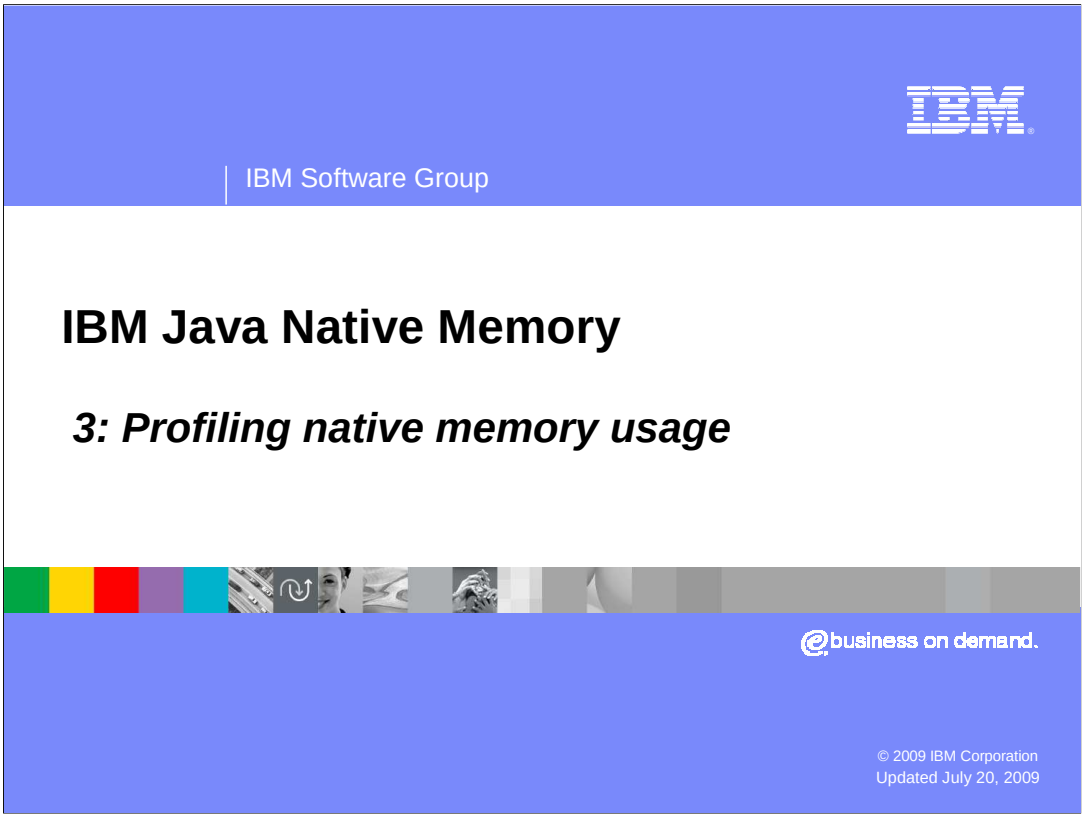


This is the third of three presentations on troubleshooting Java™ memory issues.

Profiling native heap exhaustion

- How to profile a native heap out-of-memory error

This presentation covers how to find the cause of an OutOfMemory error having determined that it is a problem with the native heap.

Native heap exhaustion

- Causes of native heap exhaustion:
  - Footprint:
    - Native heap space = Process address space – Java heap
    - If the Java heap is too large, the native heap is limited
  - Java heap leak:
    - Objects on the Java heap could have backing resources on the native heap
    - Small leak of objects of the Java heap could manifest itself as large native heap increase
  - “Standard” memory leak:
    - Memory is malloc() allocated with no corresponding free()

There are three main causes of native heap exhaustion.

You need to consider if it could be a footprint issue, which is when the native memory required is just bigger than is available. If this is the case, you have two options – either make the Java heap smaller so more space is available for the Native Heap, or, if you cannot make either smaller, you must remove data or move to 64-bit. If you move to 64-bit, there are more performance considerations and you will use more memory.

