Technical white paper

Heap management in HP NonStop Server for Java 7

Table of contents

This paper provides an overview about Heap management in Java Hotspot Virtual Machine (JVM) in NonStop Server for Java 7 (NSJ7). Earlier versions of NSJ supported Java heap only up to 1.38 GB. This was due to NonStop Kernel (NSK) supporting only 32-bit processes.

A 32-bit process can have a maximum of 4 GB virtual address space. The Kernel uses large amount of virtual memory and leaves only 1.5 GB for applications to use. Out of 1.5 GB user space, JVM and other NSK products used by a Java process need memory and that left around 1.38 GB as a maximum available Java heap for Java applications. Due to this limitation, applications that needed large Java heap (>1.38 GB) could not be run on NonStop Server for Java.

As part of modernization of NSK, the kernel now supports 64-bit OSS applications. The 64-bit application can have very large heap, up to 512 GB. If NSJ uses 64-bit mode, then it can provide applications with a very large heap. NSJ7 is the first release that utilizes the 64-bit mode to provide large Java heap for Java applications. As JVM needs virtual memory, the theoretical limit of Java heap is around 484 GB. If application uses large heap, it is likely that the application pause time due to serial garbage collection (the only GC supported on earlier NSJ releases) may be high. So to enable applications to use large Java heap, better garbage collection, like Parallel and CMS GC, are needed in NSJ7. To avoid any degradation in performance because of using parallel or CMS GC algorithms, NSJ7 uses an innovative way to distribute the GC work to a group of processes called as GC processes. Each GC process shares the necessary virtual memory with the Java process for doing the GC. To enable ease of sharing of virtual memory, the virtual memory management is rewritten for NSJ7. This paper describes basics of the changes done in NSJ7 and provides tips and techniques on how to customize and optimize the heap usage. For details about various types of garbage collectors supported in NSJ7, please refer to the white paper "Garbage Collection in HP NonStop Server for Java 7."

This paper is organized as follows:

Section 2 provides an overview of heap implementation.

Section 3 provides an overview of how to get the virtual memory usage.

Section 4 provides an overview of swap space consideration.

Section 5 provides an overview of QIO segment consideration.

Section 6 provides references for further reading.

Section 7 provides an overview about memory leak detection.

Virtual memory management

Earlier NSJ versions used native C heap to allocate for JVM memory need including the Java heap. One exception to this is the Java code cache that is allocated as an executable flat segment. NSJ7 supports large Java heap. Large Java heap might have high pass time when serial GC is used. So NSJ7 needs parallel and CMS GC. NSJ7 uses an innovative method to perform parallel and CMS GC. It creates processes to perform parallel and CMS GC. The GC processes share the Java heap and JVM memory with the Java process. As NSK does not enable applications to share the native heap of one process with another process, JVM uses Guardian Segments for Java heap and JVM memory need. More than one process can share the segments and the only restriction they have is that the processes that share the segment need to be running on the same logical processor. NSK also provides OSS shared memory segments that can be shared across processes running on the same logical CPU. However the OSS shared memory segments are implemented using Guardian segments and hence NSJ7 JVM directly uses Guardian segments.

With Guardian segment replacing native heap segment, we need an efficient and fast memory management functions like malloc() and free(). For this NSJ7 uses NSK memory pool routines. The memory pool routines provide a mechanism to help an application developer manage extended data segments (flat or selectable segment). Instead of allocating one single flat segment, JVM allocates more than one segment for managing the virtual memory efficiently.

Allocating memory from segments reduces the memory available for Java heap in 32-bit JVM. This section describes briefly the various segments, why there is Java heap reduction on 32-bit JVM and how to get the details of the segments.

Java code cache segment

The Java code cache segment contains the dynamically generated code needed for Java application execution. The segment size is of 64 MB on 32-bit JVM and 256 MB on 64-bit JVM. The segment size is increased in 64-bit JVM because large applications might be loading lot of classes leading to higher demand in the code cache size. The segment identification number of the Java code cache segment is 1065. It must be noted that there is no change in the segment size or segment ID between earlier NSJ version and NSJ7 32-bit JDK version. The 64-bit JDK was not available in earlier NSJ version and hence, there is no data to compare for 64-bit JDK.

