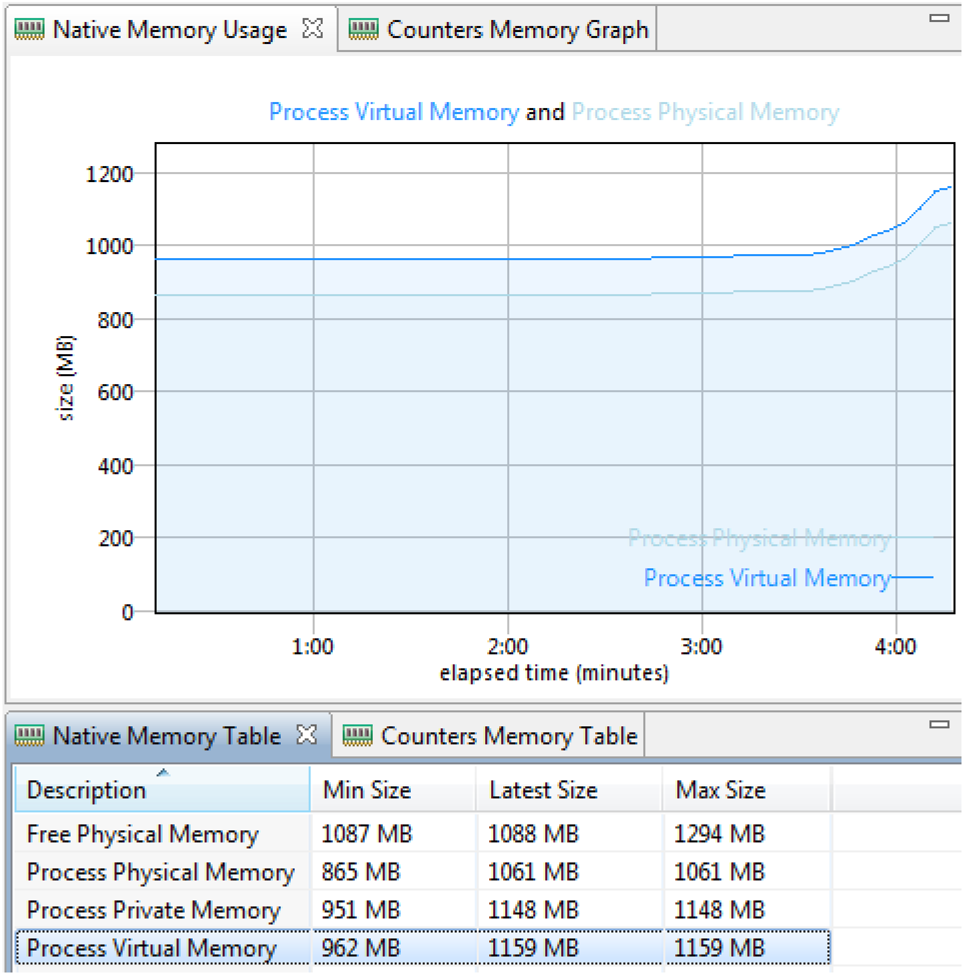


Last 1,948,389,376 Average 1,872,418,010 Minimum 1,528,492,032 Maximum 1,979,248,640 Duration 8:35:27

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
✓	🟥	0.00001	Private Bytes	java_1268	---	Process	\\IICWPS01
✓	🟩	0.000001	Virtual Bytes	java_1268	---	Process	\\IICWPS01
✓	🟦	0.00001	Private Bytes	java_16052	---	Process	\\IICWPS01
✓	🟦	0.1	Virtual Bytes	java_16052	---	Process	\\IICWPS01

Monitoring

- **Health Center has native memory monitoring**
  - Ships with the JVM, but always good to upgrade the agent
  - Generic JVM argument -Xhealthcenter
    - -Xhealthcenter:level=low to not gather profiling data
  - Visualization client in IBM Support Assistant
  - -Xhealthcenter:level=headless for writing to an HCD file



