IBM Software Group

Linux Native Memory issues for WebSphere Application Server

Kevin Grigorenko (kevin.grigorenko@us.ibm.com)
WebSphere Foundation SWAT Team
November 12, 2013

WebSphere® Support Technical Exchange

Agenda

- Overview
- Linux Native Memory Layout
- Detection
- Monitoring
- Isolation & Avoidance
- Analysis

Overview

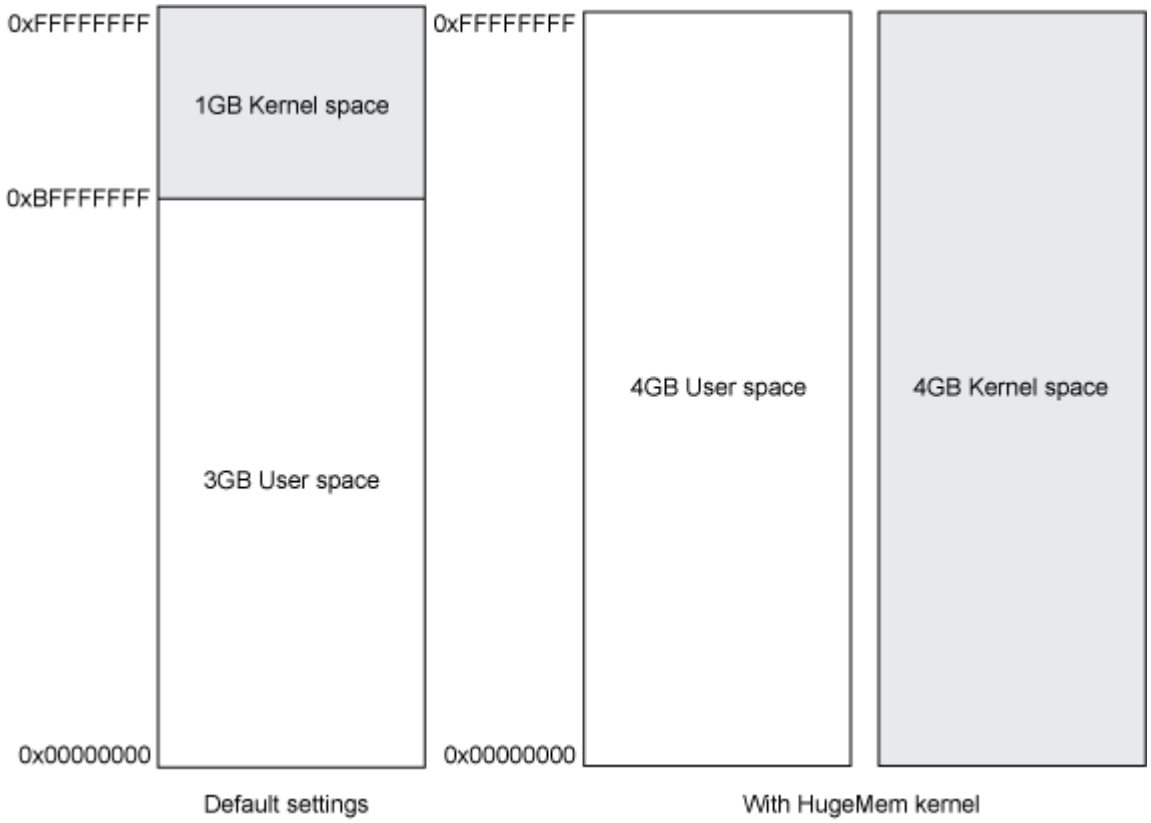
- Native memory issues are notoriously difficult
  - May cause: crashes, thrashing, OS instability, high CPU
  - Isolation and/or avoidance is often easier than analysis
- This presentation covers how to detect, monitor, avoid, isolate, and analyze, in that order.
- The bad news first: Built-in Linux native memory leak detection tools are probably the worst of all modern operating systems (glibc mtrace doesn't give stacks). AIX, z/OS, Solaris, and Windows have good, built-in leak detection.
- The good news: Linux provides a lot of different ways to probe and monitor memory usage (or is that bad news?).
- This is an educational presentation to enhance customers' self-help capabilities and does not represent specific advice or recommendations that IBM supports. For example, it may include data and techniques that IBM Support does not guarantee it can analyze.

Native Memory Basics

- Native memory generally means the virtual native memory of a process or address space. Includes:
  - Executable code and static data
  - Statically and dynamically loaded libraries (code and static data)
  - Dynamically allocated memory
  - Thread stacks
  - etc.
- This native memory 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
  - A 32-bit process running in a 64-bit OS still has a 32-bit virtual address space.
- The Linux native memory layout depends on the physical processor (e.g. x86_64/x64, PPC, 390, IA64, ARM, etc.) and the particular type of kernel built for that processor.
- This presentation looks at the common kernels for x86-32 (often referred to as i386/i586/i686/x86) and x86-64/x64 as examples (sometimes referred to as AMD64 for historical reasons)
  - Things may be different for Linux running on POWER, zLinux, Itanium, ARM, etc.