Java heap segment

JVM allocates a flat segment with segment ID 1067 for Java heap. The size of the segment is sum of the maximum Java Heap size specified by –Xmx command line option and the permanent generation size. If no maximum Java Heap size is specified, the default –Xmx value, 64 MB for 32-bit JDK and 1 GB for 64-bit JDK, is used. For 32-bit JDK, the maximum Java heap size is limited to around 1084 MB. For 64-bit JDK, it is a very large value, around 484 GB. Even though 64-bit JDK supports Java heap up to 484 GB, applications should not use very large heap. Applications must limit the memory usage to the size of the physical memory. Therefore on a 32 GB machine, a process must not use more than 32 GB of virtual memory. As the 32 GB virtual memory is applicable for process' entire address space, the available size for Java heap is less than 32 GB.

Typically JVM uses 2 GB for its needs. Also other products and native components might need memory. If we reserve 2 GB for such usage, the resultant maximum size of the Java heap is 28 GB (32 GB – 2 GB – 2 GB). The default size of permanent generation is 64 MB on 32-bit JVM and 80 MB on 64-bit JVM. If the application uses large virtual memory compared to the physical memory, memory thrashing occurs and it results in application performance degradation.

If JVM couldn't allocate Java heap segment due to unavailability of virtual memory, JVM prints an error message and aborts. It must be noted that if the initial Java heap size is less than the maximum Java heap size, it does not reduce the application memory foot print as NSJ allocates the segment with the maximum Java heap size. To reduce the memory foot print, the maximum Java heap size needs to be reduced.

JVM private segment

The Java and the GC processes allocate a private segment from which it allocates memory for private (non-shared) usage. The segment ID number of the JVM Private segment is 1068. As there is not much private data to be kept by each process, this segment size is 16 KB. This segment is the first allocated segment by JVM during the loading of libjvm.so file and hence, there should not be a case where segment allocation fails due to unavailability of virtual memory. If it happens, the process terminates.

JVM heap segment

For JVM memory need, JVM allocates one or more flat segments and use NSK pool routines to manage the memory. Memory that needs to be shared with GC processes is allocated from this segment. The number of segments and the size of the segments depend upon the JDK type, 32-bit or 64-bit. The first JVM heap segment is allocated during libjvm.so loading and subsequent segments (for 32-bit JDK) during application run if more virtual memory is needed by JVM. If any of the JVM heap segment allocation fails, the application is terminated. It should be noted that GC processes also can allocate memory into this segment and so the memory allocation and free from/to JVM Heap segment is controlled by a lock.

64-bit JDK

In 64-bit JDK, as virtual memory is plenty, JVM allocates one extensible segment with an initial size of 512 MB and a maximum size of 12 GB. When the allocated portion of the segment is full, it resizes the segment by extending the size by 512 MB or multiples of 512 MB. The hotspot compiler and the Garbage collector are the two main components that need large JVM heap space. Most of the applications should run fine with the default initial size of 512 MB for JVM heap. If applications really need large memory, it may encounter memory allocation failure and hence a large value, 12 GB is provided. As NSK commit swap space for only the allocated portion of the segment, having large maximum size for the JVM heap segment does not waste kernel managed swap (KMS) space. JVM uses segment ID 1069 for the JVM heap segment.

32-bit JDK

In 32-bit JDK, as virtual memory is limited, JVM allocates one segment of size 256 MB initially and uses it for its memory need. When this segment becomes full, it allocates an additional segment and adds it to the memory pool. It is possible that there may not be 256 MB of free virtual memory. In such situation, JVM searches for the largest free virtual memory block and uses half the size of the largest free block. The reason it uses half the size of the largest available free block is that other native components may need native heap and hence, JVM leaves half the available virtual memory for other components to use. As the maximum available user space size on NSK is around 1.5 GB, JVM can use a maximum of 6 segments and hence JVM can allocate a maximum of 6 segments. In normal applications JVM may need only one segment. This JVM segment reduces the available user space by 256 MB and hence the maximum Java heap on 32-bit JDK is reduced by 256 MB on NSJ7. JVM uses segment IDs 1069, 1070, 1071, 1072, 1073, and 1074 for JVM heap segment. The segments are allocated in the ascending order. So the first segment is allocated with segment ID 1069. The second segment is allocated with segment ID 1070, and so on.

Native heap

Most of the JVM memory need is from the shared segments. One exception to this is memory allocation for IPC messages and they are allocated from native heap. Also some of the memory allocation for JNI components can be from the native heap. The details about this are covered in later section.

Memory usage details

PSTATE, an NSK tool, can be used to display information about the native heap and the user segments allocated by the applications. Also nsjps tool can be utilized to display the total heap usage by a Java process and GC processes.