Description	Min Size	Latest Size	Max Size
Free Physical Memory	1087 MB	1088 MB	1294 MB
Process Physical Memory	865 MB	1061 MB	1061 MB
Process Private Memory	951 MB	1148 MB	1148 MB
Process Virtual Memory	962 MB	1159 MB	1159 MB

Avoidance/Isolation Techniques

- **Many of these techniques might not resolve the problem or are workarounds, and have "costs"**
- **Reduce -Xmx**
- **Reduce number of threads (or stack size [-Xss])**
- **Reduce number of classes/classloaders**
- **Fixed size thread pools (min=max)**
  - http://www-01.ibm.com/support/docview.wss?uid=swg21368248
  - Major thread pools (WebContainer, etc.), e.g. not startup
- **Ensure latest versions of native libraries** (e.g. type 2 DB drivers)
- **Reduce per-JVM max throughput, scale out JVMs**

Avoidance/Isolation Techniques

- **Obligatory, but important – use latest WAS/Java FP**
- **Ensure -Xnoclassgc is not set**
- **If a lot of sun/reflect/DelegatingClassLoader (e.g. a lot of reflection), use -Dsun.reflect.inflationThreshold=0**
- **Use com.ibm.ws.webcontainer.channelwritetype=sync**
  - http://www-01.ibm.com/support/docview.wss?uid=swg21317658
- **Disable AIO:** http://www-01.ibm.com/support/docview.wss?uid=swg21317658
- **Switch to 64-bit JVMs**
- **Links**
  - http://www-01.ibm.com/support/docview.wss?uid=swg21373312

Data in System Dumps

- **Some native memory info in system dumps. Use:**
  - -Xdump:heap:none -Xdump:java+system:events=systhrow,filter=java/lang/OutOfMemoryError,range=1..4,request=exclusive+compact+prepwalk
  - http://www.ibm.com/developerworks/opensource/library/j-memoryanalyzer/index.html
- **Install IBM Product Extensions into MAT:**
  - http://www.alphaworks.ibm.com/tech/iema
  - http://dl.alphaworks.ibm.com/ettktechnologies/updates
- **Open Query Browser → IBM Extensions → Java SE Runtime → DirectByteBuffers**

Other Analysis

- **Javacores have a wealth of native memory information related to the JVM itself (MEMINFO)**
  - 1STSEGTYPE is one of
    - Internal Memory: general segment usage
    - Object Memory: Java heap, should match verbosegc heap use
    - Class Memory: Native memory for classes
    - JIT Code Cache: JIT compiled code
    - JIT Data Cache: JIT data
- **Useful to check JIT code and/or data leaks**
- **Also has information on duplicate classes**
- **Aggregation scripts:**
  - get_memory_use.pl

Other Analysis

- **To diagnose JVM memory leaks, use the command line option:**
  - -memorycheck:callsite=1000
    - http://publib.boulder.ibm.com/infocenter/javasdk/tools/topic/com.ibm.java.doc.igaa/_1vg000121410cbe-1195c23a635-7ffd_1001.html
  - Also available on a core dump:
    - jextract -interactive core.2011...0001.dmp (Commands must be prefixed with '!') > !findallcallsites
- **If there is a leak after restarting applications, this may be a classloader leak. The IBM Extensions for Memory Analyzer has a query for this:**
  - http://www.ibm.com/developerworks/websphere/techjournal/1103_supauth/1103_supauth.html#sec10