The second cause is the case of “iceberg objects” such as a Java heap leak. A Java heap leak from iceberg objects is when you have small leaking objects on the Java heap with links to resources on the native heap which could cause the native heap to run out of memory.

Finally, it could be a standard memory leak where malloc is called but, through coding error, there is no corresponding free. It is the same as on a Java heap where you are putting objects in collections and never taking them out.

JVM native heap analysis: Java 1.4.2

- Use of `dbgmalloc` trace option
- Traces calls to malloc() and free()
- Matches allocations without `free`, including `backtrace`

If you have had an OutOfMemory problem and you know it is native heap, how do you track where it is coming from?

The JVM gives you limited capabilities because it knows only what the JVM is doing. The JVM cannot know about native code or JNI code.

Monitoring the JVM can save some performance overhead because of its restricted information, so you might want to check the in-built data first.

To do this in Java 1.4.2, you have the dbgmalloc trace option which traces every malloc and free made by the JVM itself and matches allocations and frees. This means that the JVM can identify every allocation that never gets free.

The output includes backtrace, so you can tell which part of the code was requesting it.

Backtrace

- Gives a native stack trace for a tracepoint
- Can be triggered on any tracepoint
- Uses existing javacore code
- Not available on Windows® or Linux® IA64

Backtrace can produce native stack trace for any tracepoint (not just malloc and free) and shows you the last 4 to 6 frames, or however many you request, that got to that point in C code. This does not work for Java methods, but when running JNI code, the first C function has same name as the Java method that called it.

Calling Backtrace

Called from command line or properties file.

For example:

```
-Dibm.dg.trc.print=backtrace,hpi
-Dibm.dg.trc.maximal=backtrace,8,tpid(12345)
```

Example output:

```
13:21:32.432 0x81ad608 0200A0   > sysMonitorExit(), tid = 0x81ad7e0, mid = 0x8074018
13:21:32.433 0x81ad608 0002C1   - Backtrace:
                          0x403cdcdb sysMonitorExit /java/jre/bin/libhpi.so
                          0x4033cb0b xmIsThreadInterrupted
   /java/jre/bin/classic/libjvm.so
                          0x4033ba22 xmSetAsyncEventBehavior
   /java/jre/bin/classic/libjvm.so
                          0x4033bfe5 xmThreadFree /java/jre/bin/classic/libjvm.so
```


You can call Backtrace using the –Didm.dg.trc option, which is a standard prefix for trace at Java 1.4.2.

If using the Print option to go to the standarderror file, use the backtrace keyword. This will start doing Backtrace for any keyword that you specify.

To do this for memory, you use the backtrace,dbgmalloc command.

The output format is shown on this slide. In this case, it traces sysmonitorexit. When it left a lock, the thread was freed.

JVM Native Heap analysis: Java 5.0 and 6.0

- Using –memorycheck with the callsite option:
  - `-memorycheck:callsite=1000.`
  - Causes the program to print the callsite information every n number of allocations.
  - Frees are not factored into this.
  - This option is available only in 5.0 or higher VMs.
- Note: Memorycheck knows about the VMs allocations only; allocations from JNI code are not covered.
- If process address space increase is not accounted for, the problem is outside the JVM:
  - Use operating system based tools to profile all the allocations.

For Java 5 and Java 6, you can use the –memorycheck option to do the same sort of trace using the callsite option. It tracks every allocation and free to produce a report for a piece of code and checks if you have allocated memory that has not been subsequently freed.

When using the callsite option, the number parameter controls how often the report gets printed and it will report after a chosen number of allocations.1000 is reasonable for some applications, but you might need 100,000 for larger applications.