Linux Native Memory Layout (32-bit)

- By default, the Linux 32-bit virtual memory space is split into a 3GB user space and a 1GB kernel space
- A 32-bit process on a 64-bit kernel will have almost all 4GB.
- Similarly, the 32-bit hugemem kernel supports a 4GB user space
  - May have significant overhead because the hardware's address translation buffer can't be shared between kernel and user space



Malloc versus mmap

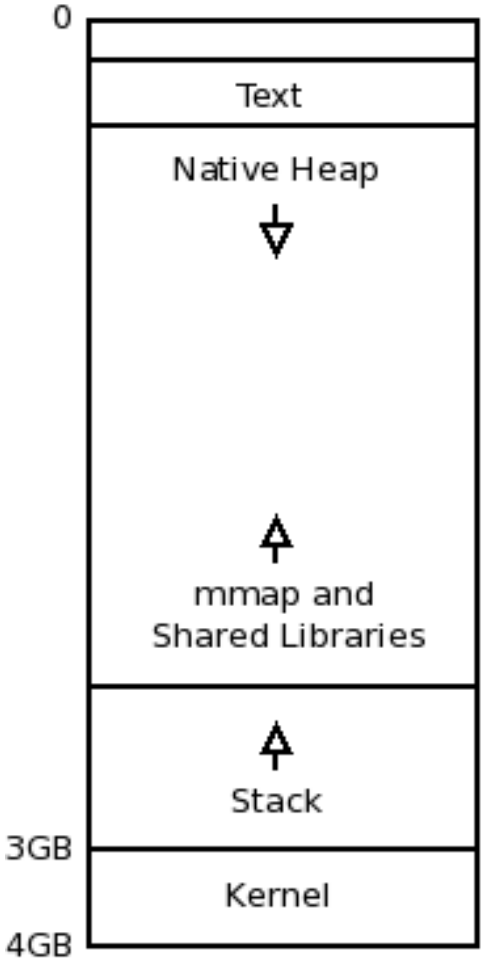
- Both return raw virtual address space memory, so they are both “native memory”
- Malloc
  - This is a POSIX standard, C library function that requests virtual address space memory
  - Different implementations of malloc with different characteristics (performance, space usage, etc.)
  - Most malloc implementations will manage freed mallocs for more quickly returning available space in future calls, including coalescing nearby free areas (potential for fragmentation)
    - This means that there is additional virtual memory “management” overhead
  - Some malloc implementations such as glibc will actually use mmap based on allocation sizes
  - Uses the brk/sbrk system calls to increase or decrease what's classically called the “heap” or “data segment” which is a contiguous block of memory designed for this purpose.
  - Mallocs may be aligned (often to 4 or 8 byte boundaries)
- Mmap
  - Not part of a contiguous area, so fragmentation is not as much of a problem
  - Freed mmap's are immediately returned to the system
  - Anonymous mmap pages are automatically zero-filled, so a bit more overhead
  - Always page-sized, so might be more memory waste (and the whole page has to be zero'd)
- Generally, malloc is faster than mmap for small, periodic, transient allocations
- Other related functions: calloc, realloc

More on malloc

- When glibc malloc detects mutex contention (i.e. concurrent mallocs), then the native heap is broken up into sub-pools called arenas.
  - This is achieved by assigning threads their own memory pools and by avoiding locking in some situations. The amount of additional memory used for the memory pools (if any) can be controlled using the environment variables MALLOC_ARENA_TEST and MALLOC_ARENA_MAX. MALLOC_ARENA_TEST specifies that a test for the number of cores is performed once the number of memory pools reaches this value. MALLOC_ARENA_MAX sets the maximum number of memory pools used, regardless of the number of cores.
- This was introduced in glibc 2.11 (for example, customers upgrading from RHEL 5 to RHEL 6)
- The default maximum arena size is 1MB on 32-bit and 64MB on 64-bit. The default maximum number of arenas is the number of cores multiplied by 2 for 32-bit and 8 for 64-bit.
- This can increase fragmentation because the free trees are separate.
- In principle, the net performance impact should be positive of per thread arenas, but testing different arena numbers and sizes may result in performance improvements depending on your workload.
  - You can revert the arena behavior with the environment variable MALLOC_ARENA_MAX=1

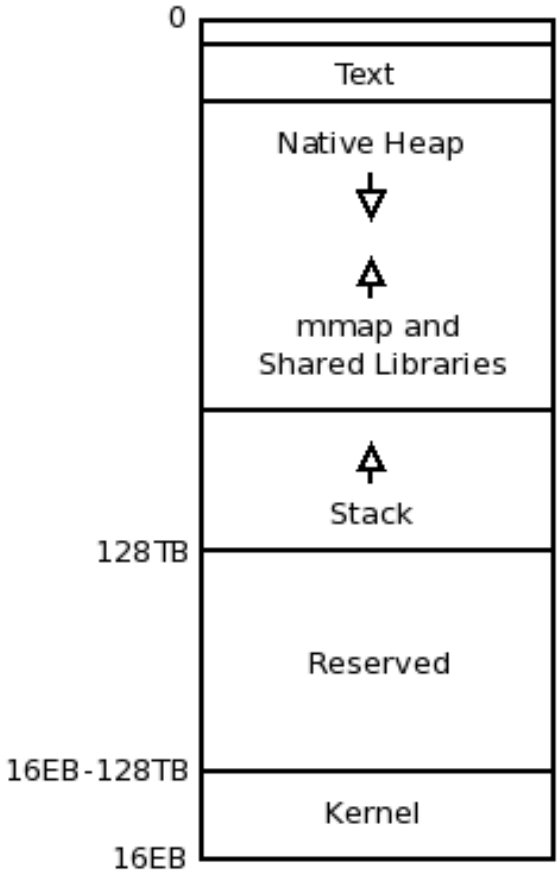
Linux Native Memory Layout (32-bit)

- In recent kernels, the text is at the bottom, stack at the top, and mmap/heap sections grow towards each other in a shared space (although they cannot overlap).
- By default, the malloc implementation in glibc (which was based on ptmalloc, and that was based on dlmalloc) will allocate into either the native heap (sbrk) or mmap space, based on various heuristics and thresholds:
  - If there's enough free space in the native heap, allocate there.
  - Otherwise, if the allocation size is greater than some threshold (slides between 128KB and 32/64MB based on various factors [1]), allocate a private, anonymous mmap instead of native heap (mmap isn't limited by ulimit -d)
- [1]: http://man7.org/linux/man-pages/man3/mallopt.3.html