Analysis

- **Four primary tools covered (all from Microsoft®)**
  - DebugDiag: http://www.microsoft.com/downloads/en/details.aspx?FamilyID=28bd5941-c458-46f1-b24d-f60151d875a3&displaylang=en
  - UMDH:
    - http://publib.boulder.ibm.com/infocenter/javasdk/tools/topic/com.ibm.java.doc.igaa/_1vg000121410cbe-1195c23a635-7ffe_1005.html
    - http://www-01.ibm.com/support/docview.wss?uid=swg21313578#trackingWithUMDH
  - LeakDiag: ftp://ftp.microsoft.com/PSS/Tools/Developer Support Tools/LeakDiag
  - VMMap: http://technet.microsoft.com/en-us/sysinternals/dd535533
- **Install Windows Debugging Tools:**
  - 32-bit: http://msdn.microsoft.com/en-us/windows/hardware/gg463016
  - 64-bit: http://msdn.microsoft.com/en-us/windows/hardware/gg463012
    - If installing the SDK, you can just install the tools

Frame Pointer Omission (FPO)

- **UMDH is different from the other three tools. Whereas the other tools “inject” themselves into the executing process (e.g. using the Detours API), UMDH consults an allocation “database” created by the kernel itself (because gflags was set).**
  - If gflags is set, this database can be seen in windbg also.
- **The kernel does not have access to symbols, but most 32-bit programs use a register convention in which the EBP register points to the previous method.**
  - A different convention is used for 64-bit so the problems discussed on the next slide do not affect 64-bit apps.

Frame Pointer Omission (FPO)

- **There is an optimization called Frame Pointer Omission (FPO) which makes the kernel stack walker not work. A big problem occurred when MS Visual C++ 2005 used this optimization.**
- **Visual C++ 2005 SP1 and future versions no longer use this optimization (/Oy-), so given that Windows itself is built without FPO, then applications also should not use it. The Java libraries do not appear to use it.**
- **Check if FPO is used by an EXE or DLL:**
  - dumpbin.exe /fpo %MODULE%
  - If there are "FPO Data" lines, then it is used.

Symbols

- **In general, Windows executables and libraries (DLLs) are “stripped” – i.e. no symbols**
- **Symbol files are .PDB files with the same name as the EXE or DLL. Two types: public or private**
  - http://msdn.microsoft.com/en-us/library/ff550665.aspx
- **Java ships with PDBs. WAS does not (only a few libraries - AIO, WsProcessManagement, few others)**
- **Install the Windows symbols (retail):**
  - http://msdn.microsoft.com/en-us/windows/hardware/gg463028
- **Try to get PDBs for third party libraries**

Symbol Path

- **Some tooling requires a symbol path to make sense of the data. Takes the form:**
  - c:\path1;c:\path2;c:\winsympath;srv*c:\winsympath*http://msdl.microsoft.com/download/symbols
  - Just a list of paths that have PDBs, except the “srv” one which is actually a * delimited list to define the Microsoft symbol server, from which symbols can be dynamically downloaded. c:\winsympath in the example should be where you installed the Windows symbols
  - http://support.microsoft.com/kb/311503

Symbol Path

- **Assuming WAS in C:\WAS and Windows symbols in C:\Windows\Symbols, example sympath:**
  - c:\Windows\Symbols\;C:\WAS\java\jre\bin\;C:\WAS\java\jre\bin\j9vm\;srv*c:\Windows\Symbols\*http://msdl.microsoft.com/download/symbols
- **To avoid issues, set this as a System Environment Variable with the name _NT_SYMBOL_PATH**
- **In windbg, when loading a customer dump, you also need their PDBs (or from JIM), then set with:**
  - .sympath c:\Windows\Symbols\;C:\customer\java\jre\bin\;C:\customer\java\jre\bin\j9vm\;srv*c:\Windows\Symbols\*http://msdl.microsoft.com/download/symbols
    .reload

Symbols

- **In general, you want “retail” symbols and not “checked” symbols. The latter are for debug builds of Windows.**
- **Some production system cannot use a symbol server. In this case, run symchk /om on the box to get a list of symbols that it needs. Then run symchk /im with this list to download those symbols and then transfer them over.**

