IBM Software Group

AIX Native Memory Problem Determination Techniques and Tools for WebSphere Application Server

Kevin Grigorenko (kevin.grigorenko@us.ibm.com)
IBM® WAS SWAT Team
October 16, 2012

WebSphere® Support Technical Exchange

Agenda

- Overview
- AIX Native Memory Layout
- Detection
- Monitoring
- Isolation & Avoidance
- Analysis
  - MALLOCDEBUG

Overview

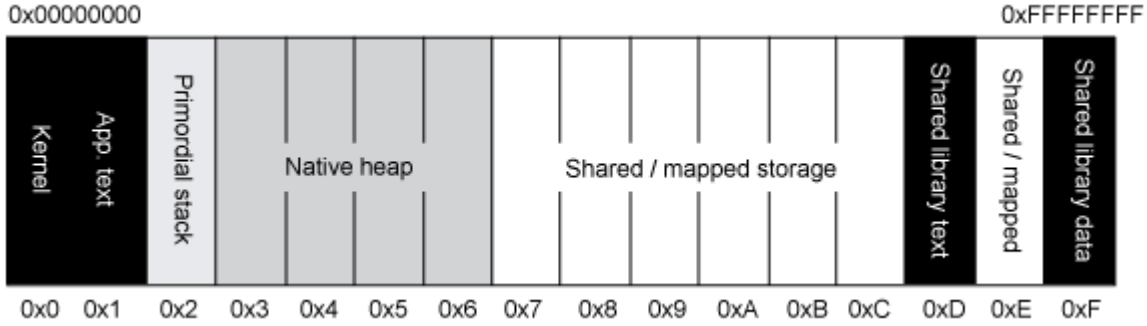
- Native memory issues are notoriously difficult
  - May cause: crashes, thrashing, OS instability, high CPU
  - Isolation and/or avoidance is often easier than analysis
  - Operating systems provide analysis tools - difficult to use
- This presentation covers how to detect, monitor, avoid, isolate, and analyze, in that order.

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).
  - Resident native memory must, by definition, be backed by physical memory (RAM). Insufficient RAM for the peak, active real 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
  - A 32-bit process running in a 64-bit OS still has a 32-bit virtual address space.
- A Java process has a native heap and a Java heap. These are both carved out of the native memory.

AIX Native Memory Layout (32-bit)

- The 32-bit AIX virtual memory space is split into 16, 256MB segments (0x0 – 0x15)
- Segment 0x0 is always reserved for the kernel
- Segment 0x1 is always reserved for the executable code (java)
- The rest of the segments may be laid out in different ways depending on the LDR_CNTRL=MAXDATA environment variable or the maxdata parameter compiled in the executable
- Example for the most common MAXDATA=0xA@DSA



MAXDATA

- By default, IBM Java will choose a generally appropriate MAXDATA value depending on -Xmx:
  - -Xmx > 3GB: MAXDATA=0@DSA
    - 3.5GB user space, 256MB malloc, 3.25GB mmap
  - 2.25GB < -Xmx <= 3GB: MAXDATA=0XB0000000@DSA
    - 3.25GB user space, malloc grows up, mmap grows down
  - -Xmx <= 2.25GB: MAXDATA=0XA0000000@DSA
    - 2.75GB user space, malloc grows up, mmap grows down, shared libraries in 0xD and 0xF
  - http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/topic/com.ibm.java.doc.user.aix32.60/user/aix_auto_ldr_cntrl.html
- 0@DSA is not very practical because it only leaves a single segment for native heap (malloc) which is usually insufficient

MAXDATA

- If you need more native memory (i.e. native OOM but not a leak), and your -Xmx is less than 2.25GB, explicitly setting 0xB@DSA may be useful by increasing available native memory by approximately 400MB to 600MB
- This causes the shared/mapped storage to start at 0xF and grow down. The cost is that shared libraries are loaded privately which increases system-wide virtual memory load (and thus potentially physical memory requirements!)
  - If you change X JVMs on one machine to the 0xB@DSA memory model, then the total virtual and real memory usage of that machine may increase by up to (N*(X-1)) MB, where N is the size of the shared libraries' code and data. Typically for stock WebSphere Application Server, N is about 50MB to 100MB.
  - Additionally, kernel trace in perfpmr can no longer trace detailed shared library activity which is sometimes needed by IBM support for performance issues.
- The change should not significantly affect performance, assuming you have enough additional physical memory.

