

Virtual Memory Usage from Java under Linux, too much memory used

Asked 15 years, 10 months ago Modified 4 years, 8 months ago

Viewed 297k times



I have a problem with a Java application running under Linux.

306



When I launch the application, using the default maximum heap size (64 MB), I see using the `top`s application that 240 MB of virtual Memory are allocated to the application. This creates some issues with some other software on the computer, which is relatively resource-limited.



The reserved virtual memory will not be used anyway, as far as I understand, because once we reach the heap limit an `OutOfMemoryError` is thrown. I ran the same application under windows and I see that the Virtual Memory size and the Heap size are similar.

Is there anyway that I can configure the Virtual Memory in use for a Java process under Linux?

Edit 1: The problem is not the Heap. The problem is that if I set a Heap of 128 MB, for example, still Linux allocates 210 MB of Virtual Memory, which is not needed, ever.**

Edit 2: Using `ulimit -v` allows limiting the amount of virtual memory. If the size set is below 204 MB, then the application won't run even though it doesn't need 204 MB, only 64 MB. So I want to understand why Java requires so much virtual memory. Can this be changed?

Edit 3: There are several other applications running in the system, which is embedded. And the system does have a virtual memory limit (from comments, important detail).

java

linux

memory

virtual-memory

Share

Improve this question

Follow

edited Nov 22, 2017 at 11:45



Lii

12.1k ● 9 ● 68 ● 89

asked Feb 18, 2009 at 14:24



Mario Ortégón

18.9k ● 18 ● 72 ● 81

Why are you concerned with virtual memory usage? If you really want to be concerned, look at resident memory usage and read up on the following commands: `free`, `ps`, `top`.

– [basszero](#) Feb 18, 2009 at 15:10

2 There is several other applications running in the system, which is embedded. And the system does have a virtual memory limit. – [Mario Ortégón](#) Feb 18, 2009 at 15:12

ahhhh, devil is in the details – [basszero](#) Feb 18, 2009 at 15:55

Which implementation of Java are you using. IIRC, the bog standard (non-OpenJDK) free Sun JRE is not licensed for embedded use. – [Tom Hawtin - tackline](#) Feb 18, 2009 at 19:41

I think I miss-used the "embedded" part... it is memory limited and the hardware is customized, but it is still a standard computer – [Mario Ortégón](#) Feb 20, 2009 at 12:40

8 Answers

Sorted by:

Highest score (default)



725



+250



This has been a long-standing complaint with Java, but it's largely meaningless, and usually based on looking at the wrong information. The usual phrasing is something like "Hello World on Java takes 10 megabytes! Why does it need that?" Well, here's a way to make Hello World on a 64-bit JVM claim to take over 4 gigabytes ... at least by one form of measurement.

```
java -Xms1024m -Xmx4096m com.example.Hello
```

Different Ways to Measure Memory

On Linux, the [top](#) command gives you several different numbers for memory. Here's what it says about the Hello World example:

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM
TIME+	COMMAND								
2120	kgregory	20	0	4373m	15m	7152	S	0	0.2
0:00.10	java								

- VIRT is the virtual memory space: the sum of everything in the virtual memory map (see below). It is largely meaningless, except when it isn't (see below).
- RES is the resident set size: the number of pages that are currently resident in RAM. In almost all cases, this is the only number that you should use when saying "too big." But it's still not a very good number, especially when talking about Java.
- SHR is the amount of resident memory that is shared with other processes. For a Java process, this is typically limited to shared libraries and memory-mapped JARfiles. In this example, I only had one Java process running, so I suspect that the 7k is a result of libraries used by the OS.
- SWAP isn't turned on by default, and isn't shown here. It indicates the amount of virtual memory that is currently resident on disk, *whether or not it's actually in the swap space*. The OS is very good about keeping active pages in RAM, and the only cures for swapping are (1) buy more memory, or (2) reduce the number of processes, so it's best to ignore this number.

The situation for Windows Task Manager is a bit more complicated. Under Windows XP, there are "Memory Usage" and "Virtual Memory Size" columns, but the [official documentation](#) is silent on what they mean. Windows Vista and Windows 7 add more columns, and they're actually [documented](#). Of these, the "Working Set" measurement is the most useful; it roughly corresponds to the sum of RES and SHR on Linux.