DebugDiag

- **Start DebugDiag; Click on Cancel on the Select Rule Type Dialog Box**
- **Tools → Options and Settings**
  - Set symbol path in “Symbol Search Path for Analysis”
  - Preferences → Check "Record call stacks immediately when monitoring for leaks"
    - If customer experiences large overhead, uncheck this
- **Attach to the target WAS process:**
  - Process tab → Right Click → Monitor for Leaks
- **Take snapshots over time → Right Click → Create Full Userdump**

DebugDiag

- **Alternatively, create a rule through the wizard. This also allows setting up when dumps are automatically taken (e.g. after virtual and/or private bytes increases past some # of MB and at MB intervals after that)**
  - Also allows for catching a crash when virtual bytes limit hit
  - http://msdn.microsoft.com/en-us/library/ff420662.aspx

DebugDiag

- **To analyze the data**
  - Set your symbol path
  - Load the user dumps in the Advanced Analysis tab
  - Click Memory Pressure Analyzers
  - Click Start Analysis
- **Analysis creates an HTML file (IE .mht file)**
  - If you're running Linux®, you can open .mht in Opera
- **It's much easier if the customer runs the analysis and then just sends the .mht file. This avoids having to get the same PDBs and transferring the dumps**

DebugDiag

- **How it works**
  - Most memory allocations belong in one of three groups: caching, short term allocations that will be released later, and memory leaks. All three allocation methods have very distinct allocation patterns when measured over time. The leak-tracking feature calculates a leak probability using a formula that is based on these allocation patterns as measured over a specific time period. More precisely, leak probability is a number between 0 and 100 that measures how allocations are spread over time. Empirical studies show that leak allocations tend to be evenly spread over time. When allocations are evenly spread over time, leak probability equals 100. If allocations are bunched either at the beginning or at the end of the tracking duration, this usually indicates either caching allocations or short term allocations, respectively. If all allocations occurred at the beginning or at the end of a process, then leak probability will equal zero. Additional studies show that a high allocation count accompanied by leak probability higher than 75 percent indicates memory leaks. A properly functioning process could be experiencing heavy caching or short-term allocations and these phenomena could mask other behaviors, so it is important to use this synthetic time distribution calculation to get stack samples for those functions.