MAXDATA

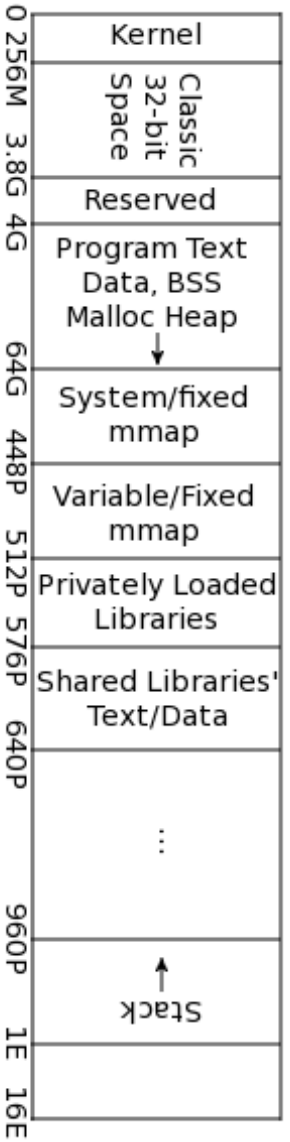
- Setting 0xB@DSA:
  - Admin Console -> Application Servers -> $SERVER -> Java and Process Management -> Process Definition -> Environment Entries
  - Click New. Name: LDR_CNTRL
    - Value: MAXDATA=0XB0000000@DSA
    - Click OK
  - Click New. Name: IBM_JVM_LDR_CNTRL_NEW_VALUE
    - Value: MAXDATA=0XB0000000@DSA
    - Click OK
  - Save, synchronize, and restart
- http://ibm.com/support/docview.wss?uid=swg21450144

MAXDATA

- Another effect of changing to the 0xB@DSA memory model is that segment 0xE is no longer available for mmap/shmat, but instead those allocations grow down in the same way as the Java heap. If your -Xmx is a multiple of 256MB (1 segment), and your process uses mmap/shmat (e.g. client files), then you will have one less segment for native memory. This is because native memory allocations (malloc) cannot share segments with mmap/shmat (Java heap, client files, etc.).
- To fully maximize this last segment for native memory, you can calculate the maximum amount of memory that is mmapped/shmat'ed at any one time using svmon (find mmapped sources other than the Java heap and clnt files), and then subtract this amount from -Xmx. -Xmx is not required to be a multiple of 256MB, and making room available in the final segment may allow the mmapped/shmatted allocations to be shared with the final segment of the Java heap, leaving the next segment for native memory.
  - This only works if said mmaps/shmats are not made to particular addresses.

AIX Native Memory Layout (64-bit)

- Segment 0x0 (0x0 through 0x10000000) is always reserved for the kernel
- 0xF0000000 (3.8GB) through 0xFFFFFFFF (4GB) is reserved.
- 0x1_00000000 (4GB) to 0x07FFFFFF_FFFFFFFF (512PB) - Contains the application program text, application program data, the process heap, and shared memory or mmap services.
  - 0x10_00000000 (64GB) to 0x06FFFFFF_FFFFFFFF (448PB) – Fixed or system loader Mmaps
  - 0x07000000_00000000 (448PB) to 0x07FFFFFF_FFFFFFFF (512PB) – Fixed/variable mmaps
  - ~60GB of malloc heap before reconfiguring (note -Xcompressedrefs).
- 0x08000000_00000000 (512PB) to 0x08FFFFFF_FFFFFFFF (576PB) - Privately loaded objects.
- 0x09000000_00000000 (576PB) to 0x09FFFFFF_FFFFFFFF (640PB) - Shared library text/data.
- 0x0F000000_00000000 (960PB) to 0x0FFFFFFF_FFFFFFFF (1EB) - Application stack.