Understanding the Virtual Memory Map

The virtual memory consumed by a process is the total of everything that's in the process memory map. This includes data (eg, the Java heap), but also all of the shared libraries and memory-mapped files used by the program. On Linux, you can use the [pmap](#) command to see all of the things mapped into the process space (from here on out I'm only going to refer to Linux, because it's what I use; I'm sure there are equivalent tools for Windows). Here's an excerpt from the memory map of the "Hello World" program; the entire memory map is over 100 lines long, and it's not unusual to have a thousand-line list.

```
0000000040000000      36K r-x--  
/usr/local/java/jdk-1.6-x64/bin/java  
0000000040108000       8K rwx--  
/usr/local/java/jdk-1.6-x64/bin/java  
0000000040eba000    676K rwx--    [ anon ]
```

```

000000006fae00000 21248K rwx-- [ anon ]
000000006fc2c0000 62720K rwx-- [ anon ]
00000000700000000 699072K rwx-- [ anon ]
0000000072aab0000 2097152K rwx-- [ anon ]
000000007aaab0000 349504K rwx-- [ anon ]
000000007c0000000 1048576K rwx-- [ anon ]
...
00007fa1ed00d000 1652K r-xs-
/usr/local/java/jdk-1.6-x64/jre/lib/rt.jar
...
00007fa1ed1d3000 1024K rwx-- [ anon ]
00007fa1ed2d3000 4K ----- [ anon ]
00007fa1ed2d4000 1024K rwx-- [ anon ]
00007fa1ed3d4000 4K ----- [ anon ]
...
00007fa1f20d3000 164K r-x--
/usr/local/java/jdk-1.6-
x64/jre/lib/amd64/libjava.so
00007fa1f20fc000 1020K -----
/usr/local/java/jdk-1.6-
x64/jre/lib/amd64/libjava.so
00007fa1f21fb000 28K rwx--
/usr/local/java/jdk-1.6-
x64/jre/lib/amd64/libjava.so
...
00007fa1f34aa000 1576K r-x-- /lib/x86_64-linux-
gnu/libc-2.13.so
00007fa1f3634000 2044K ----- /lib/x86_64-linux-
gnu/libc-2.13.so
00007fa1f3833000 16K r-x-- /lib/x86_64-linux-
gnu/libc-2.13.so
00007fa1f3837000 4K rwx-- /lib/x86_64-linux-

```

A quick explanation of the format: each row starts with the virtual memory address of the segment. This is followed by the segment size, permissions, and the source of the segment. This last item is either a file or "anon", which indicates a block of memory allocated via [mmap](#).

Starting from the top, we have

- The JVM loader (ie, the program that gets run when you type `java`). This is very small; all it does is load in the shared libraries where the real JVM code is stored.
- A bunch of anon blocks holding the Java heap and internal data. This is a Sun JVM, so the heap is broken into multiple generations, each of which is its own memory block. Note that the JVM allocates virtual memory space based on the `-Xmx` value; this allows it to have a contiguous heap. The `-Xms` value is used internally to say how much of the heap is "in use" when the program starts, and to trigger garbage collection as that limit is approached.
- A memory-mapped JARfile, in this case the file that holds the "JDK classes." When you memory-map a JAR, you can access the files within it very efficiently (versus reading it from the start each time). The Sun JVM will memory-map all JARs on the classpath; if your application code needs to access a JAR, you can also memory-map it.
- Per-thread data for two threads. The 1M block is the thread stack. I didn't have a good explanation for the 4k block, but @ericsoe identified it as a "guard block": it does not have read/write permissions, so will cause a segment fault if accessed, and the JVM catches that and translates it to a `StackOverflowError`. For a real app, you will see dozens if not hundreds of these entries repeated through the memory map.

- One of the shared libraries that holds the actual JVM code. There are several of these.
- The shared library for the C standard library. This is just one of many things that the JVM loads that are not strictly part of Java.

The shared libraries are particularly interesting: each shared library has at least two segments: a read-only segment containing the library code, and a read-write segment that contains global per-process data for the library (I don't know what the segment with no permissions is; I've only seen it on x64 Linux). The read-only portion of the library can be shared between all processes that use the library; for example, `libc` has 1.5M of virtual memory space that can be shared.

When is Virtual Memory Size Important?

The virtual memory map contains a lot of stuff. Some of it is read-only, some of it is shared, and some of it is allocated but never touched (eg, almost all of the 4Gb of heap in this example). But the operating system is smart enough to only load what it needs, so the virtual memory size is largely irrelevant.

