Java using much more memory than heap size (or size correctly Docker memory limit)

Asked 1 year, 1 month ago Active 2 months ago Viewed 16k times

96

77

For my application, the memory used by the Java process is much more than the heap size.

The system where the containers are running starts to have memory problem because the container is taking much more memory than the heap size.

The heap size is set to 128 MB (`-Xmx128m -Xms128m`) while the container takes up to 1GB of memory. Under normal condition, it needs 500MB. If the docker container has a limit below (e.g. `mem_limit=mem_limit=400MB`) the process gets killed by the out of memory killer of the OS.

Could you explain why the Java process is using much more memory than the heap? How to size correctly the Docker memory limit? Is there a way to reduce the off-heap memory footprint of the Java process?

I gather some details about the issue using command from Native memory tracking in JVM.

From the host system, I get the memory used by the container.

```
$ docker stats --no-stream 9afcb62a26c8
CONTAINER ID        NAME
CPU %               MEM USAGE / LIMIT   MEM %               NET I/O             BLOCK I/O
PIDS
9afcb62a26c8        xx-xxxxxxxxxxxxx-
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx.0acbb46bb6fe3ae1b1c99aff3a6073bb7b7ecf85   0.93%
461MiB / 9.744GiB   4.62%               286MB / 7.92MB      157MB / 2.66GB      57
```

From inside the container, I get the memory used by the process.

```
$ ps -p 71 -o pcpu,rss,size,vsize
%CPU   RSS  SIZE    VSZ
11.2 486040 580860 3814600
```

```
$ jcmd 71 VM.native_memory
71:

Native Memory Tracking:

Total: reserved=1631932KB, committed=367400KB
-                 Java Heap (reserved=131072KB, committed=131072KB)
                            (mmap: reserved=131072KB, committed=131072KB)
```

```
-                     Class (reserved=1120142KB, committed=79830KB)
                            (classes #15267)
                            (  instance classes #14230, array classes #1037)
                            (malloc=1934KB #32977)
                            (mmap: reserved=1118208KB, committed=77896KB)
                            (  Metadata:   )
                            (    reserved=69632KB, committed=68272KB)
                            (    used=66725KB)
                            (    free=1547KB)
                            (    waste=0KB =0.00%)
                            (  Class space:)
                            (    reserved=1048576KB, committed=9624KB)
                            (    used=8939KB)
                            (    free=685KB)
                            (    waste=0KB =0.00%)

-                    Thread (reserved=24786KB, committed=5294KB)
                            (thread #56)
                            (stack: reserved=24500KB, committed=5008KB)
                            (malloc=198KB #293)
                            (arena=88KB #110)

-                      Code (reserved=250635KB, committed=45907KB)
                            (malloc=2947KB #13459)
                            (mmap: reserved=247688KB, committed=42960KB)

-                        GC (reserved=48091KB, committed=48091KB)
                            (malloc=10439KB #18634)
                            (mmap: reserved=37652KB, committed=37652KB)

-                  Compiler (reserved=358KB, committed=358KB)
                            (malloc=249KB #1450)
                            (arena=109KB #5)

-                  Internal (reserved=1165KB, committed=1165KB)
                            (malloc=1125KB #3363)
                            (mmap: reserved=40KB, committed=40KB)

-                     Other (reserved=16696KB, committed=16696KB)
                            (malloc=16696KB #35)

-                    Symbol (reserved=15277KB, committed=15277KB)
                            (malloc=13543KB #180850)
                            (arena=1734KB #1)

-    Native Memory Tracking (reserved=4436KB, committed=4436KB)
                            (malloc=378KB #5359)
                            (tracking overhead=4058KB)

-        Shared class space (reserved=17144KB, committed=17144KB)
                            (mmap: reserved=17144KB, committed=17144KB)

-               Arena Chunk (reserved=1850KB, committed=1850KB)
                            (malloc=1850KB)

-                   Logging (reserved=4KB, committed=4KB)
                            (malloc=4KB #179)

-                 Arguments (reserved=19KB, committed=19KB)
                            (malloc=19KB #512)

-                    Module (reserved=258KB, committed=258KB)
                            (malloc=258KB #2356)
```

```
$ cat /proc/71/smaps | grep Rss | cut -d: -f2 | tr -d " " | cut -f1 -dk | sort -n | awk '{
sum += $1 } END { print sum }'
491080
```

The application is a web server using Jetty/Jersey/CDI bundled inside a fat far of 36 MB.