Segment information

The PSTATE output contains user segments allocated by a process. The JVM allocated user segment have segment IDs in the range 1065 to 1074. The number of segments in the output may differ between 32-bit and 64-bit JDK.

In 32-bit JDK

Following is the segment related information from the PSTATE output for a 32-bit Java process. In the output only one JVM heap segment with ID 1069 is shown. The text given in parenthesis is not part of PSTATE output. This text is added to indicate the type of the segment.

```
segment: Id: 1065                    (Java code cache segment)
Add: 0x0000000050000000
Size: 67108864 Max: 67108864
Flat, Unaliased, KMS Backed

segment: Id: 1067                    (Java heap segment)
Add: 0x0000000044000000
Size: 134299648 Max: 134299648
Flat, Unaliased, KMS Backed

segment: Id: 1068                    (JVM private segment)
Add: 0x0000000067FBC000
Size: 16384 Max: 16384
Flat, Unaliased, KMS Backed

segment: Id: 1069                    (JVM heap segment )
Add: 0x0000000054000000
Size: 268435456 Max: 268435456
Flat, Unaliased, KMS Backed
```

In 64-bit JDK

Following is the segment related information from the PSTATE output for a 64-bit Java process. The text given in parenthesis is not part of PSTATE output. This text is added to indicate the type of the segment.

```
segment: Id: 1065                    (Java code cache segment)
Add: 0x0000000054000000
Size: 268435456 Max: 268435456
Flat, Unaliased, KMS Backed

segment: Id: 1067                    (Java heap segment)
Add: 0x0000000700000000
Size: 6526418944 Max: 6526418944
Flat, Unaliased, KMS Backed

segment: Id: 1068                    (JVM private segment)
Add: 0x0000000067FBC000
Size: 16384 Max: 16384
Flat, Unaliased, KMS Backed

segment: Id: 1069                    (JVM heap segment)
Add: 0x0000000400000000
Size: 2147483648 Max: 12884901888
Flat, Unaliased, KMS Backed
```

Native heap usage

In earlier NSJ versions, almost all the virtual memory need of JVM was allocated from native heap. With NSJ7, the virtual memory is allocated from shared segments using NSK pool routines. This reduces the native heap usage in NSJ7.

32-bit JDK

In 32-bit JDK, the maximum size of native heap available is less compared to the earlier NSJ version. The maximum size of the native heap is reduced by the sum of Java heap segment, JVM heap segment, JVM private segment, and Java code cache segment. If user native component or other NonStop component allocates flat segment, then the size of the maximum native heap size is reduced accordingly. One example is the QIO segment. It normally starts at address 0x20000000 and it is of size 512 MB.

Following is the PSTATE output for native heap. It should be noted that the current size of the native heap is 16 KB. This is very less compared to earlier NSJ versions. This is because NSJ7 allocates segments for most of its memory need.

```
Heap          Origin    0x08001800            Size 16384,     Max 1610606592
```

64-bit JDK

In 64-bit JDK, the flat segments and the native heap segment uses different address range and so the native heap size is large. NSK has a default maximum size for the native heap and it is 12 GB. This heap is sufficient for other NSK components and user written native components. Following is the PSTATE output for the native heap for 64-bit process. The size is only 272 KB.

```
Heap (64-bit) Origin    0x0000000100000000,    Size 278528,    Max 12884901888
```

Changing native heap size

As mentioned earlier, the default maximum native heap size is 12 GB for 64-bit applications. If a 64-bit Java application needs more than 12 GB of native heap, then it can be done using "eld" as given below:

```
eld -change heap_max <heapMaxSize>    <JavaLauncher>
```

Where, `<heapmaxSize>` is the size of the maximum native heap in bytes

`<JavaLauncher>` is the Java launcher process

Example:

```
eld -change heap_max 17179869184 /usr/tandem/java/bin/oss64/java
```

The above command changes the native heap maximum size to 17 GB for processes started with `/usr/tandem/java/bin/oss64/java` binary file.

The nsjps Tool

The nsjps tool is a process status tool that lists and monitors the Java processes running on a NonStop system. Unlike PSTATE, nsjps tool can be used to get the total heap (JVM allocated segments + native heap) used by a Java process and/or used by the GC processes. This section briefly describes how to run nsjps tool to get the heap information.