Where virtual memory size is important is if you're running on a 32-bit operating system, where you can only allocate 2Gb (or, in some cases, 3Gb) of process address

space. In that case you're dealing with a scarce resource, and might have to make tradeoffs, such as reducing your heap size in order to memory-map a large file or create lots of threads.

But, given that 64-bit machines are ubiquitous, I don't think it will be long before Virtual Memory Size is a completely irrelevant statistic.

When is Resident Set Size Important?

Resident Set size is that portion of the virtual memory space that is actually in RAM. If your RSS grows to be a significant portion of your total physical memory, it might be time to start worrying. If your RSS grows to take up all your physical memory, and your system starts swapping, it's well past time to start worrying.

But RSS is also misleading, especially on a lightly loaded machine. The operating system doesn't expend a lot of effort to reclaiming the pages used by a process. There's little benefit to be gained by doing so, and the potential for an expensive page fault if the process touches the page in the future. As a result, the RSS statistic may include lots of pages that aren't in active use.

Bottom Line

Unless you're swapping, don't get overly concerned about what the various memory statistics are telling you. With the caveat that an ever-growing RSS may indicate some sort of memory leak.

With a Java program, it's far more important to pay attention to what's happening in the heap. The total amount of space consumed is important, and there are some steps that you can take to reduce that. More important is the amount of time that you spend in garbage collection, and which parts of the heap are getting collected.

Accessing the disk (ie, a database) is expensive, and memory is cheap. If you can trade one for the other, do so.

[Share](#) [Improve this answer](#)

[edited Apr 25, 2020 at 13:27](#)

[Follow](#)

answered Feb 18, 2009 at 15:17




[kdgregory](#)

39.6k ● 11 ● 82 ● 105

9 You should take into account that portions of memory which are currently swapped out are missing from the RES measure. So you might have a low RES value but only because the application was inactive and much of the heap was swapped out to disk. Java does a very bad job wrt to swap: On each full GC most of the heap is walked and copied so if much of your heap was in swap, the GC has to load it all back into main memory. – [jrudolph](#) May 20, 2010 at 8:11

1 Great answer kdggregory! I'm running in an embedded environment using a CF which has NO swap space. So based on your answer all of my VIRT, SWAP and nFLT values are from memory mapped files... which now makes sense to mew. Do you know if the SWAP value represent pages that have not yet been loaded into memory or pages that have been swapped out of memory, or both? How can we get an idea of possible thrashing (continuous map in then swap out)? – [Jeach](#) Sep 21, 2010 at 19:13

2 @Jeach - I was surprised that any swap was reported, so booted my "traveling Linux" (a thumb drive with Ubuntu 10.04 and no swap). When I enabled the "SWAP" column in *top*, I saw Eclipse had 509m. When I then looked at it with *pmap*, the total virtual space was 650m. So I suspect that the "SWAP" figure represents all on-disk pages, not just those that aren't in memory. – [kdgregory](#) Sep 26, 2010 at 13:32

2 As for your second question: if you're constantly reading pages from the flash card, your IO wait time (shown in the summary of *top* as "%wa") should be high. Beware, however, that this will be high for any activity, particularly writes (assuming that your program does any). – [kdgregory](#) Sep 26, 2010 at 13:33 

1 > The 1M block is a thread stack; I don't know what goes into the 4K block. The 4K block - which is marked as having neither read nor write permissions - is likely a guard block.

On stack overflow, this area is accessed, which triggers a fault, which the JVM can then handle by generating a Java StackOverflowException. This is way cheaper than checking the stack pointer at each method call. Guard areas with no permissions set can also be seen used in other contexts.

– [eriksoe](#) Apr 20, 2020 at 12:31



There is a known problem with Java and glibc >= 2.10 (includes Ubuntu >= 10.04, RHEL >= 6).

52



The cure is to set this env. variable:

```
export MALLOC_ARENA_MAX=4
```



If you are running Tomcat, you can add this to

```
TOMCAT_HOME/bin/setenv.sh
```

 file.

For Docker, add this to Dockerfile

```
ENV MALLOC_ARENA_MAX=4
```

There is an IBM article about setting

MALLOC_ARENA_MAX

https://www.ibm.com/developerworks/community/blogs/kevgrig/entry/linux_glibc_2_10_rhel_6_malloc_may_show_excessive_virtual_memory_usage?lang=en

[This blog post says](#)

resident memory has been known to creep in a manner similar to a memory leak or memory fragmentation.