Linux Native Memory Layout (64-bit)

- The x86_64 processor memory management unit supports up to 48-bit virtual addresses (256TB).
  - https://www.kernel.org/doc/ols/2001/x86-64.pdf
- The “canonical form” of addresses creates two ranges to use these 48 bits: 0x through 0x00007FFF'FFFFFFFF, and from 0xFFFF8000'00000000 through 0xFFFFFFFF'FFFFFFFF, thus providing two 128TB spaces (think of the second space as “signed” - i.e. -128TB to 0).
- Starting in kernel 2.6.11, the user space gets the lower half, i.e. up to 128TB, and the kernel the other half:
  - https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt
  - “0000000000000000 - 00007fffffffffff (=47 bits) user space, different per mm”
  - $ sudo ls -lh /proc/kcore
    -r-------- 1 root root 128T Jul 9 12:07 /proc/kcore



Linux Native Memory Layout (64-bit)

- 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, top, etc.)

Linux Native Memory - /proc/(s)maps

- Starting with Linux 2.4, the /proc filesystem contains a “maps” file for each process that describes the virtual address space layout of a process.
  - https://www.kernel.org/doc/Documentation/filesystems/proc.txt
- Example:
  - $ cat /proc/28685/maps
    address perms offset dev inode pathname
    00400000-0040c000 r-xp 00000000 fd:01 7570 java
    01f50000-01f71000 rw-p 00000000 00:00 0 [heap]
    3e9c58d000-3e9c58e000 rw-p 0018d000 fd:01 1043 /lib64/libc-2.12.so
    7f1600000000-7f160023c000 rw-p 00000000 00:00 0
    7f17161cd000-7f17161cf000 rw-p 00007000 fd:01 8 libjli.so
    7fff89c65000-7fff89c78000 rwxp 00000000 00:00 0 [stack]
    7fff89d5d000-7fff89d5e000 r-xp 00000000 00:00 0 [vdso]
    ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
- Permissions: r = read, w = write, x = execute, s = shared, p = private (copy on write)
- [heap] = the native heap of the program, [stack] = the stack of the main process, [vdso]/[vsyscall] = the "virtual dynamic shared object" (used for kernel system calls)
- In theory, a more detailed understanding of which process is using which page can be done with /proc/pid/pagemap, /proc/kpageflags, and /proc/kpagecount
  - https://www.kernel.org/doc/Documentation/vm/pagemap.txt

Linux Native Memory - /proc/(s)maps

- In recent kernel versions, /proc/pid/smaps provides the same information, but also breaks down each line, for example:
  - 01f50000-01f71000 rw-p 00000000 00:00 0 [heap]
    Size: 132 kB
    Rss: 132 kB
    Pss: 132 kB
- Pss is the proportional set size which is a subset of RSS. If a region is mapped across multiple processes (e.g. shared library), the amount resident is accounted to each process proportionally.
- pmap -x $PID is another, prettier way to look at the maps/smaps data
  - Address Kbytes RSS Dirty Mode Mapping
    0000000000400000 48 44 0 r-x-- java...
    00007fff03f68000 80 20 20 rwx-- [stack]
    ---------- ------ ------ ------
    total kB 4914116 369140 310696

Linux Native Memory

- In recent distributions, the “smem” program may be installed to better understand proportional set size across processes. For example:
  - $ smem -t -R -w -s pss

```
  PID User     Command    Swap     USS     PSS     RSS ...
12268 kevin    dia           0   11556   13127   24240
15836 kevin    plugin        0   33048   36205   45620
29054 kevin    firefox       0  310200  314430  330004
-------------------------------------------------------------
  140 1                      0 1145896 1204860 1566956
```

- /proc/PID/status also provides some interesting total statistics:
  - VmPeak: Peak virtual memory size.
  - VmSize: Virtual memory size.
  - VmHWM: Peak resident set size ("high water mark").
  - VmRSS: Resident set size.
  - VmData: Size of data (native heap)
  - VmStk: Size of stack
  - VmExe: Size of text segments
  - VmLib: Shared library code size.

Overcommit and the OOM Killer

- “By default, Linux follows an optimistic memory allocation strategy. This means that when malloc() returns non-NULL there is no guarantee that the memory really is available. In case it turns out that the system is out of memory, one or more processes will be killed by the OOM killer.”
  - http://man7.org/linux/man-pages/man3/calloc.3.html
- “/proc/sys/vm/overcommit_memory: This file contains the kernel virtual memory accounting mode. Values are:
  0: heuristic overcommit (this is the default)
  1: always overcommit, never check
  2: always check, never overcommit

  In mode 0, calls of mmap(2) with MAP_NORESERVE are not checked... leading to the risk of getting a process "OOM-killed"... In mode 2 (available since Linux 2.6), the total virtual address space on the system is imited to (SS + RAM*(r/100)), where SS is the size of the swap space, and RAM is the size of the physical memory, and r is the contents of the file /proc/sys/vm/overcommit_ratio.”
  - http://man7.org/linux/man-pages/man5/proc.5.html