Memorycheck output

```
total alloc   | total freed   | delta alloc   | delta freed   | high water    | largest
 blocks| bytes | blocks| bytes | blocks| bytes | blocks| bytes | blocks| bytes | bytes | num   | callsite
-----------------------------------------------------------------------------------------------
      1     32       0       0       0       0       0       0       1     32     32       1 FinalizerSupport:584
      1  65560       0       0       0       0       0       0       1  65560  65560       1 FinalizeListManager:87
      3    768       0       0       0       0       0       0       3    768    256       1 allocateClassLoader
      3    134       0       0       0       0       0       0       3    134     49       2 classLoaderRegLibrary
      4  65104       2   32320       0       0       0       0       3  51560  28088       4 hashtable.c:595
      8  10280       2    5856       0       0       0       0       8  10280   4684       4 hashtable.c:159
    156 274405      71    4165       4     142      58    1912     139 272010   8232      26 tracewrappers.c:786
      9   1680       9    1680       0       0       0       0       3    560    512       3 loadLibraryWithPath
      2     30       2      30       0       0       0       0       2     30     17       2 trcengine.c:821
      1   4608       0       0       0       0       0       0       1   4608   4608       1 bcverify.c:570
      5  20480       5   20480       0       0       0       0       1   4096   4096       1 bcverify.c:2830
      1     28       0       0       0       0       0       0       1     28     28       1 MultiCodeCache.cpp:86
                3      36       1      12       1      12       0       0       2      24      12       1
MultiCodeCache.cpp:1992
      1   4096       0       0       0       0       0       0       1   4096   4096       1 MultiCodeCache.cpp:2262
     15   3780       2     914       0       0       0       0      13   3450   1112       3 vmprops.c:359
    164 7902132     157 7851956      39 2969472      92 5047644      65 2735816 106264     140 vmthread.c:605
      4     68       4      68       0       0       0       0       1     17     17       1 trcengine.c:1007
      1    256       1     256       0       0       0       0       1    256    256       1 trcengine.c:576
      1   1940       1    1940       0       0       0       0       1   1940   1940       1 trcengine.c:508
      1   1940       1    1940       0       0       0       0       1   1940   1940       1 trcengine.c:577
     17   4352      17    4352       0       0       0       0       1    256    256       1 zipsup.c:518
    140 571200       0       0       0       0       0       0     140 571200   4080       1 zipcache.c:435
    379 24256000     379 24256000      11  704000      11  704000       1  64000  64000       1 zipsup.c:1858
     17 542514      17  542514       0       0       0       0       1  65536  65536       2 zipsup.c:525
      1     94       1      94       0       0       0       0       1     94     94       1 zipsup.c:828
-----------------------------------------------------------------------------------------------
```

total alloc (blocks/bytes)	The total allocations for a particular callsite.
total freed (block/bytes)	The total of freed information for a particular callsite
delta alloc (blocks/bytes)	The allocation information for a particular callsite since the last printout
delta free (blocks/bytes)	The freed information for a particular callsite since the last printout
highwater (blocks/bytes)	The highest memory consumption by this callsite
largest (bytes/num)	num is the allocation ID of the largest allocation in bytes for this callsite
callsite	the sourcefile:linenumber (or assembly function name) of the callsite

Here is sample memorycheck output.

On the far right is callsite – in Java 5 and Java 6 this tells you file and line in the file which did the allocation.

The interesting values are those for Total Alloc Bytes, Total Freed Bytes and High Water Bytes.

High Water is the largest amount of memory allocated for this particular callsite. If this value is growing over time, that pool of memory is growing over time.

If the High Water value matches Total Alloc, you know you have never freed anything, which is a memory leak

If you are freeing data in the memory pool and it is growing over time, this could be legitimately growing to a larger size, but the cap for it could be bigger than you can allow, which might be a footprint issue.

This could mean that the code which should be freeing it cannot free the memory fast enough.

To summarize, if the High Water value is growing over time, the collection is getting bigger. If you are not freeing anything at all, it is a memory leak. And if you are not freeing enough, you need to find out why; and that could be a footprint issue.

OS-based native heap analysis

- The operating system based tools can profile all allocations:
  - Gives a more complete picture
  - Results in reduced performance
- The quality and the method that tools use vary between operating systems

The content in the VM can be deployed quickly because it is already built, but it does not measure anything. This is both an advantage and disadvantage. It is an advantage because there is less performance overhead, and a disadvantage because it could not find a problem at all because the problem is outside the VM code.