AIX Native Memory Layout (64-bit)

- The 64-bit AIX virtual memory space continues to use 256MB segments.
- If you have a leak, switching to 64-bit will not help because although you'll no longer receive native OutOfMemoryErrors, you'll simply start to page, which is just as bad (or even worse, because performance degrades but the process may not crash – a zombie).
- Always monitor paging (vmstat, topas, nmon, etc.)

Compressed References

- With 64-bit IBM Java >= 6, the generic JVM argument -Xcompressedrefs may be used:
  - Reduces processor cache miss rate, bus utilization & native memory used by the Java heap (thus less garbage collection as well)
  - In some cases, this option may reduce the performance overhead of switching the same application from 32-bit to 64-bit to within 5%, and reduce memory footprint by up to 50%
- The JVM accomplishes this by using 32-bit references with shifting & offsetting.
- With WAS >= 7, -Xcompressedrefs is enabled by default for -Xmx <= 25GB
- If -Xmx is small enough that it can fit in the 32-bit address space, then bit shifting is not necessary. If it can fit below 32GB, then bit offsetting is not necessary. Shift+Offsetting on Power CPUs is done with specialized processor instructions.
- Where the JVM puts the heap by default (based on size) may limit malloc heap to below this mmapped space (use svmon or truss to understand this).
  - Experiment by increasing -Xmx, or, starting in Java 6 SR 7 (WAS 7.0.0.9), explicitly set (try to fit it below 32GB and on page boundary): -Xgc:preferredHeapBase=0x400000000
    - May decrease throughput by a few %
  - http://publib.boulder.ibm.com/infocenter/javasdk/v6r0/topic/com.ibm.java.doc.diagnostics.60/diag/understanding/mm_compressed_references.html

Some Native Allocations

- The Java Heap itself is allocated using mmap/shmat
- Some malloc allocations include (but not limited to):
  - Native treads accompanying the Java Thread classes
  - Just-in-Time Compiler (JIT)
    - Various heuristics and levels of JIT compilation based on the number of calls, etc., so this may grow over time
    - Monitor with -Xjit:verbose={compileStart|compileEnd}
  - Class & Classloader data accompanying the Java Class & Classloader objects (monitor with javacores)
  - JNI, DirectByteBuffers, NIO, etc.
  - Type 2 DB Drivers, certain MQ clients, other 3rd party libs

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
- With verbosegc enabled, the Javacore has a GC flight recorder section, which may show:
  - J9AllocateIndexableObject() returning NULL!

Monitoring

- Use svmon to monitor native memory usage
  - #!/bin/sh
    # The process id to monitor is the first and only argument.
    PID=$1
    # The interval between command invocations, in seconds.
    INTERVAL=3
    # Echo the date line to record the start of monitoring.
    echo timestamp = `date +%s`
    # Echo the interval frequency.
    echo "svmon interval = $INTERVAL"
    # Run the system command at intervals.
    while ([-d /proc/$PID]) do
        svmon -r -m -P $PID
        sleep $INTERVAL
    done
- Run with:
  - nohup ./monitor.sh 9633862 > svmon.out 2>&1 &
- GCMV can graph this output

timestamp	Memory in the system virtual space	Memory in use	Pinned memory	Reserved address space (virtual memory)
date	MB	MB	MB	MB
1336152622000	509	509	0.06	615
1336152625000	509	509	0.06	615
1336152628000	509	509	0.06	615
1336152631000	509	509	0.06	615
1336152634000	509	509	0.06	615
1336152637000	509	509	0.06	615
1336152640000	509	509	0.06	615
1336152643000	509	509	0.06	615
1336152646000	509	509	0.06	615
1336152649000	509	509	0.06	615
1336152652000	509	509	0.06	615
1336152655000	509	509	0.06	615