Serial GC

If a Java process uses the serial GC, the heap reported by the nsjps tool is the sum of sizes of JVM allocated segments plus the native heap size. For the process whose output is given in section 3.1.1, the nsjps output with "-h" option is as given below:

Java command:

```
java <classFile>
```

NSJPS command:

```
nsjps -h
```

NSJPS output:

```
PID    Cmd    HEAP
989855801   java   470056960
```

Parallel/CMS GC

If a Java process uses the parallel or CMS GC, the heap reported by the nsjps tool can include the native heap of the GC processes also or it can list the information for the GC processes separately.

Consolidated output for Java and GC processes

By default the nsjps tool consolidates the native heap of the GC process with the heap information of the Java process and prints it. The GC processes share the flat segments allocated by the Java process. As the flat segments are accounted in the Java process itself, the GC process heap information contains only the amount of native heap used by it. The following is the sample output for the Java process with 4 parallel GC processes. It is assumed that Java is invoked on a four core CPU machine and hence there will be 4 GC processes.

Java command:

```
java -XX:+UseParallelGC <classFile>
```

nsjps command:

```
nsjps -h
```

nsjps output:

```
PID    Cmd    HEAP
1056964665 java    470450176
```

Compared to the serial GC, the heap value reported by nsjps is more when an application is run with Parallel/CMS GC. The heap size reported is sum of the heap used for Java process and the GC process.

Separate output for Java and GC processes

If nsjps tool is invoked with "-gc" option, it prints the heap information separately for each process in the Java process group. First the information about the Java process is printed and then information for each GC process is printed. As the GC processes share the flat segments allocated by the Java process, the GC process heap information contain only the native heap size. The following is the sample output for the Java process with 4 parallel GC processes. It is assumed that Java is invoked on a four core CPU machine and hence there will be 4 GC processes.

Java command:

```
java -XX:+UseParallelGC <classFile>
```

nsjps command:

```
nsjps -gc -h
```

nsjps output:

```
PID          Cmd       HEAP
1056964665   java      470056960
788529191    javagc    98304
436207691    javagc    98304
822083656    javagc    98304
268435533    javagc    98304
```

If the individual heap sizes are added, it equals the heap size printed when the consolidated heap size for the Java and GC processes are printed when "nsjps" is run without "-gc" option.

Swap space considerations

When a user starts a process, kernel managed swap facility (KMSF) provides the swap space that the process requires at process creation. A process can request additional swap space using the way of native heap expansion or user allocated segments. JVM does not use much of the native heap space and hence the initial allocation of swap space for native heap might be sufficient. However JVM allocates many flat segments for its operation. Among them the Java heap segment is the major consumer of KMS space for large Java heap. This also needs some consideration while configuring the KMS Files. JVM allocates a single segment for the Java heap. When Kernel reserves KMS space for a segment, it needs some extra space for managing the KMS space. So the KMS File should contain a free block of size little larger than the Java heap segment. For example, if an application is using 24 GB of Java heap, then KMS should contain at least one free block with

size little larger than 24 GB. For this reason, care should be taken while creating KMS space so that the KMS space is larger than the maximum Java heap space used by application in the system. One logical CPU can have more than one swap volumes to reduce the disc access. One block of KMS space can't span across volume and hence care should be taken while creating the KMS space.

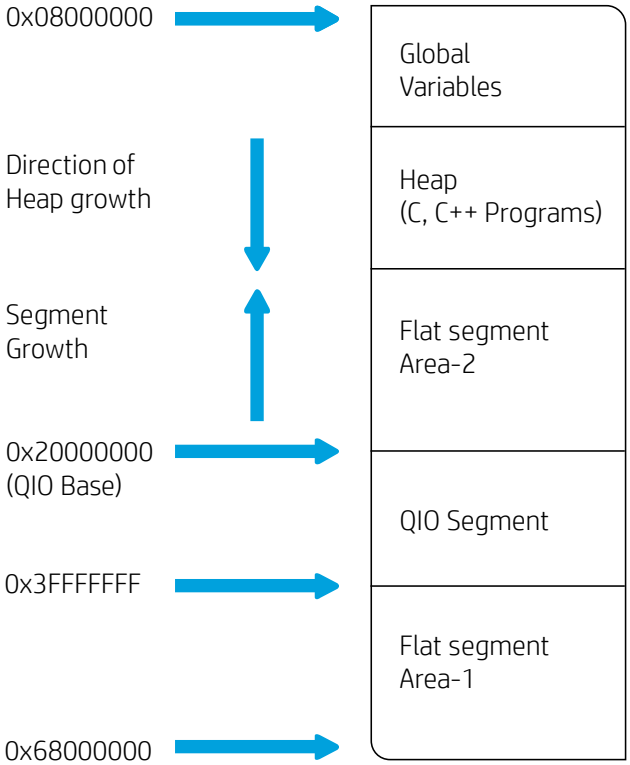
The KMS Files can be configured by NSKCOM utility. Please refer to "Kernel-Managed Swap Facility (KMSF) Manual" for more details.