The following version of OS and Java are used (inside the container). The Docker image is based on `openjdk:11-jre-slim` .

```
$ java -version
openjdk version "11" 2018-09-25
OpenJDK Runtime Environment (build 11+28-Debian-1)
OpenJDK 64-Bit Server VM (build 11+28-Debian-1, mixed mode, sharing)
$ uname -a
Linux service1 4.9.125-linuxkit #1 SMP Fri Sep 7 08:20:28 UTC 2018 x86_64 GNU/Linux
```

https://gist.github.com/prasanthj/48e7063cac88eb396bc9961fb3149b58

java linux docker memory jvm

edited Dec 5 '18 at 15:14

asked Nov 23 '18 at 17:52

Nicolas Henneaux
7,788 5 38 60

6 The heap is where objects are allocated, however the JVM has many other memory regions including shared libraries, direct memory buffers, thread stacks, GUI components, metaspace. You need to look at how large the JVM can be and make the limit high enough that you would rather the process die than use any more. – Peter Lawrey Nov 23 '18 at 19:01

2 It looks like the GC is using a lot of memory. You could try using the CMS collector instead. It looks like ~125 MB is used for metaspace + code, however without shrinking your code base, you are unlikely to be able to make that smaller. The committed space is close to your limit so it's not surprising it gets killed. – Peter LawreyNov 23 '18 at 19:08

where / how do you set the -Xms and -Xmx configuration? – Mick Dec 4 '18 at 14:05

In the Java command line `java -Xmx128m -Xms128m -jar the-bundle-package.jar` – Nicolas Henneaux Dec 4 '18 at 14:07

2 Note: Java 8 u212+ has now (Apr. 2019) a better Docker support when it comes to memory – VonC May 15 at 16:06

5 Answers

167

Virtual memory used by a Java process extends far beyond just Java Heap. You know, JVM include many subsytems: Garbage Collector, Class Loading, JIT compilers etc., and all these subsystems require certain amount of RAM to function.

JVM is not the only consumer of RAM. Native libraries (including standard Java Class Library) may also allocate native memory. And this won't be even visible to Native Memory Tracking. Java application itself can also use off-heap memory by means of direct ByteBuffers.

So what takes memory in a Java process?

+100

JVM parts (mostly shown by Native Memory Tracking)

1. Java Heap

   The most obvious part. This is where Java objects live. Heap takes up to `-Xmx` amount of memory.

2. Garbage Collector

   GC structures and algorithms require additional memory for heap management. These structures are Mark Bitmap, Mark Stack (for traversing object graph), Remembered Sets (for recording inter-region references) and others. Some of them are directly tunable, e.g. `-XX:MarkStackSizeMax`, others depend on heap layout, e.g. the larger are G1 regions (`-XX:G1HeapRegionSize`), the smaller are remembered sets.

   GC memory overhead varies between GC algorithms. `-XX:+UseSerialGC` and `-XX:+UseShenandoahGC` have the smallest overhead. G1 or CMS may easily use around 10% of total heap size.

3. Code Cache

   Contains dynamically generated code: JIT-compiled methods, interpreter and run-time stubs. It size is limited by `-XX:ReservedCodeCacheSize` (240M by default). Turn off `-XX:-TieredCompilation` to reduce the amount of compiled code and thus the Code Cache usage.

4. Compiler

   JIT compiler itself also requires memory to do its job. This can be reduced again by switching c Tiered Compilation or by reducing the number of compiler threads: `-XX:CICompilerCount`.

5. Class loading

   Class metadata (method bytecodes, symbols, constant pools, annotations etc.) is stored in off-heap area called Metaspace. The more classes are loaded - the more metaspace is used. Tota usage can be limited by `-XX:MaxMetaspaceSize` (unlimited by default) and `-XX:CompressedClassSpaceSize` (1G by default).

6. Symbol tables

   Two main hashtables of the JVM: the Symbol table contains names, signatures, identifiers etc. and the String table contains references to interned strings. If Native Memory Tracking indicate significant memory usage by a String table, it probably means the application excessively calls `String.intern`.