Monitoring

- Understanding svmon:
  - Running with more details (range adds the Addr Range, mpss breaks down multi-size segments & pgsz summarizes by page):
    - svmon -P $PID -O range=on,mpss=on,pgsz=on
  - Example output:

```
   Pid Command          Inuse      Pin     Pgsp  Virtual
7798816 java           853330     9701     2464   876033
   PageSize            Inuse       Pin       Pgsp    Virtual
   s    4 KB           6210        21         0       4257
   m   64 KB           52945       605        154     54486
```

  - Values are in page sizes and default for the first line may depend, so just sum the values for each page size
    - Above example, total virtual = (4257*4KB)+(54486*64KB) ~= 3.3GB
  - A segment cannot contain both mallocs and mmap/shmats
  - A clnt page may be a filecache page which can be pushed out by the kernel if under memory pressure.

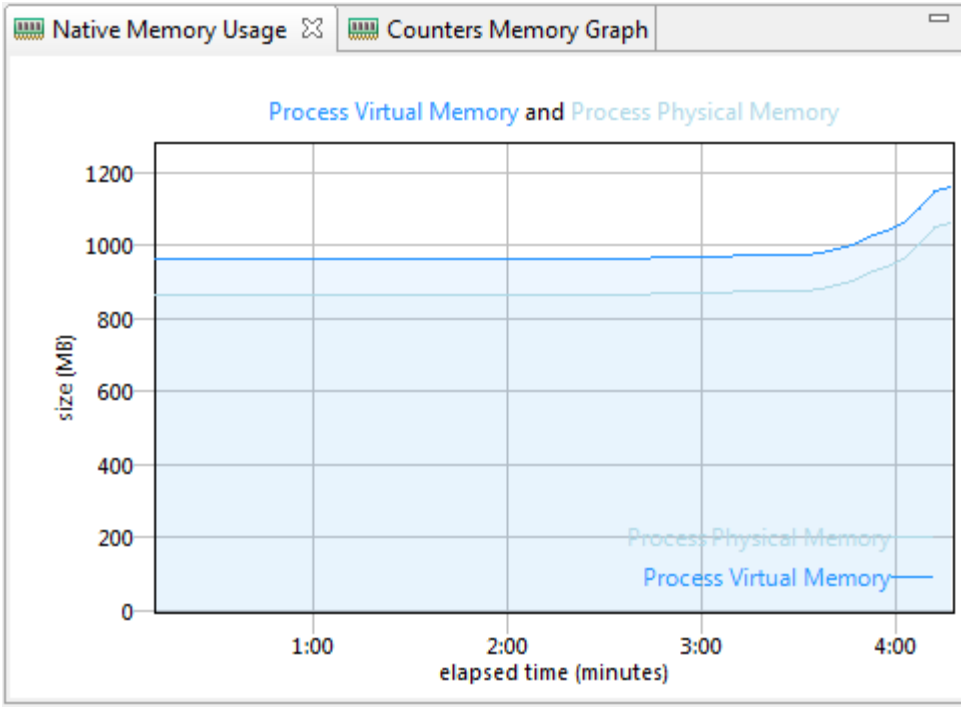
Monitoring

```
  Vsid      Esid Type Description              PSize  Inuse   Pin Pgsp Virtual
1097296         3 work working storage             m   3275     0    0    3275
                  Addr Range: 0..3554
```

- Vsid: Unique segment identifier, not very interesting.
- Esid: Virtual memory segment identifier. Example: Segment 0x3
- Psize: The size of pages in this segment (s=4KB, m=64KB, L=16MB, S=16GB)
- Inuse: Pages in memory – may be shared with other processes (by default, sorted by this column, descending)
- Virtual: Pages in virtual memory
- Addr Range: One or more ranges of counts of pages allocated. This may be more than inuse or virtual if they haven't been used
  - Large discrepancy with InUse may mean free page fragmentation
- **In general, for leaks**, you want to look at total virtual memory over time
- Look at the virtual column * the page size.