QIO segment consideration

The QIO segment can be configured in the user space or in Kernel space. If it is configured in user space, the maximum and the default size of the QIO segment is 512 MB. If it is configured in the kernel space, the maximum size of the QIO segment is 512 MB and the default size is 256 MB. Customers configure the QIO segment in the user space if there is a need for lot of Kernel space. The default starting address of the QIO segment in user space is 0x20000000 and the ending address is 0x3FFFFFFF. This limits the maximum address to which the native heap can grow to 0x1FFFFFFF. There may be unused virtual memory starting at address 0x40000000 and the native heap cannot use this unused virtual memory as native heap cannot support split heap on NSK. Figure 1 illustrates this.

In earlier NSJ versions, 512 MB of native heap was insufficient for many Java applications and they used to encounter "Out of memory" error. The solution for the problem is to move the QIO segment to Kernel space. In NSJ7, the unused virtual memory beyond the QIO segment (Flat Segment Area-1) can be utilized by JVM for allocating segments (JVM Heap Segment, Java Heap Segment, JVM Private Segment). This reduces the native heap consumption by JVM, and the native heap is available for native components or other NonStop products.

Figure 1. Virtual Memory map of User Data Area



Memory allocation by native code

On other platforms, the GC threads have access to the entire virtual memory of the process. So any memory allocated by non-JDK component code, native component or library component, can be accessed by GC threads. In NSJ7, JVM tries to retain this functionality. However there is an issue if the code explicitly calls memory allocation to native heap using "malloc()" type functions. The malloc() set of functions allocate virtual memory only from native heap. This section describes how memory is allocated for C/C++ code.

If the JNI code and other library code are written in C, the virtual memory allocation will be from native heap. This is the case with earlier NSJ version also.

If the JNI code and other library code is written in C++, the virtual memory allocation by the application might be from native heap or from JVM heap segment depending whether the application uses "malloc()" or "new/new[]" and whether JVM library is loaded or not. If the C++ code uses "malloc()" for memory allocation, the memory will be allocated from native heap. If the C++ code uses "new" or "new[]" for memory allocation, then the memory is allocated from native heap or JVM heap depending whether JVM library, libjvm.so, is loaded or not. If the JVM library is already loaded, then virtual memory allocation for "new" and "new[]" is from JVM heap. If the JVM library is not already loaded, then the virtual memory allocation for "new" and "new[]" is from native heap. If "delete" or "delete[]" is called after JVM library is loaded to free memory for memory allocated before JVM library is loaded, JVM recognizes that the memory was allocated from native heap and handle free accordingly.

Whether the memory is allocated from native heap or JVM heap there is not much performance impact.

Memory leak detection

In earlier NSJ versions, memory leak can be detected by observing growth of native heap usage. The PSTATE tool provides the size of the native heap used. In NSJ7, the JVM allocates flat segments for its memory need and uses NSK pool routines to manage the pool as mentioned earlier. PSTATE does not report pool usage. The memory leak may be in the JVM or in the user application. This section describes how to identify the memory leak. Also the memory leak detection mechanism available in the native inspect (eInspect) is available only for JNI code written in C.

Memory leak in JNI code written in C

If the JNI code is written in C, then the memory allocated by the JNI code will be in native heap. So if the native heap utilization increases over a period of time, then it may indicate that there is memory leak.

Memory leak in JNI code written in C++

If the JNI code is written in C++, then the memory allocated using malloc will be from the native heap and "new" and "new[]" operators will be from JVM heap segment. So if either the native heap size increases or the JVM heap segment size increases, it indicates a possible memory leak.

Memory leak in 32-bit JDK

In 32-bit JDK, one or more (up to maximum of 6) JVM heap segments will be allocated. If the number of JVM heap segments increases, it indicates that memory utilization is increasing. This may indicate that there is a memory leak.

Memory leak in 64-bit JDK

In 64-bit JDK, one JVM heap segment with initial size of 512 MB and a maximum size of 12 GB is allocated. If the allocated size of the JVM heap segment increases, it indicates that there is a possibility of memory leak.