7. Threads

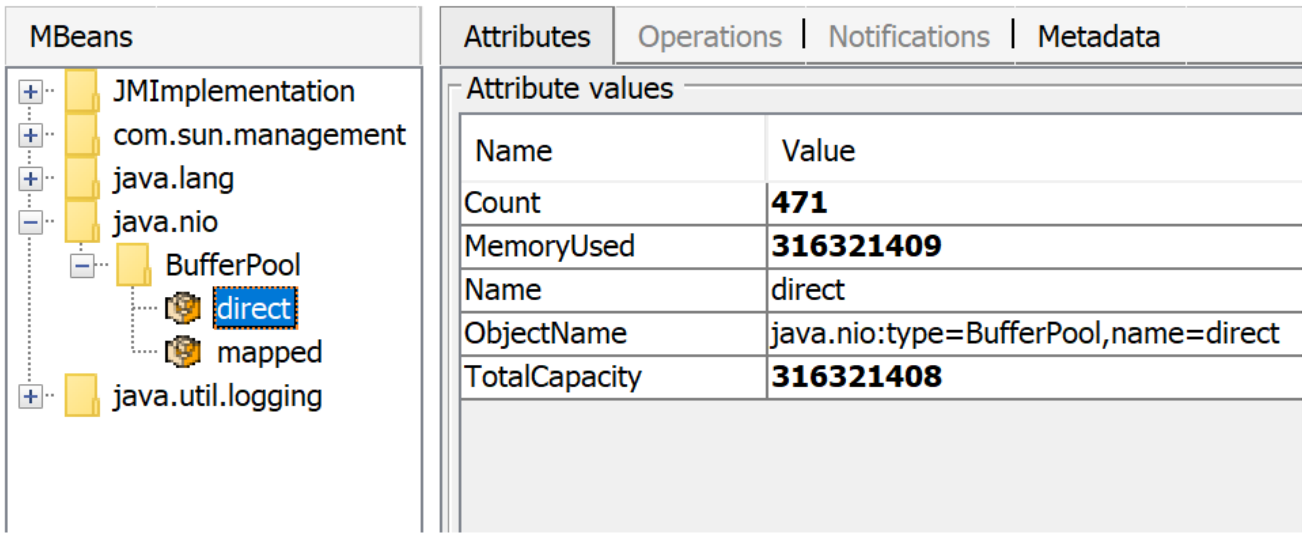
   Thread stacks are also responsible for taking RAM. The stack size is controlled by `-Xss`. The default is 1M per thread, but fortunately the things are not so bad. OS allocates memory pages lazily, i.e. on the first use, so the actual memory usage will be much lower (typically 80-200 KB per thread stack). I wrote a script to estimate how much of RSS belongs to Java thread stacks.

   There are other JVM parts that allocate native memory, but they do not usually play a big role i total memory consumption.

Direct buffers

An application may explicitly request off-heap memory by calling `ByteBuffer.allocateDirect` . The default off-heap limit is equal to `-Xmx` , but it can be overridden with `-XX:MaxDirectMemorySize` . Direct ByteBuffers are included in `Other` section of NMT output (or `Internal` before JDK 11).

The amount of used direct memory is visible through JMX, e.g. in JConsole or Java Mission Contro



Name	Value
Count	**471**
MemoryUsed	**316321409**
Name	direct
ObjectName	java.nio:type=BufferPool,name=direct
TotalCapacity	**316321408**

Besides direct ByteBuffers there can be `MappedByteBuffers` - the files mapped to virtual memory of a process. NMT does not track them, however, MappedByteBuffers can also take physical memory. And there is no a simple way to limit how much they can take. You can just see the actual usage by looking at process memory map: `pmap -x <pid>`

```
Address           Kbytes     RSS   Dirty Mode  Mapping
...
00007f2b3e557000   39592   32956       0 r--s- some-file-17405-Index.db
00007f2b40c01000   39600   33092       0 r--s- some-file-17404-Index.db
                           ^^^^^               ^^^^^^^^^^^^^^^^^^^^^^^^
```

Native libraries

JNI code loaded by `System.loadLibrary` can allocate as much off-heap memory as it wants with nc control from JVM side. This also concerns standard Java Class Library. In particular, unclosed Java resources may become a source of native memory leak. Typical examples are `ZipInputStream` or `DirectoryStream` .

JVMTI agents, in particular, `jdwp` debugging agent - can also cause excessive memory consumption.

This answer describes how to profile native memory allocations with async-profiler.

Allocator issues

A process typically requests native memory either directly from OS (by `mmap` system call) or by using `malloc` - standard libc allocator. In turn, `malloc` requests big chunks of memory from OS using `mmap` , and then manages these chunks according to its own allocation algorithm. The probler is - this algorithm can lead to fragmentation and excessive virtual memory usage.

`jemalloc` , an alternative allocator, often appears smarter than regular libc `malloc` , so switching to `jemalloc` may result in a smaller footprint for free.

Conclusion

There is no guaranteed way to estimate full memory usage of a Java process, because there are to many factors to consider.

```
Total memory = Heap + Code Cache + Metaspace + Symbol tables +
               Other JVM structures + Thread stacks +
               Direct buffers + Mapped files +
               Native Libraries + Malloc overhead + ...
```

It is possible to shrink or limit certain memory areas (like Code Cache) by JVM flags, but many others are out of JVM control at all.

One possible approach to setting Docker limits would be to watch the actual memory usage in a "normal" state of the process. There are tools and techniques for investigating issues with Java memory consumption: Native Memory Tracking, pmap, jemalloc, async-profiler.

Update

Here is a recording of my presentation Memory Footprint of a Java Process.

In this video, I discuss what may consume memory in a Java process, how to monitor and restrain the size of certain memory areas, and how to profile native memory leaks in a Java application.

edited Oct 19 at 16:13

answered Dec 5 '18 at 2:48

apangin
61.6k 9 130 164