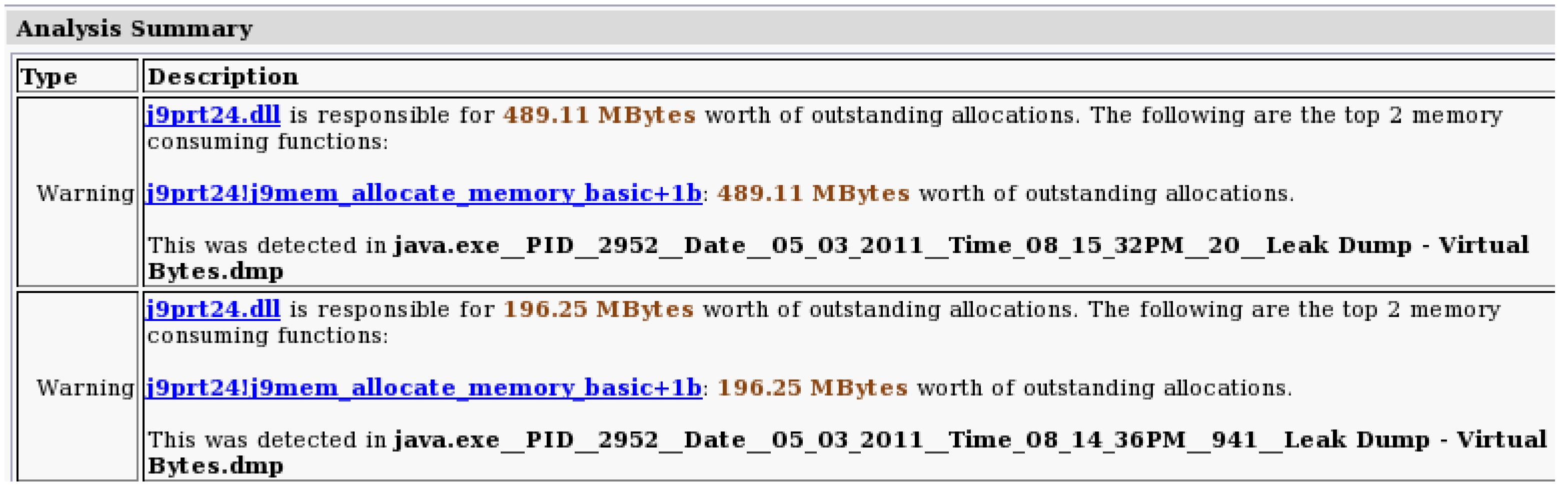
DebugDiag

- … How it works …
  - As they accumulate, these allocations are sorted based on the following filters:
    Top 10 functions sorted by allocation count.
    Top 10 functions sorted by total allocation size.
    Top 10 functions sorted by leak probability.

    The final sorted combination determines which functions must have stack samples associated with them.
  - A stack sample is a heuristic record based on the X86 op codes of possible return addresses that are found on the stack at execution time. In almost all cases, these samples will contain spurious addresses. MemoryExt.dll, the analysis module extension, uses the symbol information found in these samples to reconstruct a stack that will resemble the stack at the time of failure as closely as possible. This method is used because debug symbols are not used at run time, and reading and accessing the stack at every allocation would cause significant performance overhead.
  - Q: I am debugging a high CPU issue, and I can't use a hang rule because the process is not an IIS process, and I can't take a manual dump because the problem is random. What can I do?
    A: Modify the service script that is included with the tool to set a trigger on process CPU usage. Service scripts (dbgsvc.vbs) are located in the \Samples folder.

DebugDiag

- **Analysis first summarizes curious things**



Analysis Summary

Type	Description
Warning	j9prt24.dll is responsible for 489.11 MBytes worth of outstanding allocations. The following are the top 2 memory consuming functions: j9prt24!j9mem_allocate_memory_basic+1b: 489.11 MBytes worth of outstanding allocations. This was detected in **java.exe__PID__2952__Date__05_03_2011__Time_08_15_32PM__20__Leak Dump - Virtual Bytes.dmp**
Warning	j9prt24.dll is responsible for 196.25 MBytes worth of outstanding allocations. The following are the top 2 memory consuming functions: j9prt24!j9mem_allocate_memory_basic+1b: 196.25 MBytes worth of outstanding allocations. This was detected in **java.exe__PID__2952__Date__05_03_2011__Time_08_14_36PM__941__Leak Dump - Virtual Bytes.dmp**

- **Note that each core is treated independently, so best to look at most recent (time is in the file name)**

DebugDiag

- **Search for "Call stack sample," e.g.:**

```
Call stack sample 4
Address   0x25f60020
Allocation Time   00:01:55 since tracking started
Allocation Size   97.66 MBytes

Function   Source   Destination
j9prt24!j9mem_allocate_memory_basic+1b   c:\cygwin\home\foreman\sandbox\jvm-src\src\j9\port\win32\j9mem.c @ 32   ntdll!RtlAllocateHeap
j9prt24!j9mem_allocate_memory+4a   c:\cygwin\home\foreman\sandbox\jvm-src\src\j9\port\common\j9memtag.c @ 160   j9prt24!j9mem_allocate_memory_basic
jclscar_24!sun_misc_Unsafe_allocateMemory+8d   C:\Cygwin\home\Foreman\sandbox\jvm-src\src\j9\jcl\inl\smunsafe.asm @ 644
oleaut32!DllMain+2c
j9prt24!j9sig_protect+41   c:\cygwin\home\foreman\sandbox\jvm-src\src\j9\port\win32\j9signal.c @ 144  …
```

- **No Java methods, but it's a start. In this case, someone is calling sun/misc/Unsafe.allocateMemory() for 97MB**