Memory corruption

In earlier NSJ versions, memory for JVM was allocated in the native heap. So if native code allocates memory using malloc() and corrupts the memory beyond the allocated region, it may corrupt the JVM data structures. With NSJ7, as memory is allocated from flat segments for JVM need, native code corrupting the native memory does not have any effect on JVM data structures. As mentioned earlier, memory allocated by "new" and "new[]" of C++ code is allocated in the JVM heap segment. So any memory corruption issue in C++ application might affect JVM data structures.

Conclusion

To enable more efficient garbage collection by the parallel and CMS garbage collectors, JVM allocates segments and manages them. Segmentation has a side effect of reducing the maximum Java heap for 32-bit JDK. So if an application needs more than 1084 MB of Java heap space, 64-bit JDK needs to be used. This paper described how the NonStop Server for Java 7 segments the available heap memory and uses it for different purposes. NSJ 7, in 64-bit installation, supports large heaps for applications–much larger than the 1.38 GB supported by NSJ 6. However for better performance, applications should try to use only as much Java heap as they require and in any case it should not be greater than the size of the physical memory of the CPU.

FAQs

This section provides answers for the frequently asked question.

1. What is the default Java heap size for 32-bit JDK?
   64 MB is the default Java heap size on 32-bit JDK.
2. What is the default Java heap size for 64-bit JDK?
   1 GB is the default Java heap size on 64-bit JDK.
3. What is the maximum Java heap size for 32-bit JDK?
   1084 MB is the maximum Java heap size for 32-bit JDK.
4. What is the maximum Java heap size for 64-bit JDK?
   484 GB is the maximum Java heap size for 64-bit JDK.
5. What are the segment IDs used by JVM?
   JVM uses segment IDs in the range 1065 to 1074.
6. My system has 16 GB of physical memory per CPU. Can I specify 32 GB for Java heap?
   It is recommended that one process does not exceed the maximum physical memory. So if a system has 16 GB of physical memory per CPU, it uses less than 14 GB for Java heap.
7. I have KMS space of 16 GB free space. Still my Java application with 16 GB Java heap fails. Why?
   The KMS does not have single free block large enough to satisfy the 16 GB swap space needed for the Java heap. The solution is to increase the size of the KMS file so that 16 GB of swap space can be accommodated in single file.
8. I have added a new KMS file of 16 GB size. Still my Java application with 16 GB Java heap fails. Why?
   The KMS has some overhead when a file is added for KMS space. So a 16 GB file contains little less than 16 GB and hence the allocation fails.
9. Does the QIO segment in user space affect the size of the Java heap available in 64-bit JDK?
   No. The QIO segment exists in the 32-bit addressable user space. The default starting address of QIO segment is 0x0000000020000000 which is the sign extension of 32-bit address, 0x20000000.
10. Does the QIO segment in user space affect the size of the Java heap available in 32-bit JDK?
    Yes, the QIO segment reduces the size of the Java heap available in 32-bit JDK. However as JVM allocates segments for its memory need, the available Java heap for the same QIO configuration is more in NSJ7 compared to earlier NSJ
11. How do I get the total heap usage including the JVM allocated segments?
    The nsjps tool reports the total heap usage including the JVM allocated segments.
12. Can I use memory leak detection feature available in eInspect to detect JVM memory leak?
    The memory leak detection feature available in eInspect is of no use for Java process as most of the virtual memory need is allocated from the flat segments using pool routines. Allocation to flat segments can't be tracked by eInspect and so it can't be used to detect memory leak in JVM.

References

1. Also see Garbage Collection in HP NonStop Server for Java white paper http://h20195.www2.hp.com/V2/GetDocument.aspx?docname=4AA4-7346ENW&cc=us&lc=en
2. For general background on Java garbage collection, refer to http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf.
3. Suggestions for tuning by adjusting heap and garbage collection parameters can be found at http://java.sun.com/javase/technologies/hotspot/gc/gc_tuning_6.html.
4. Details on using the HPjmeter tool can be found at hp.com/go/hpjmeter.
5. Additional information on garbage collection in NonStop systems can be found in the NonStop Server for Java 7.0 Programmer's Reference. It can be downloaded by going to hp.com/go/nonstop -> Click on "Products" -> Click on "Middleware and Java" -> Click on "User guides and technical documentation" and then click on "HP Integrity NonStop H-Series or HP Integrity NonStop J-Series."

Learn more at
hp.com/go/nonstop

Sign up for updates
hp.com/go/getupdated


Share with colleagues


Rate this document

4AA4-7347ENW, July 2013