There are many ways of profiling all allocations, and this is mainly based on the operating system tools because it is the operating system doing the allocations. The quality of tools and method varies between platform.

AIX: MALLOCDEBUG trace

- AIX® provides a debug extension directly into the malloc subsystem:
  - Activates trace points in the allocation routines themselves
  - Records allocations that are not subsequently freed
- Activated using the following environment variable:
  - `MALLOCDEBUG=[trace,output:malloc.out,report_allocations,verbose,stack_depth:3]`
- Where the options have the following meanings:

trace	**Activate tracing of the malloc subsystem**
output	**Specify tracing to a log file, in this case malloc.out**
report_allocations	**Report outstanding allocations when the application stops**
stack_depth	**Report outstanding allocations with the specified stack depth, in this case 3**

- Notes:
  1. The stack depth of 3 provides only a limited stack trace. However, the use of larger numbers with a Java application might lead to crashes because the debug malloc facility does not understand the stack frames used for JIT-compiled code.
  2. Some of these options are available only in AIX 5.3

An AIX equivalent tool is mallocdebug trace.

This is an in-built debug extension in the malloc subsystem of the operating system.

Inside malloc and free routines there are tracepoints that record any allocation not subsequently freed and the stacktrace for that allocation.

You can activate it using an environment variable. For Java, it is trace,output:[File name to trace to],report_allocations (which generates the allocation) ,verbose (gives you more information and recommended to use it), stack_depth:3 (how much trace back that you want. In this case for Java got to set it to three because traceback in mallocdebug does not understand Java methods, so when the debug tool leaves C code and reach Java method it does not just not print them – it crashes, prevented by setting to three and means data you get often is not that good).

The table shows the meanings of the parameters in this command.

Some features are available only for AIX 5.3, such as the ability to send the result to a file before that it gets dumped to standard error. A restriction is that the tool writes only when the application finishes, so you either need to exit the application or kill it to get the data. The command is available on all AIX levels that are currently supported, but it is better at 5.3.

MALLOCDEBUG output

```
Allocation #1111: 0x3174BC68
        Allocation size: 0x400
        Allocation traceback:
        0xD02F4370  malloc
        0x31791258  Java_com_ibm_jtc_demos_LeakyJNIApp_nativeMethod
        0x301A477C  ??

    Allocation #1112: 0x3174C078
        Allocation size: 0x400
        Allocation traceback:
        0xD02F4370  malloc
        0x31791258  Java_com_ibm_jtc_demos_LeakyJNIApp_nativeMethod
        0x301A477C  ??

    Allocation #1113: 0x3174C488
        Allocation size: 0x400
        Allocation traceback:
        0xD02F4370  malloc
        0x31791258  Java_com_ibm_jtc_demos_LeakyJNIApp_nativeMethod
        0x301A477C  ??
```

- The JVM makes a number of allocations that are never freed:
  - The memory is required for the lifetime of the process, so process close down is used to release the memory
  - Ignoring the first 1000 entries will remove most of these entries
- Ability to combine allocations with the same stack trace not currently provided:
  - You can download a script to group all the same allocations together from Developer Works:

http://www.ibm.com/developerworks/aix/library/au-mallocdebug.html?ca=dgr-lnxw82MALLOCDEBUG

Here is the sample mallocdebug output.

It lists every outstanding allocation at the end, showing the location it was allocated, the size of it, and the traceback.