DebugDiag

- **In this case, I know that DirectByteBuffers are one caller of Unsafe.allocateMemory. Using IEMA:**
- Alignment size of 4096 bytes, and word size of 4 bytes.
  105 total instances of java.nio.DirectByteBuffer
  103 total non-viewed* DirectByteBuffers. Sum capacity (with overhead)=517472736 (493.50 MB). Sum capacity (without overhead)=517050848 (493.09 MB). Overhead=0.08%, 421888 (412.0 KB)
  103 total non-viewed*, non-phantomed** DirectByteBuffers. Sum capacity (with overhead)=517472736 (493.50 MB). Sum capacity (without overhead)=517050848 (493.09 MB). Overhead=0.08%, 421888 (412.0 KB)
  Maximum non-viewed*, non-phantomed** DirectByteBuffer = 102406496 (97.66 MB)
  => Sum DirectByteBuffer capacity available for GC: 0 (0.0 B)
  => Sum DirectByteBuffer capacity not available for GC: 517472736 (493.50 MB)

- Histogram of Incoming References (*, **)

  6 instances incoming from java.lang.Object[10]=512037600 (488.31 MB)...

```
Class Name                                              | Shallow Heap | Retained Heap |    Capacity |IsViewed| # Inbound
----------------------------------------------------------------------------------------------------------------------------
java.nio.DirectByteBuffer @ 0x2b94dc8                   |           72 |            72 | 102,406,496 |false   |         2
|- java.lang.Object[10] @ 0xbffd98                      |           56 |            56 |           0 |false   |        -1
|  '- java.util.ArrayList @ 0xbffd78                    |           32 |            88 |           0 |false   |        -1
|     '- class com.ibm.AllocateNativeMemory @ 0x218cc20|          250 |           690 |           0 |false   |        -1
java.nio.DirectByteBuffer @ 0x2b83ba8                   |           72 |            72 | 102,406,496 |false   |         2
java.nio.DirectByteBuffer @ 0x2b83920                   |           72 |            72 | 102,406,496 |false   |         2
java.nio.DirectByteBuffer @ 0x23b6800                   |           72 |            72 | 102,406,496 |false   |         2
java.nio.DirectByteBuffer @ 0x2394e20                   |           72 |            72 | 102,406,496 |false   |         2
----------------------------------------------------------------------------------------------------------------------------
```

UMDH

- **Basic process to get UMDH data:**
  - Install debugging tools; Set symbol path
  - Run gflags on WAS Java executable (sets registry flag):
    - gflags -i <WAS>\java\bin\java +ust
    - gflags -i <WAS>\java\jre\bin\java +ust
  - Restart WAS; Start Perfmon
  - Reproduce problem
    - Take native mem snapshots over time: umdh -p:<pid> -f:umdhN.txt
  - Compare snapshots (either first to last, or in pairs):
    - umdh -v umdhX.txt umdhY.txt -f:umdhdiffZ.txt
  - Undo gflags – same as above, except -ust

UMDH

- **Analyzing the data**
  - See previous links
  - UMDH attempts to figure out which allocations have been not freed, then aggregates by stack trace
  - The diff file is a text file that sorts by largest difference in total allocation size by back trace; e.g. leak of 0x55500=341KB:

```
+   55500 ( 77700 - 22200)    16c allocs              BackTrace5E929EF0
+     104 (   16c -    68)        BackTrace5E929EF0              allocations
ntdll!RtlAllocateHeap+0000021D
MSVCR90!malloc+00000079
(f:\dd\vctools\crt_bld\self_x86\crt\src\malloc.c, 163)
MSVCR90!operator new+0000001F
(f:\dd\vctools\crt_bld\self_x86\crt\src\new.cpp, 59)...
```

LeakDiag

- **Older tool, not much documentation**
- **Tools → Options**
  - Set Symbol search path
  - Check “Use DbgHelp StackWalk API to walk stacks”
  - Change “Max stack depth” to 10 or more
- **Select process (gflags not needed)**
- **Select the allocator (usually Windows Heap or NT Allocator is what's needed)**
- **Click Start button**
- **Take N snapshots over time with Log button**