Monitoring

- Recent versions of Health Center have 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



Native Memory Table | Counters Memory Table

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

Javacore Native Memory Information

- In IBM Java >= 626 (WAS >= 8)

```
0SECTION       NATIVEMEMINFO subcomponent dump routine
NULL           ================================
0MEMUSER
1MEMUSER       JRE: 558,675,856 bytes / 1523 allocations
1MEMUSER       |
2MEMUSER       +--VM: 556,110,160 bytes / 1086 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Classes: 2,485,496 bytes / 112 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Memory Manager (GC): 547,800,800 bytes / 194 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Java Heap: 536,870,912 bytes / 1 allocation
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Other: 10,929,888 bytes / 193 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Threads: 4,880,596 bytes / 175 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Java Stack: 122,672 bytes / 17 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Native Stack: 4,620,288 bytes / 19 allocations
3MEMUSER       |  |  |
4MEMUSER       |  |  +--Other: 137,636 bytes / 139 allocations
```

Javacore Native Memory Information (Continued)

```
3MEMUSER       |  +--Trace: 170,488 bytes / 249 allocations
2MEMUSER       |  |
3MEMUSER       |  +--JVMTI: 17,328 bytes / 13 allocations
2MEMUSER       |  |
3MEMUSER       |  +--JNI: 20,032 bytes / 51 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Port Library: 7,264 bytes / 60 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Other: 728,156 bytes / 232 allocations
1MEMUSER       |
2MEMUSER       +--JIT: 1,692,744 bytes / 171 allocations
2MEMUSER       |  |
3MEMUSER       |  +--JIT Code Cache: 524,288 bytes / 1 allocation
2MEMUSER       |  |
3MEMUSER       |  +--JIT Data Cache: 524,336 bytes / 1 allocation
2MEMUSER       |  |
3MEMUSER       |  +--Other: 644,120 bytes / 169 allocations
1MEMUSER       |
2MEMUSER       +--Class Libraries: 872,952 bytes / 266 allocations
2MEMUSER       |  |
3MEMUSER       |  +--Harmony Class Libraries: 1,024 bytes / 1 allocation
2MEMUSER       |  |
3MEMUSER       |  +--VM Class Libraries: 871,928 bytes / 265 allocations
```

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.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.ibm.com/support/docview.wss?uid=swg21317658
- Disable AIO: http://www.ibm.com/support/docview.wss?uid=swg21366862
- Switch to 64-bit JVMs
- Links
  - http://www.ibm.com/support/docview.wss?uid=swg21373312

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 (thread structs, etc)
    - 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
- Aggregation scripts:
  - get_memory_use.pl

Other Analysis

- To diagnose JVM memory leaks within the Java product itself, use the command line option:
  - -Xcheck:memory:callsite=1000
    - http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/topic/com.ibm.java.lnx.70.doc/diag/problem_determination/win_mem_trace_memorycheck.html
  - On older versions: -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
    - > !findallcallsites
- If there is a leak after restarting applications, 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

Other Analysis

- If you suspect third party JNI code:
  - -Xcheck:jni:all will print warnings for improper behavior (equivalent to -Xrunjnichk)
- Look for “JVM” messages in stderr

Other Analysis

- Classic eye catcher hunting is another technique. This is essentially what the callsite analysis is.
- If third party native libraries use eye catchers, you can search for them to estimate each library's allocated memory.
  - https://www.ibm.com/developerworks/mydeveloperworks/blogs/kevgrig/entry/native_c_c_eye_catcher?lang=en

```
$ od -N 100 -xc core.dmp +0x0
0000000    0077   0000   0fee   ddb2   0000   0000   0000   1870
      \0   w  \0  \0 017         \0  \0  \0  \0  \0  \0 030   p
0000010    0000   0000   0000   1a30   0000   0000   0000   1328
      \0  \0  \0  \0  \0  \0 032   0  \0  \0  \0  \0  \0  \0 023   (
0000020    0000   0014   0000   0000   0000   0000   0000   2d58
      \0  \0  \0 024  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0   -   X
0000030    0000   0000   0000   0006   0000   0000   0000   7b78
      \0  \0  \0  \0  \0  \0  \0 006  \0  \0  \0  \0  \0  \0   {   x
```