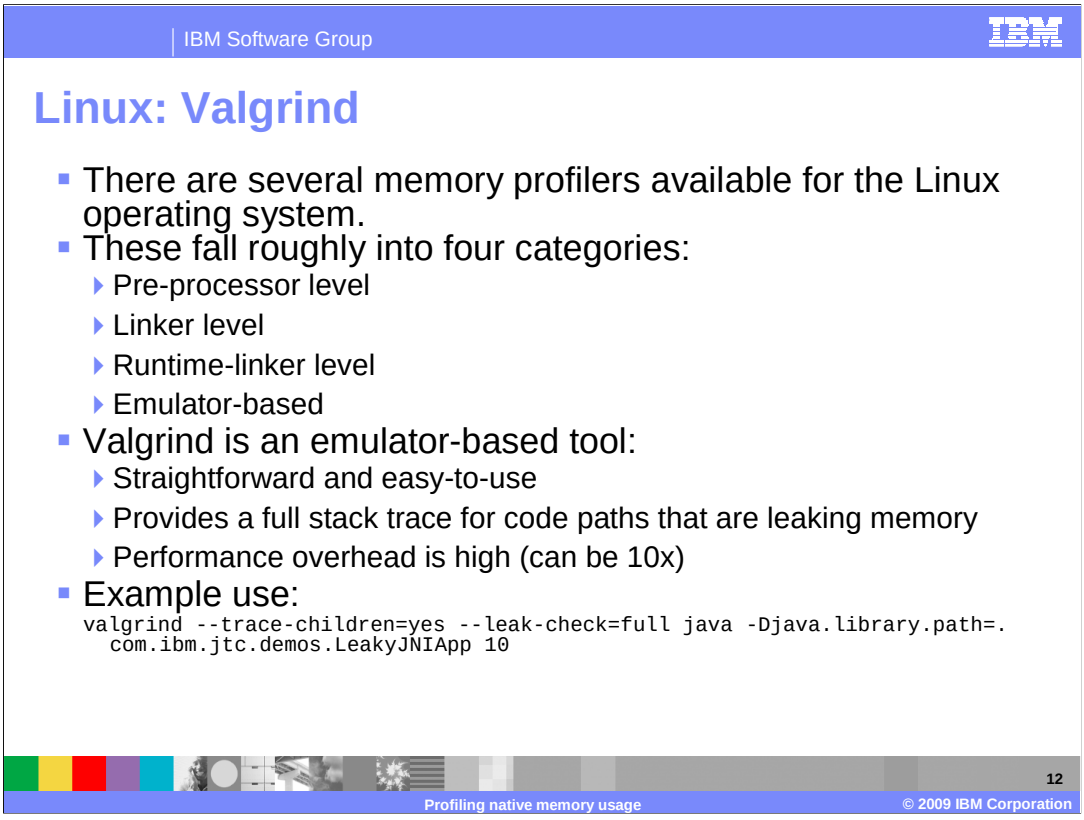
The JVM makes large numbers of allocations that it will never free, because those allocations will be used for the entirety of the process. The operating system automatically releases all the memory that has not been freed when the program exits anyway.

It is normal to see about a thousand entries from the JVM, so you can ignore anything with an allocation number under 1000. The later ones will be more relevant for diagnosing the leak.

The command currently lists all the individual allocations, even if they are the same.

The link at the bottom of the slide is to an article on Developer Works which contains a script that groups all the same allocations to give you the total size of the address.

Because the functionality is built into malloc, the profiling overhead is lower compared to other tools.



For Linux, there are many profilers available, which fall roughly into four categories. Some are at pre-processor level, so you must compile them in. Others are at link-time level, so you need to re-link your library, which means you need access to the source and object files.

The runtime-linker level tools link at runtime and do not require a recompile, but do need to use lD preload. Others are emulator based.

Valgrind is straightforward to use and user friendly, but has a higher overhead. It is good for running on a test system if you can reduce load, but can result in up to 10x performance reduction. Because it is an emulator, you run Valgrind and then Valgrind runs your application on top of it.