Overcommit and the OOM Killer

- Check /var/log/messages for “Out of memory.” For example:
  - kernel: Out of memory: Killed process 1126 (db2sysc).
- Snapshot total system memory usage: cat /proc/meminfo. Key statistics:
  - MemTotal: Total usable RAM
  - MemFree: Subset of MemTotal that is available
  - Buffers: Temporary disk block storage
  - Cached: Total filesystem page cache size
  - CommitLimit: If overcommit_memory=2, max memory that can be requested.
- The free command summarizes /proc/meminfo:
  - $ free -m

```
             total  used  free  shared buffers    cached
Mem:         15569  5900  9669     0    222      3612
-/+ buffers/cache:  2065 13504
```

  - Buffers/cache can be pushed out under pressure, so available memory for programs is under the free column on the “-/+ buffers/cache” line, i.e. 13.5G
  - Same info in vmstat by adding buff and cache columns to the free column

Overcommit and the OOM Killer

- The kernel has an algorithm for deciding who process(es) to kill on OOM:
  - /proc/[pid]/oom_score_adj (since Linux 2.6.36) [use oom_adj in previous versions]: The badness heuristic assigns a value to each candidate task ranging from 0 (never kill) to 1000 (always kill) to determine which process is targeted... For example, if a task is using all allowed memory, its badness score will be 1000. If it is using half of its allowed memory, its score will be 500... root processes are given 3% extra memory over other tasks.
  - The value of oom_score_adj is added to the badness score before it is used to determine which task to kill. Acceptable values range from -1000 (OOM_SCORE_ADJ_MIN) to +1000... The lowest possible value, -1000, is equivalent to disabling OOM- killing entirely for that task, since it will always report a badness score of 0.
  - http://man7.org/linux/man-pages/man5/proc.5.html
- On 32-bit kernels, there is an important distinction between “low” (kernel) and “high” (user) memory, and exhaustion of low memory will also run the OOM killer. On 64-bit, this distinction is not necessary.

Overcommit and the OOM Killer

- The big question: should you disable overcommit?
  - sysctl vm.overcommit_memory=2
    echo "vm.overcommit_memory=2" >> /etc/sysctl.conf
- Perhaps an analogy from the Linux kernel mailing list will help:
  - An aircraft company discovered that it was cheaper to fly its planes with less fuel on board.The planes would be lighter and use less fuel and money was saved. On rare occasions however the amount of fuel was insufficient, and the plane would crash. This problem was solved by the engineers of the company by the development of a special OOF (out-of-fuel) mechanism. In emergency cases a passenger was selected and thrown out of the plane... A large body of theory was developed and many publications were devoted to the problem of properly selecting the victim to be ejected. Should the victim be chosen at random? Or should one choose the heaviest person? ... Now that the OOF mechanism existed, it would be activated every now and then, and eject passengers even when there was no fuel shortage. The engineers are still studying precisely how this malfunction is caused.

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%. These numbers are for a benchmark relative to the same workload on 64-bit without uncompressed references.
- 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.
- Some native structures such as those representing classes will be allocated below 4GB. Where the JVM puts the heap by default (based on size) may limit available space to the allocations that need to be put below 4GB.
  - 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
- 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 ps 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 "ps interval = $INTERVAL"

    # Run the system command at intervals.
    while ([-d /proc/$PID]) do
        ps -p $PID -o pid,vsz,rss
        sleep $INTERVAL
    done
- Run with:
  - nohup ./monitor.sh 9633862 > native.out 2>&1 &
- GCMV can graph this output

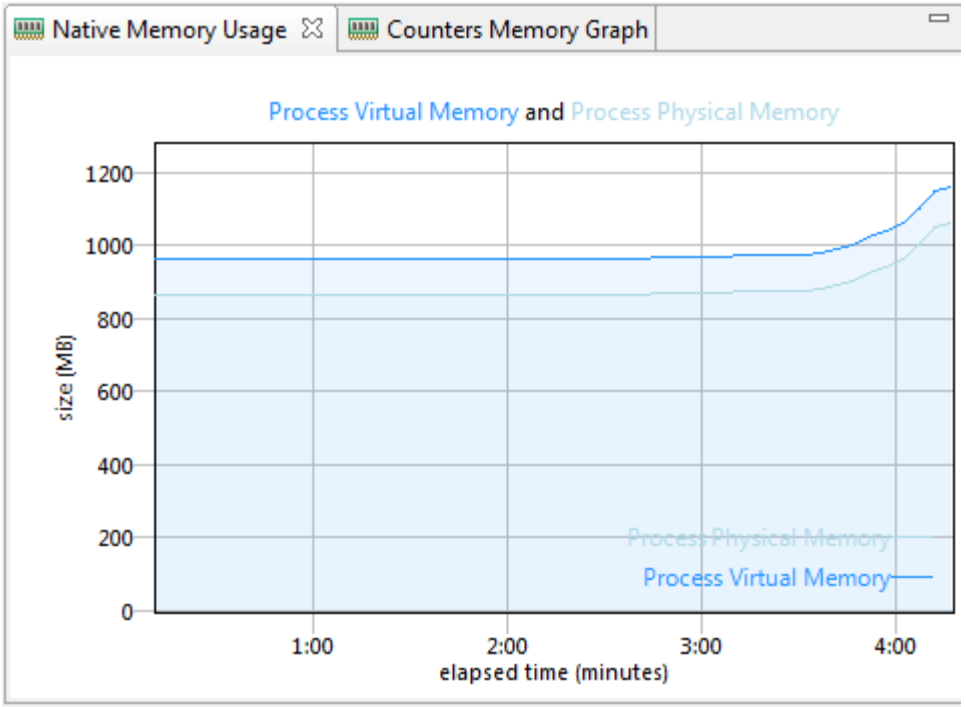
timestamp	Memory in the system virtual space	Memory in use	Pinned memory	Reserved address space (virtual memory)
date	MB	MB	MB	MB
1336152622000	509	509	0.06	615
1336152625000	509	509	0.06	615
1336152628000	509	509	0.06	615
1336152631000	509	509	0.06	615
1336152634000	509	509	0.06	615
1336152637000	509	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 ps:
  - The two key values are VSZ and RSS
  - VSZ is the virtual size of the process in KB
  - RSS is the resident set size in KB
- Historical monitoring is very important:
  - atop
  - nmon
  - sar (system level only, no process historical data)