Data in System Dumps

- For any advanced analysis of native memory, system dumps (core dumps) are required.
  - Jextract each system dump because that makes it easier to run dbx and other utilites on it
- Critical: Ensure you have proper ulimits set:
  - http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/topic/com.ibm.java.aix.70.doc/diag/problem_determination/aix_setup_full_core.html
- Some analysis such as native memory held by DirectByteBuffers is available in the IBM Extensions for Memory Analyzer
  - http://www.ibm.com/developerworks/java/jdk/tools/iema/
  - Query Browser → IBM Extensions → Java → DirectByteBuffers

Getting a System Dump

- Create a system dump on OOM instead of PHDs:
  - -Xdump:heap:none -Xdump:system:events=systhrow,filter=java/lang/OutOfMemoryError,range=1..1,request=serial+exclusive+prepwalk
  - Starting in WAS 8.0.0.2, a system dump is produced on the first OOM by default.
- Wsadmin (WAS >= 7): AdminControl.invoke(AdminControl.completeObjectName("type=JVM,process=server1,*"), "generateSystemDump")
- Programmatically with com.ibm.jvm.Dump.SystemDump()
- Jextract: <WAS>/java/jre/bin/jextract $DUMP
  - Upload just the produced ZIP file. The ZIP contains the core dump, so you can delete the core dump after the ZIP is produced.
  - Also save the console output of jextract in a separate file and upload as well

Getting a System Dump (Continued)

- IBM Health Center can acquire a system dump
- The trace engine allows system and PHD dumps to be triggered on method entry or exit. This produces a system dump when the Example.trigger() method is called
  - -Xtrace:maximal=mt,trigger=method{com/ibm/example/Example.trigger,sysdump}
- Set a range to take dumps between the first and 5th method invocations:
  - -Xtrace:maximal=mt,trigger=method{com/ibm/example/Example.trigger,sysdump,,5,1}
- Ensure enough physical memory for best performance.

dbx

- How to load dbx
  - Unzip the jextracted zip file
  - Run: dbx -p /=./ ./java $COREDUMP
  - The -p argument ensures that dbx loads the libraries from the ZIP (which lets you run dbx on the dump from a different machine than where the dump was generated)

MALLOCDEBUG

- Use the dbx malloc command
  - (dbx) malloc
    …Statistical Report on the Malloc Subsystem:
    Heap 0
    bytes acquired from sbrk()..... 1112935520
    bytes in the freespace tree..... 1018816
    bytes held by the user..... 1111916704
    allocations currently active..... 2279
    The Process Heap
    Initial process brk value..... 0x000001001001baa0
    current process brk value..... 0x00000100525b0700
- By default, malloc just calls sbrk to increase the data segment
  - Note: Returns ENOMEM if sbrk runs into a shmat/mmap segment.
  - http://pic.dhe.ibm.com/infocenter/aix/v7r1/topic/com.ibm.aix.basetechref/doc/basetrf1/brk.htm
- Therefore, subtract “initial brk” from “current brk” above to get the total virtual memory used by malloc (above example=1GB)
- The allocator then breaks this down into “heaps.” Normally, there is just a single heap (Heap 0).
  - Acquired=Total space used. Freespace=In the free pool. Held=Active.