LeakDiag

- **Creates XML files, no diff function built in**
- **XML files group by stack trace. Example:**

```
<STACK numallocs="042" size="024" totalsize="01008">
<STACKSTATS>
<SIZESTAT size="024" numallocs="042"/>
<HEAPSTAT handle="572c0000" numallocs="042"/>
</STACKSTATS>
<FRAME num="0" dll="MSVCR80.dll" function ="malloc" offset="0x1F5" filename="" line=""
addr="0x73F44EFE" />
<FRAME num="1" dll="MSVCR80.dll" function ="calloc" offset="0x18" filename="" line="" addr="0x73F44F70" />
<FRAME num="2" dll="gxiosa.dll" function ="osaMutexCreateGlobalMutex" offset="0x8F" filename="" line=""
addr="0x563E73BF" />
…
<STACKID>5EFA57E8</STACKID>
</STACK>
```

VMMap

- **When starting, click cancel**
- **Options → Configure Symbols**
- **File → Select Process → Launch and Trace a new Process**
  - Unfortunately, this requires the full Java args to WAS
  - You can use startServer.bat -script to generate this
- **Click Trace button**
- **You can also select heap block and click the Heap Allocations and Calltree buttons to see details**
- **Also can do differences between snapshots**

Other Information

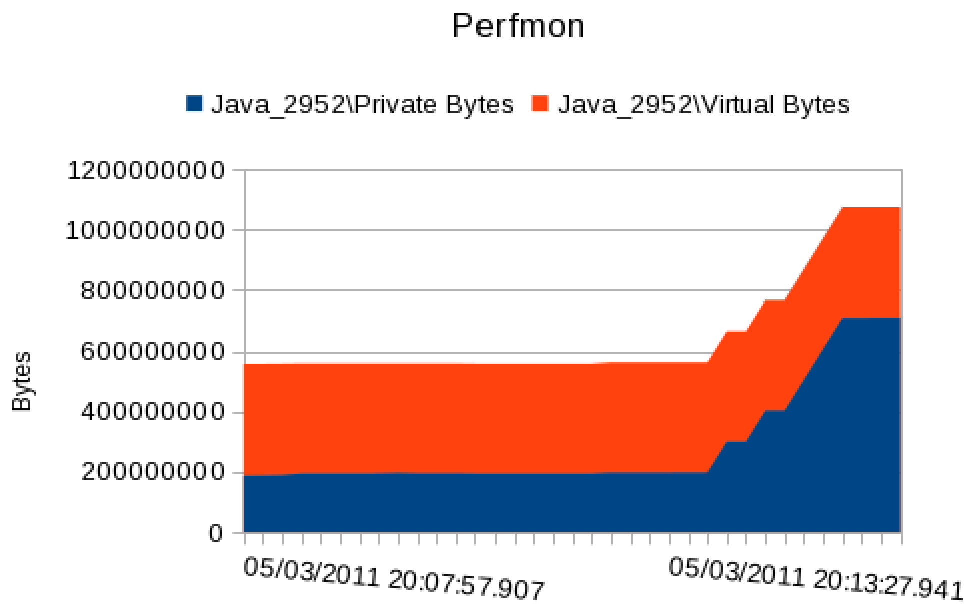
- You can get some really weird stack traces with wrong or incomplete symbols
- Some tools show truncated stacks with all precautions taken (FPO, symbols) – when you hit this and tried everything you can, open a support ticket with Microsoft
- Sometimes dumps can fail to be created:
  - http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/topic/com.ibm.java.doc.diagnostics.60/diag/tools/nodump_winmemory.html
- Taking manual dump using userdump.exe:
  - http://support.microsoft.com/kb/241215
- Some allocations never freed (cache) - may be ok