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...

```
2MEMUSER       |  |
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
```

Ulimits and NOOMs

- Ulimits commonly cause NOOMs. Actually, a NOOX, where X is:
  - open files (-n): maximum number of open file descriptors
    - A socket uses a file descriptor
    - Cannot be set to -1/unlimited, max value is 1048576
    - Consider: 50000
  - nproc (-u): max user processes
    - User-wide, not process-wide (a Java thread counts as a process)
    - RHEL 6 (bug 432903) added a soft limit of 1024 to guard against “forkbombing.” Added to /etc/security/limits.d/90-nproc.conf. This file takes precedence over /etc/security/limits.conf (see bug 919793).
    - OutOfMemoryError" "Failed to create a thread: retVal ..., errno 11
    - Consider: 131072
  - Also, data segment (-d) (malloc), virtual size (-v) (all virtual process memory)
  - cat /proc/$PID/limits to display current values for a process
  - Either update limits.conf or the node agent, and restart
  - While you're there, update core (-c) and file size (-f) to unlimited for sysdumps!

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
  - One symptom of this is if the current thread in the javacore at the time of a native OOM is in:
    - at java/nio/DirectByteBuffer.<init>
      at java.nio.ByteBuffer.allocateDirect
      at
      com.ibm.ws.buffermgmt.impl.WsByteBufferPoolManagerImpl.allocateBufferDirect
- 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

DirectByteBuffers

- java.nio.DirectByteBuffer (DBB) is a class provided by the Java Development Kit class libraries to allocate and manipulate native memory byte arrays.
- When all Java references are gone and the DBB is ready for garbage collection, a PhantomReference is used to actually destroy it (so that it can free the native memory). This is similar to a finalizer and PhantomReference garbage collection may be delayed, particularly with generational garbage collection policies (the classic “iceberg” problem).
- Use -Dsun.nio.MaxDirectMemorySize=BYTES (e.g. 1073741824) which forces a System.gc that cleans up any PhantomReferences when that number of bytes is allocated for DBBs.
  - If this value is too low, until recent JVM versions, you could get “full GC storms” when multiple threads are waiting to allocate a DBB and System.gcs are run.
- If you can get a system dump, you can find out all DirectByteBuffer instances and how much native memory they hold with the IBM Extensions for Memory Analyzer
  - http://www.ibm.com/developerworks/java/jdk/tools/iema/
  - Query Browser → IBM Extensions → Java → DirectByteBuffers
- Before Java 7, every DBB request rounded to the page size, so there could be tremendous waste (shown in the IEMA query): http://bugs.sun.com/view_bug.do?bug_id=4837564

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
- If there is a leak after restarting applications, may be a classloader leak. The IBM Extensions for Memory Analyzer has a query for this:
  - Classic example is an application spawning an unmanaged thread – when it's restarted, the old classloader still has a reference
  - http://www.ibm.com/developerworks/websphere/techjournal/1103_supauth/1103_supauth.html#sec10
- The IBM ClassLoader Analyzer is a graphical tool to also understand classloader leaks: https://www.ibm.com/developerworks/mydeveloperworks/groups/service/html/communityview?communityUuid=a0a94b0d-38fe-4f4e-b2e6-4504b9d3f596

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

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

```
$ hexdump -C core.dmp | grep "d0 fa ad de"
00002cb0  00 00 00 00 d0 fa ad de  00 00 00 00 00 00 00 00  |................|
00002cd0  00 00 00 00 d0 fa ad de  7b 00 00 00 00 00 00 00  |........{.......|
```

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 gdb and other utilites on it
- 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
- Critical: Ensure you have proper ulimits set before starting the process:
  - http://publib.boulder.ibm.com/infocenter/java7sdk/v7r0/topic/com.ibm.java.lnx.70.doc/diag/problem_determination/linux_setup.html
  - ulimit -c unlimited; ulimit -f unlimited
    - This is normally done in the node agent's startNode.sh script
- 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

Total Virtual Memory Usage in a Core

- Linux core dumps are ELF-structured files (just like an executable).
- When you use IBM Java's mechanisms to produce a core dump, it will indirectly use the kernel:
  - Linux does not provide an operating system API for generating a system dump from a running process. The [IBM] JVM produces system dumps on Linux by using the fork() API to start an identical process to the parent JVM process. The JVM then generates a SIGSEGV signal in the child process. The SIGSEGV signal causes Linux to create a system dump for the child process. The parent JVM processes and renames the system dump, as required, by the -Xdump options, and might add additional data into the dump file. The system dump for the child process contains an exact copy of the memory areas used in the parent. The SDK dump viewer can obtain information about the Java threads, classes, and heap from the system dump. However, the dump viewer, and other system dump debuggers [such as gdb] show only the single native thread that was running in the child process.
- IBM Java then may look append some information to the core dump that wouldn't otherwise be there to enhance core dump information.
- -Xdump “request” settings improve the quality of a dump produced by IBM Java. Recommended:
  - exclusive: quiesces Java threads for the duration of the dump
  - serial: Makes sure other dump agents aren't running concurrently
- The GDB gcore command (gcore $PID) has its own algorithm to attach to a process and create a core dump and this has slightly less than the kernel mechanism. The differences are not usually important.
- With kill -6 or kill -11, the process will be killed and the IBM Java dump mechanism is not triggered.