dbx

- The other major user of memory is mmap or shmat
  - Run the commands “coremap shm” and “coremap mmap”
  - (dbx) coremap shm
    Mapping: Shared Memory (**size=0x9c400000**)
    from (address): 0x40000000 – 0xdc400000
    to (offset) : 0x429582a5 – 0xded582a5
    in file : 10420228.dmp
  - (dbx) coremap > outputfile.txt
    Mapping: Data (**size=0x425b0700**)
    from (address): 0x10010000000 – 0x100525b0700
    to (offset) : 0x3a7805 – 0x42957f05
    in file : 10420228.dmp
    Mapping: Stack (size=0x2000)
    from (address): 0xfffffffffffe000 – 0x1000000000000000
    to (offset) : 0x7c38 – 0x9c38
    in file : 10420228.dmp …
- In this example, one of the shared memory areas is 2.5GB, the size of our Java heap
- Also, you can see the Data section is 1GB which is what we saw on the last slide

dbx

- You can also check ulimits (particularly DATA):
  - Print ulimits

```
(dbx) proc rlimit
rlimit name:          rlimit_cur              rlimit_max       (units)
 RLIMIT_DATA:        (unlimited)             (unlimited)       bytes
 RLIMIT_STACK:         33554432              4294967296        bytes
 RLIMIT_CORE:        (unlimited)             (unlimited)       bytes
 RLIMIT_NOFILE:          2000                (unlimited)       descriptors
…
```

  - List file descriptors
    - (dbx) fd

```
0: { fp = 0x0000000000000001, flags = ALLOCATED, count = 0 }
1: { fp = 0x0000000000000001, flags = ALLOCATED, count = 0 }
2: { fp = 0x0000000000000001, flags = ALLOCATED, count = 0 }
```

Analysis

- MALLOCDEBUG
  - Environment variable which instructs the AIX malloc implementation(s) to track various aspects of native memory allocation through malloc.
  - See
    - http://pic.dhe.ibm.com/infocenter/aix/v6r1/topic/com.ibm.aix.genprogc/doc/genprogc/debug_malloc.htm#nad200003150712pm
    - http://www.ibm.com/developerworks/java/library/j-nativememory-aix/
    - http://www.ibm.com/developerworks/aix/library/au-mallocdebug.html
- This option may have a very significant performance overhead. Before running in production, test it with a similar load in a test environment.

MALLOCDEBUG

- Set environment variable on the target application server:
  - http://www.ibm.com/support/docview.wss?uid=swg21254153
  - MALLOCDEBUG=report_allocations,stack_depth:3
- Implicitly enables the malloc log (MALLOCDEBUG=log) which is an in-memory database held in the same virtual memory as the JVM. Adds at least 50-100 bytes for 32-bit and 100-200 bytes for 64-bit for each malloc (varies based on stack_depth).
- Only use log:extended if you need this extra information:
  - The process ID and thread ID of the allocation.
  - The sequence number of the allocation from process startup.
  - The real time in which the allocation was made.

MALLOCDEBUG

- Avoid using a stack depth greater than 3
  - “The stack depth of 3 provides only a limited stack trace. However, the use of larger stack depths with a Java application can cause crashes because the debug malloc facility does not understand the stack frames used for JIT compiled code.”
    - http://publib.boulder.ibm.com/infocenter/javasdk/tools/topic/com.ibm.java.doc.igaa/_1vg000121410cbe-1195c23a635-7ffe_1003.html
- stack_depth default is 4 and maximum is 64

MALLOCDEBUG

- Reproduce the problem and stop the JVM. Report produced in native_stderr.log with each un-freed allocation:
  - Allocation #0: 0x3002FD60
    Allocation size: 0x2A0
    Allocated from heap: 0
    Allocation traceback:
    0xD01DC934 malloc
    0xD01288AC init_malloc
    0xD012A1F4 malloc
- Run format_mallocdebug_op.sh on native_stderr.log
  - http://www.ibm.com/developerworks/aix/library/au-mallocdebug.html
- Aggregates all stacks and reports largest leaks
  - ZIP_Put_In_Cache
    readCEN
    calloc_common
    malloc
    ##################################
    533676 bytes leaked in 127 Blocks