Windbg Useful Commands

- **Open Crash Dump**
  - List all loaded libraries → lmf
  - Write output to file → .logopen %SOMEFILE%
  - Virtual memory info → !address -summary
  - List all native heaps → !heap -s
  - List details of a particular heap (Heap ID is first column in !heap -s) → !heap -stat -h <Heap ID>
  - Given a UserPtr and EXE has gflags +ust, dump stack → !heap -p -a <UserPtr>
  - Was gflags set? → !gflag
  - Dump arbitrary address → db 0x123...

windbg

- **Perfmon data from a previous example to correlate with windbg**



windbg

- **!address -summary to see dumps' virtual memory**
- **Example:**
  - Dump#1

```
--- State Summary ---------------- RgnCount ----------- Total Size -------- %ofBusy %ofTotal
MEM_FREE                                120          9231e000 (   2.284 Gb)           76.14%
MEM_COMMIT                              993          1d2f8000 ( 466.969 Mb)  63.72%   15.20%
MEM_RESERVE                             269          109da000 ( 265.852 Mb)  36.28%    8.65%
```

  - Dump#2

```
MEM_FREE                                123          7fe20000 (   1.998 Gb)           66.61%
MEM_COMMIT                              992          2f7c1000 ( 759.754 Mb)  74.06%   24.73%
MEM_RESERVE                             270          10a0f000 ( 266.059 Mb)  25.94%    8.66%
```

- **Add MEM_COMMIT+MEM_RESERVE. This lines up with perfmon (~700MB, then 1.1GB)**
- **Also look at MEM_FREE. In this case, I'm running with /3GB (e.g. in the last example, 1.9+1.1=3)**

Notes & Links

- **Notes**
  - Summing virtual bytes over processes using the same shared libraries (e.g. multiple WAS JVMs on a node) will have some double counting.
    - Similarly, summing private bytes will be under-counting.
- **Links**
  - Native OOM MustGather for Windows:
    - http://www-01.ibm.com/support/docview.wss?uid=swg21313578
  - https://www-950.ibm.com/events/wwe/impact/impact10cms.nsf/download/k8a14cc9d282c282c128831c1151/$FILE/IMPACT_native_memory.pdf
  - http://www-01.ibm.com/support/docview.wss?uid=swg27013819&aid=1

Summary

- In summary, native OutOfMemoryErrors (NOOMs) are one of the most difficult classes of problems.
- 32-bit Windows programs are particularly prone due to the default 2GB user virtual address space limit.
- Monitoring native memory is absolutely essential.
- Various workarounds exist for NOOMs but have various costs.
- Various tools exist to investigate root cause, with DebugDiag and UMDH being the best.

Additional WebSphere Product Resources

- Learn about upcoming WebSphere Support Technical Exchange webcasts, and access previously recorded presentations at: http://www.ibm.com/software/websphere/support/supp_tech.html
- Discover the latest trends in WebSphere Technology and implementation, participate in technically-focused briefings, webcasts and podcasts at: http://www.ibm.com/developerworks/websphere/community/
- Join the Global WebSphere Community: http://www.websphereusergroup.org
- Access key product show-me demos and tutorials by visiting IBM Education Assistant: http://www.ibm.com/software/info/education/assistant
- View a webcast replay with step-by-step instructions for using the Service Request (SR) tool for submitting problems electronically: http://www.ibm.com/software/websphere/support/d2w.html
- Sign up to receive weekly technical My Notifications emails: http://www.ibm.com/software/support/einfo.html

Connect with us!

1. **Get notified on upcoming webcasts**
   Send an e-mail to wsehelp@us.ibm.com with subject line “wste subscribe” to get a list of mailing lists and to subscribe
2. **Tell us what you want to learn**
   Send us suggestions for future topics or improvements about our webcasts to wsehelp@us.ibm.com
3. **Be connected!**
   Connect with us on Facebook
   Connect with us on Twitter

Questions and Answers