Getting a System Dump

- Wsadmin (WAS >= 7):
  AdminControl.invoke(AdminControl.completeObjectName("type=JVM,process=server1,*"), "generateSystemDump")
- Programmatically with com.ibm.jvm.Dump.SystemDump() (for example, in a JSP)
- IBM Health Center can acquire a system dump
- The IBM Java Dump agent “system” can take a system dump on various events. See Table 2 in Debugging from Dumps. For example, this will create a core dump when the Example.bad method throws a NullPointerException:
  -Xdump:system:events=throw,filter=java/lang/NullPointerException#com/Example.bad
- The IBM Java trace engine allows system 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}
- Use IBM Runtime Diagnostic Code Injection for Java (Java Surgery). For example: java -jar surgery.jar -pid 16715 -command SystemDump
- 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

System Dump Performance

- System dumps are quite large and can take 30 seconds or more to produce.
- The best way to improve system dump performance is to increase the available operating system file cache.
- In this way, the core will be written directly to RAM, the process will continue, and then the core will be asynchronously written to disk.
- Unfortunately, there is no easy way to “control” the amount of file cache available or what is written to it, so the best way is simply to have a lot of available RAM.
  - This can be somewhat tweaked with /proc/sys/vm/swappiness
    - Value of 100 means aggressively swap program pages (i.e. increase file cache)
    - Value of 0 means swap program pages if necessary (i.e. potentially decrease file cache)
    - Default 60
    - http://man7.org/linux/man-pages/man5/proc.5.html
- Disk speed is another important factor and it is recommended to have a dedicated filesystem for dump artifacts, both to allow for adding a fast disk, and also to reduce any potential issues of running out of space and affecting the applications.
  - When using the IBM Java dump engine to produce system dumps, you can control where they go with the generic JVM argument:
    - -Xdump:system:defaults:file=/somepath/core.%Y%m%d.%H%M%S.%pid.%seq.dmp

Linux core dump contents

- You can control what parts of the virtual address space the kernel dumps:
  - When a process is dumped, all anonymous memory is written to a core file as long as the size of the core file isn't limited. But sometimes we don't want to dump some memory segments, for example, huge shared memory. Conversely, sometimes we want to save file-backed memory segments into a core file, not only the individual files. /proc/PID/coredump_filter allows you to customize which memory segments will be dumped when the PID process is dumped. coredump_filter is a bitmask of memory types. If a bit of the bitmask is set, memory segments of the corresponding memory type are dumped, otherwise they are not dumped. The following 7 memory types are supported:
    - - (bit 0) anonymous private memory
    - - (bit 1) anonymous shared memory
    - - (bit 2) file-backed private memory
    - - (bit 3) file-backed shared memory
    - - (bit 4) ELF header pages in file-backed private memory areas (it is effective only if the bit 2 is cleared)
    - - (bit 5) hugetlb private memory
    - - (bit 6) hugetlb shared memory
  - https://www.kernel.org/doc/Documentation/filesystems/proc.txt
- Unfortunately, even with all bits set (0x7f), we still don't get the whole picture.
- GDB “info files” and search for the core dump section entries. Also: readelf and IDDE !info mmap
  - https://www.ibm.com/developerworks/community/blogs/kevgrig/entry/linux_understanding_total_virtual_memory_usage_from_a_core_dump_part_31?lang=en

Linux core dump file name pattern

- Linux 2.5 provides /proc/sys/kernel/core_pattern which controls the name of a kernel-produced core
  - Defaults to “core” in the current working directory
  - May include /'s which allow directing the core to a different directory
  - Some substitutions (max 64/128 characters)
    - %p PID of dumped process
    - %u (numeric) real UID of dumped process
    - %s number of signal causing dump
    - %t time of dump, expressed as seconds since the Epoch
    - %h hostname (same as nodename returned by uname(2))
    - %e executable filename (without path prefix)
    - %E pathname of executable, with slashes ('/') replaced by exclamation marks ('!').
    - %c core file size soft resource limit of crashing process
    - If no %p and core_uses_pid != 0, then .$PID added
- If the dump is produced by IBM Java, then it will take core_pattern into account and try to rename as per -Xdump settings (unless %t is specified in core_pattern)

gcore

- If you need to use gcore, then make sure to gather /proc/PID/maps at the same time. Here's a script:

```
#!/bin/sh
# This script automates taking a core dump using gcore.
# It also updates coredump_filter to maximize core dump contents.
# Usage: ./ibmgcore.sh PID [SEQ] [PREFIX]
#        PID - The process ID. You must have permissions (owner or sudo/root).
#        SEQ - Optional sequence number. Defaults to 1.
#        PREFIX - Optional prefix (e.g. directory and file name). Defaults to ./
PID=$1
SEQ=$2
PREFIX=$3
if [ -z "$PREFIX" ]; then
  PREFIX="./"
fi
if [ -z "$SEQ" ]; then
  SEQ=1
fi
DT=`date +%Y%m%d.%H%M%S`
LOG="${PREFIX}core.${DT}.$PID.000$SEQ.dmp.log.txt"
COREFILE="${PREFIX}core.${DT}.$PID.000$SEQ.dmp"
echo 0x7f > /proc/$PID/coredump_filter
date > ${LOG}; echo $PID >> ${LOG} 2>&1; cat /proc/$PID/coredump_filter >> ${LOG} 2>&1; echo "maps" >> ${LOG} 2>&1; cat /proc/$PID/maps >> ${LOG} 2>&1; echo
"smaps" >> ${LOG} 2>&1; cat /proc/$PID/smaps >> ${LOG} 2>&1; echo "limits" >> ${LOG} 2>&1; cat /proc/$PID/limits >> ${LOG} 2>&1; echo "gcore start" >> ${LOG}
2>&1; date >> ${LOG}; echo "status" >> ${LOG} 2>&1; cat /proc/$PID/status >> ${LOG} 2>&1; echo "pmap" >> ${LOG} 2>&1; pmap -x $PID >> ${LOG} 2>&1; echo
"meminfo" >> ${LOG} 2>&1; cat /proc/meminfo >> ${LOG} 2>&1; echo "fds" >> ${LOG} 2>&1; ls -l /proc/$PID/fd >> ${LOG} 2>&1; date >> ${LOG};
gcore -o $COREFILE $PID >> ${LOG} 2>&1
echo "gcore finish" >> ${LOG} 2>&1
date >> ${LOG}
echo "Gcore complete. Now renaming. This may take a few moments, but your process has now continued running."
# gcore adds the PID to the end of the file, so just remove that
mv $COREFILE.$PID $COREFILE
date >> ${LOG}
echo "Completely finished." >> ${LOG} 2>&1
```