An example is at the bottom. Children=yes means that if your process forms a child process it will follow that as well, which is the way Java starts. The launcher starts another process so children=yes must be on. You must also do full checking using check=full and then add the command line that you are running, in this case: **`com.ibm.jtc.demos.LeakyJNIApp.`**

Valgrind output

```
==20494== 65,536 (63,488 direct, 2,048 indirect) bytes in 62 blocks are definitely
lost in loss record 42 of 45
==20494==    at 0x4024AB8: malloc (vg_replace_malloc.c:207)
==20494==    by 0x460E49D: Java_com_ibm_jtc_demos_LeakyJNIApp_nativeMethod (in
   /libleakyjni.so)
==20494==    by 0x535CF56: ???
==20494==    by 0x46423CB: gpProtectedRunCallInMethod (in /usr/jre/bin/libj9vm23.so)
==20494==    by 0x46441CF: signalProtectAndRunGlue (in /usr/jre/bin/libj9vm23.so)
==20494==    by 0x467E0D1: j9sig_protect (in /usr/jre/bin/libj9prt23.so)
==20494==    by 0x46425FD: gpProtectAndRun (in /usr/jre/bin/libj9vm23.so)
==20494==    by 0x4642A33: gpCheckCallin (in /usr/jre/bin/libj9vm23.so)
==20494==    by 0x464184C: callStaticVoidMethod (in /usr/jre/bin/libj9vm23.so)
==20494==    by 0x80499D3: main (in /usr/jre/bin/java)
==20494==
==20494== LEAK SUMMARY:
==20494==   definitely lost: 63,957 bytes in 69 blocks.
==20494==   indirectly lost: 2,168 bytes in 12 blocks.
==20494==     possibly lost: 8,600 bytes in 11 blocks.
==20494==   still reachable: 5,156,340 bytes in 980 blocks.
==20494==        suppressed: 0 bytes in 0 blocks.
==20494== Reachable blocks (those to which a pointer was found) are not shown.
==20494== To see them, rerun with: --leak-check=full --show-reachable=yes
```

- The second line of the stack shows that the memory was leaked by:
  - com.ibm.jtc.demos.LeakyJNIApp.nativeMethod() method

The output this tool provides is quite good. In this example, 69 blocks have definitely been lost. Because you have allocated memory but no longer have a reference to a pointer for it, you have no way of deleting it so you can definitively identify memory leaks.

It also identifies sections where it thinks memory can be lost; for example when a pointer might still be around but does not think you will use it. In this case, the value for definitely lost was 63,957, possibly lost was 8,600, and so on.

At the bottom, you can see it has condensed similar stack traces together and displays the totals.

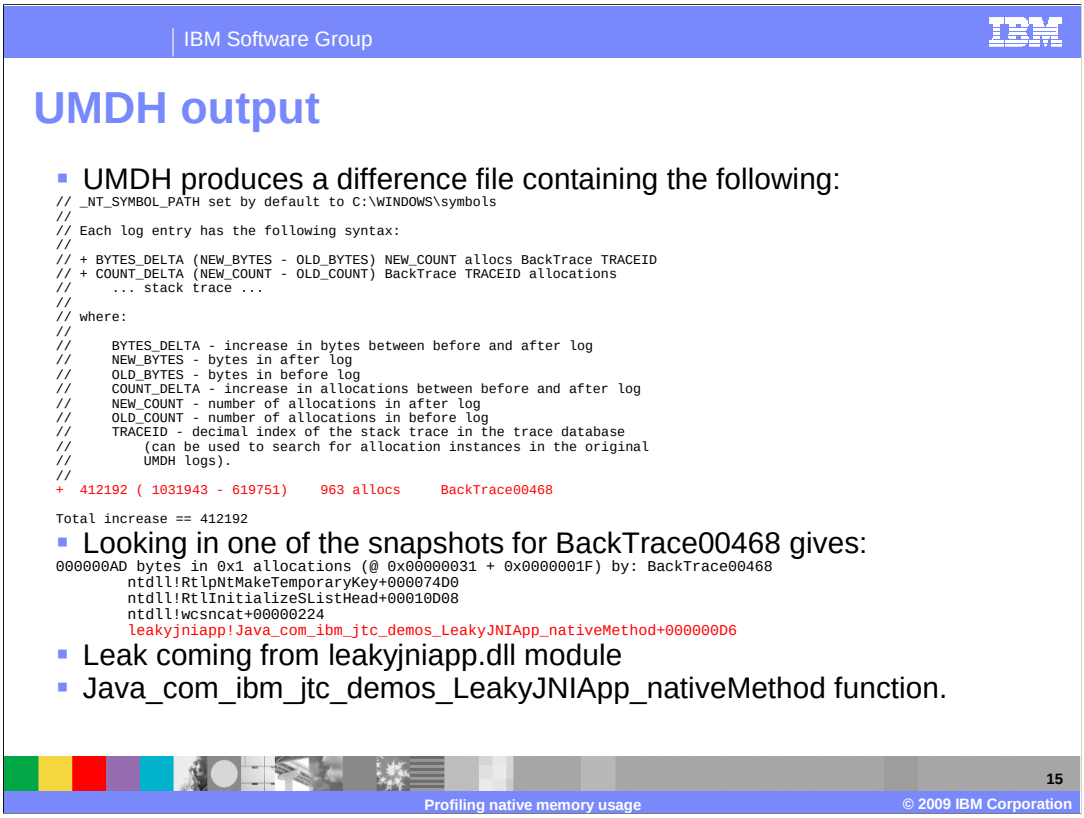
The line in red has identified the LeakyJNIApp_nativeMethod function, and you might remember that JNI methods in the first frame match the name for the Java method, so you can locate the LeakyJNIApp Java method with that function (nativemethod).