System Dumps

- Along with MALLOCDEBUG, gather:
  - Periodic system dumps (at least one after startup and one before stopping) – jextract each one
  - Run the svmon script from the previous slide as well as the following in the background:
    - nohup vmstat -It 2 > vmstat.out 2>&1 &

MALLOCDEBUG

- If MALLOCDEBUG is enabled, dbx can query the log:
  - (dbx) malloc allocation
    Allocations Held by the Process:

```
          ADDRESS       SIZE HEAP     SEQ STACK TRACEBACK
0x000001001001bab0          32    0       0 0x0000000000000000
                                    0x090000000002a910
0x000001001001baf0        1232    0       1 0x0000000000000000
                                    0x090000000004a6a8
```

- Seeing the MALLOCTYPE and debug options:
  - (dbx) malloc
    The following options are enabled:
    Implementation Algorithm..... Default Allocator (Yorktown)
    Malloc Log
    Stack Depth..... 3
    Report Allocations

dbx

- Useful dbx commands
  - MALLOCDEBUG log unfreed allocations of particular size
    (dbx) malloc allocation size == 0x10
  - Various process information including PID, PPID, uid, data size

```
(dbx) proc
-Identification/Authentication Info--------------------------
 pi_pid:       10420228              pi_sid:        6488316
 pi_ppid:       6488316              pi_pgrp:             0
 pi_uid:              0           pi_suid:             0
-Memory Usage-----------------------------------------------
 pi_drss:      0x00000000000415fb     pi_trss:      0x0000000000000020
 pi_dvm:       0x00000000000415fb     pi_pi_prm:    0x0000000000000019
 pi_tsize:     0x0000000000013f13     pi_dsize:     0x00000000425b0700
 pi_sdsize:    0x0000000000000000
```

dbx

- Useful dbx commands
  - Print memory at an address as hex: Address / X
    - 0x4EA07000/X
  - Multiple bytes: Address / NX
    - 10 Bytes > 0x4EA07000/10X
  - Print null terminated string at address: Address / s
    - 0x4EA07000/s
- Related AIX commands:
  - All strings and their hex offsets (at least N printable characters followed by a null, default N is 4): strings -t x $CORE_FILE
  - All bytes in both hex & characters starting at offset 0x988 and only show 100 bytes: od -N 100 -xc $CORE_FILE +0x988
    - Alternatively: od -v -A x -N 100 -j 0x2B521000 -t xc $CORE_FILE

Truss

- If you want to know exactly where the heap is being mmapped
  - Use startServer.sh -script to generate the WAS command
  - Prepend the following to the java command:
    - exec truss -f -d -t kmmap,mmap,munmap -i -s!all -o truss.out java ...
- You could also use this to dynamically attach to a process to watch mallocs and frees
  - truss -f -d -t malloc,free -i -s!all -p $PID
- If using bash, escape the ! with \!
- The list of system calls may be longer (e.g. shmat, xmalloc, etc.)

ProbeVue

- Similar to DTrace/SystemTap. Ships with AIX >= 6.1
- Example malloc.e file that prints every malloc stack:
  - Not a practical example – needs to track __rv, hold stack in array, then remove from free based on __arg1, then dump in @@END
  - ```
    char * malloc(int size);
    @@BEGIN { printf("probevue script started\n"); }
    @@uft:$1:*:malloc:entry {
      stktrace_t t1;
      t1 = get_stktrace(5);
      printf("malloc(%d)\n%t", __arg1, t1);
    }
    @@END { printf("probevue script ended\n"); }
    ```
- Running it: LDR_CNTRL=MAXDATA=0XB0000000@DSA probevue -o probevue.out malloc.e $TARGETPID
- Watch for trace buffers filling in stderr – too many printfs (see probevctrl)
    ```

Conclusion

- Native memory issues are notoriously difficult.
- MALLOCDEBUG works well but may have a very large overhead.
- Consider isolating or avoiding the issue instead.

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