gdb

- gdb is the native Linux debugger. First, extract the jextract zip to a directory.
- If jextract does not gather all used libraries, try libsgrabber.sh (essentially runs gdb “info shared”):
  - http://www-01.ibm.com/support/docview.wss?uid=swg21104706&aid=3
- You may also need the debuginfo packages for every library (for example, accurate native stacks)
  - For example, Fedora and RedHat do not ship binaries unstripped, but instead they separate the symbols into matching debuginfo packages which you have to download separately.
  - For RHEL, the debuginfo repository is not available by default. To add this repository, login to RHN to see the topic how to download debuginfo packages like kernel-debuginfo ?
    - https://access.redhat.com/knowledge/solutions/9907
  - Once you've got the debuginfo repository, how do you know which debuginfo packages to install?
  - GDB has will tell you what you need and the command that you need to run. For example:
    - $ gdb /usr/bin/java core.9501...
      Core was generated by `java HelloWorld'.
      Missing separate debuginfos, use: debuginfo-install libgcc-4.4.6-4.el6.x86_64 numactl-2.0.7-3.el6.x86_64
  - debuginfo-install will indirectly call yum, enable any debuginfo repositories, and install the specific packages that are needed.
- Pass the java process and the core file
  - gdb ./usr/IBM/WebSphereAppServer/java/bin/java core.20121014.210909.5785.0001.dmp

gdb

- When loading from another machine:
  - Run gdb without any parameters
    (gdb) set solib-absolute-prefix ./
    (gdb) set solib-search-path .
    (gdb) file ./path_to_java
    (gdb) core-file ./path_to_core
- Useful commands:
  - Print the current thread's stack: bt
  - List all threads: info threads
  - Switch to a different thread: thread N
  - Print stacks for all threads: thread apply all bt
  - GDB Batch: gdb --batch --quiet -ex "thread apply all bt" -ex "quit" $EXE $CORE

gdb

- Useful commands (continued)
  - Print memory in hex: x/Nx 0xabc... where N is the number of bytes
  - Print register: p $rax
  - Print current instruction register: x/i $pc
  - Disassemble function at address: disas 0xabc..
  - Print structure: ptype struct malloc_state
  - Print output to a file (gdb.txt): set logging on
  - Print data type of a variable: ptype var
  - Print symbol information: info symbol 0x...
- Other:
  - Check if a library is stripped: file $LIBRARY
    - Look for “stripped” or “not stripped” in the output

gdb

- How much is malloc'ed?
  - Add mp_.mmapped_mem plus system_mem for each arena starting at main_arena and following the next pointer until next==&main_arena
  - (gdb) p mp_.mmapped_mem
  - $1 = 0
  - (gdb) p &main_arena
  - $2 = (struct malloc_state *) 0x3c95b8ee80
  - (gdb) p main_arena.system_mem
  - $3 = 413696
  - (gdb) p main_arena.next
  - $4 = (struct malloc_state *) 0x3c95b8ee80
- Print system page size: getconf PAGESIZE

Linux native memory tracker tools

- The state of the art of Linux native memory tracking is poor:
  - “Currently debugging native-memory leaks on Linux with the freely available tools is more challenging than doing the same on Windows. Whereas UMDH allows native leaks on Windows to be debugged *in situ*, on Linux you will probably need to do some traditional debugging rather than rely on a tool to solve the problem for you.”
  - http://www.ibm.com/developerworks/library/j-nativememory-linux/

valgrind

- Valgrind can emulate the virtual memory system and allows tracking memory leaks.
- The program may run up to 10 to 30 times slower, so it's not practical in production.
- valgrind --trace-children=yes --leak-check=full java …
- Example output:
  - ==20494== 8,192 bytes in 8 blocks are possibly lost in loss record 36 of 45
    ==20494== at 0x4024AB8: malloc (vg_replace_malloc.c:207)
    ==20494== by 0x460E49D:
    Java_com_ibm_jtc_demos_LeakyJNIApp_nativeMethod

LinuxNativeTracker

- You may request a special debug tool called the LinuxNativeTracker from the IBM Java team through a PMR which overrides the default malloc implementation to track native memory leaks.
- Tracks stacks of unfreed allocations:
  - Stack 0
    0: /hosting/products/WebSphereU01/java/bin/java(0x8048000
).realpath+0x3ba (0x80498ae)
    1: /lib/libc.so.6(0xbb5000).__libc_start_main+0xdc (0xbcae9c)
    2: /hosting/products/WebSphereU01/java/bin/java(0x8048000
).read+0x5d (0x8049581)

mtrace

- Mtrace is a feature of glibc to hook into malloc/free calls:
  - http://www.gnu.org/software/libc/manual/html_node/Tracing-malloc.html#Tracing-malloc
- This requires C code which actually calls mtrace() at runtime
  - Also requires MALLOC_TRACE envar which points to trace file
  - http://www.gnu.org/software/libc/manual/html_node/Tips-for-the-Memory-Debugger.html#Tips-for-the-Memory-Debugger
- The output can be summarized using the mtrace user program:
  - http://www.gnu.org/software/libc/manual/html_node/Interpreting-the-traces.html#Interpreting-the-traces
- Not really practical for real world applications – not only does it dump every alloc and free, but it grabs a global lock each time, and it doesn't give stack traces.