Windows: UMDH

- Microsoft provides the UMDH (User-Mode Dump Heap) tool for debugging native memory growth:
  - Records which code path allocated a particular area of memory
  - Provides a way to locate sections of code that allocate memory that is not freed later
- Usage of UMDH:
  - UMDH command takes a snapshot of the current native heap
    - Together with native stack traces of the code paths that allocated memory
  - Second snapshot is taken and stack traces for outstanding memory is produced
  - Detailed documentation on Microsoft Web site:
    - http://support.microsoft.com/kb/268343

Finally, consider Windows and the tool called UMDH (User-Mode Dump Heap tool), which is also a useful tool. After you start it, you do not have to run the entire application, because it tracks where memory is allocated and annotates this with where it was allocated from, no matter when it is started.

UMDH takes a snapshot of the heap, then takes a second snapshot and runs UMDH as a difference tool.



If you view the difference file, you will see output similar to this.

The bottom line, highlighted in red in the first sample output, is the important bit telling you that 412 KB was allocated that was not released by backtrace00468.

To find out about backtrace00468, go to either of the snapshots and find backtrace00468 to view the stack trace. This tells you that it was for LeakyJNIApp_nativeMethod again for this example, and you can check the owning module if the function name does not give it away. From this, you can see the function identified is nativeMethod, which is the nativeMethod method in Java.

Because you do not need to start UMDH when you start the application, you can run it only when you think there is something that is worth investigating, which means there is no performance overhead through most of the application.

You can also stop the tool without stopping the application.

Other native profiling tools

- There are also several native profiling products:
  - Rational® Purify® (all platforms)
  - Rootcause (AIX)
  - ZeroFault (AIX)
  - Glowcode (Windows)
- However, these all have a license fee associated with them.

There are several products available to do profiling. Purify should work on all platforms, Rootcause and zeroFault on AIX, and Glowcode on Windows. All of these require a license and license fee.

The tools previously mentioned in this presentation are all free ones: UMDH, Valgrind, and MallocDebug on AIX.

That concludes the set of presentations on the introduction to debugging OutOfMemory errors and Native Heap exhaustion.

Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send e-mail feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_3-ProfilingNativeMemoryUsage.ppt

This module is also available in PDF format at: ../3-ProfilingNativeMemoryUsage.pdf

You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, copyrights, and disclaimers

IBM, the IBM logo, ibm.com, and the following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX Purify Rational

If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of other IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml

Rational is a trademark of International Business Machines Corporation and Rational Software Corporation in the United States, Other Countries, or both.

Microsoft, Windows, and the Windows logo are registered trademarks of Microsoft Corporation in the United States, other countries, or both.

Java, JNI, JVM, and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements or changes in the products or programs described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (for example, IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products.

IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Note to U.S. Government Users - Documentation related to restricted rights-Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract and IBM Corp.