Strace

- 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:
    - strace -f -tt -e trace=mmap,munmap -o outputfile.txt java ...
- You could also use this to dynamically attach to a process to watch mallocs and frees
  - strace -f -tt -e trace=brk,mmap -o outputfile.txt -p $PID

Ltrace

- Ltrace is similar to strace but can trace library calls (such as malloc and free)
  - ltrace -f -tt -e malloc,free -o outputfile.txt -p $PID

SystemTap

- Similar to DTrace. Must be installed (systemtap), requires root privilege, and requires debuginfo packages.
- Example malloc.stp file that prints every malloc stack:
  - Maybe not a practical example – should track mallocs and frees
  - ```
    probe process("/lib64/libc.so.6").function("malloc") {
      if (target() == pid()) {
        if($bytes > 1) {
          print_ustack(ubacktrace())
          printf("malloc=%d\n", $bytes)
        }
      }
    }
    ```
- Running it: stap -c "java ..." malloc.stp
- Finding the right module: $ ldd java
  libc.so.6 => /lib64/libc.so.6 (0x0000003c95800000)
- Find symbols/parameters by looking in /usr/src/debug
    ```

Conclusion

- Native memory issues are notoriously difficult. Always monitor paging and process resident and virtual size over time
- Use periodic javacores to track IBM Java native memory usage, particularly the number of classes/classloaders, number of threads, and JVM memory. Much more detail with WAS >= 8.
- If the native memory usage of concern is outside IBM Java, gather periodic data using /proc/$PID/smaps to help isolate the issue.
- Key things to watch for:
  - Whatever the Java heap takes (-Xmx), native mallocs don't have
  - Ulimits (-n, -u, -v, -d)
  - -Xnoclassgc, large -Xss, -Xrun/-agentlib/-javaagent (native JVMTI agent), -Xscmx
  - 64-bit with compressed references, the preferred heap base may limit native allocations
  - Upgrade native libraries such as type 2 database drivers, monitoring products, etc.
  - With generational GC, monitor icebergs (e.g. DirectByteBuffers)
- If all else fails:
  - Try using SystemTap or valgrind
  - If those fail, open a PMR and request the LinuxNativeTracker

Credits

- I would like to thank the following people for input on this presentation:
  - Neil Masson
  - Andrew Hall

Appendix

Debug kernel memory without symbols

- If you do not have a vmlinux file for your kernel with symbols (e.g. kernel debuginfo):
  - $ mkdir /tmp/vmlinux/; cd /tmp/vmlinux/
    $ wget https://raw.github.com/torvalds/linux/master/scripts/extract-vmlinux
    $ chmod u+x extract-vmlinux
    $ ./extract-vmlinux /boot/vmlinuz-`uname -r` > vmlinux
    $ gdb vmlinux /proc/kcore
    (gdb) shell grep total_forks /proc/kallsyms
    ffffffff81e96c28 B total_forks
    (gdb) p *0xffffffff81e96c28
    $1 = 11266
    (gdb) shell ls
    extract-vmlinux vmlinux
    (gdb) p *0xffffffff81e96c28
    $2 = 11266
    (gdb) core-file /proc/kcore
    [New process 1]
    #0 0x0000000000000000 in ?? ()
    (gdb) p *0xffffffff81e96c28
    $3 = 11271
  - Notice that we need to reload /proc/kcore to get the new value of total_forks

Stack Memory

- The parent process must allocate the stack of the new thread and the stack will grown downward from the top end of this allocation [1]
- There is no fundamental requirement for where the threads' stacks live. In fact, they can even be in the data segment using malloc as Linus' clone example shows [3]
- Different threading libraries, but most programs will use the pthread library (part of glibc) [2]
- Not clearly documented, but the max stack size is fixed on thread creation (although the max can be dynamically changed at runtime using pthread_attr_setstacksize for future threads) [4]
- pthread_create will create the stack using mmap [5]:
  mem = mmap (NULL, size, prot, MAP_PRIVATE | MAP_ANONYMOUS | MAP_STACK, -1, 0);
- From the documentation of mmap [6], it doesn't seem that MAP_STACK is used. So it would seem that the thread stack is just simply an anonymous mmap anywhere in the virtual address space. Presumably, then, the stack region I mentioned in the presentation (at the top of the address space) is only the primordial stack of the primary thread in the process.

  - [1] http://man7.org/linux/man-pages/man2/clone.2.html
  - [2] http://man7.org/linux/man-pages/man7/pthreads.7.html
  - [3] http://tldp.org/FAQ/Threads-FAQ/clone.c
  - [4] http://www.redhat.com/archives/phil-list/2003-January/msg00045.html
  - [5] http://sourceware.org/git/?p=glibc.git;a=blob;f=nptl/allocatestack.c;h=831e98e4ce8ee608fbd08b26bb2162fcfc4a7229;hb=HEAD#l489
  - [6] http://man7.org/linux/man-pages/man2/mmap.2.html

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