There is also an open JDK bug [JDK-8193521 "glibc wastes memory with default configuration"](#)

search for `MALLOC_ARENA_MAX` on Google or SO for more references.

You might want to tune also other malloc options to optimize for low fragmentation of allocated memory:

```
# tune glibc memory allocation, optimize for low fragm
# limit the number of arenas
export MALLOC_ARENA_MAX=2
# disable dynamic mmap threshold, see M_MMAP_THRESHOLD
export MALLOC_MMAP_THRESHOLD_=131072
export MALLOC_TRIM_THRESHOLD_=131072
export MALLOC_TOP_PAD_=131072
export MALLOC_MMAP_MAX_=65536
```

Share Improve this answer

edited Sep 12, 2019 at 14:27

Follow


answered Mar 9, 2015 at 4:17




Lari Hotari

5,300 ● 1 ● 39 ● 45

This answer really helped me on a 64bit Ubuntu Server with a TomEE server that got a little to "mem-consuming". The link to IBM-article really is a profound explanation. Thanks again for this good hint! – [MWiesner](#) Jun 20, 2015 at 19:58

- 2 The JVM might leak native memory which leads to similar symptoms. See stackoverflow.com/a/35610063/166062 . Unclosed GZIPInputStream and GZIPOutputStream instances might be a source of the leak as well. – [Lari Hotari](#) Sep 16, 2016 at 6:25 
-

- 3 There is a JVM bug in Java 8, which results in unbounded native memory growth: bugs.java.com/bugdatabase/view_bug.do?bug_id=JDK-8164293 - If this is affecting you, using `MALLOC_ARENA_MAX` may slow your memory growth, but not solve the problem entirely. – [outofcoffee](#) Jan 19, 2017 at 22:51
-

@LariHotari really appreciate your effort for pointing out glibc and redhat version – [Sam](#) May 5, 2017 at 21:25 

- 2 Java 8u131 contains backported bugfix for the related JVM bug JDK-8164293 bugs.openjdk.java.net/browse/JDK-8178124 . – [Lari Hotari](#) May 31, 2017 at 12:47
-



11



The amount of memory allocated for the Java process is pretty much on-par with what I would expect. I've had similar problems running Java on embedded/memory limited systems. Running *any* application with arbitrary VM limits or on systems that don't have adequate amounts of swap tend to break. It seems to be the nature of many modern apps that aren't design for use on resource-limited systems.

You have a few more options you can try and limit your JVM's memory footprint. This might reduce the virtual memory footprint:

-XX:ReservedCodeCacheSize=32m Reserved code cache size (in bytes) - maximum code cache size. [Solaris 64-bit, amd64, and -server x86: 48m; in 1.5.0_06 and earlier, Solaris 64-bit and amd64: 1024m.]

-XX:MaxPermSize=64m Size of the Permanent Generation. [5.0 and newer: 64 bit VMs are scaled 30% larger; 1.4 amd64: 96m; 1.3.1 -client: 32m.]

Also, you also should set your -Xmx (max heap size) to a value as close as possible to the ***actual peak memory usage*** of your application. I believe the default behavior of the JVM is still to *double* the heap size each time it expands it up to the max. If you start with 32M heap and your app peaked to 65M, then the heap would end up growing 32M -> 64M -> 128M.

You might also try this to make the VM less aggressive about growing the heap:

-XX:MinHeapFreeRatio=40 Minimum percentage of heap free after GC to avoid expansion.

Also, from what I recall from experimenting with this a few years ago, the number of native libraries loaded had a huge impact on the minimum footprint. Loading `java.net.Socket` added more than 15M if I recall correctly (and I probably don't).

Share Improve this answer

answered Feb 18, 2009 at 16:21

Follow



[James Schek](#)

18k ● 7 ● 53 ● 64



The Sun JVM requires a lot of memory for HotSpot and it maps in the runtime libraries in shared memory.

8



If memory is an issue consider using another JVM suitable for embedding. IBM has j9, and there is the Open Source "jamvm" which uses GNU classpath libraries. Also Sun has the Squeak JVM running on the SunSPOTS so there are alternatives.



Share Improve this answer

answered Mar 3, 2009 at 17:14

Follow



[Thorbjørn Ravn Andersen](#)

75.3k ● 34 ● 199 ● 352

Is it an option to disable hot spot? – [Mario Ortégón](#) Mar 4, 2009 at 10:27

Perhaps. Check the command line options for the JVM you use. – [Thorbjørn Ravn Andersen](#) Mar 4, 2009 at 10:43



5

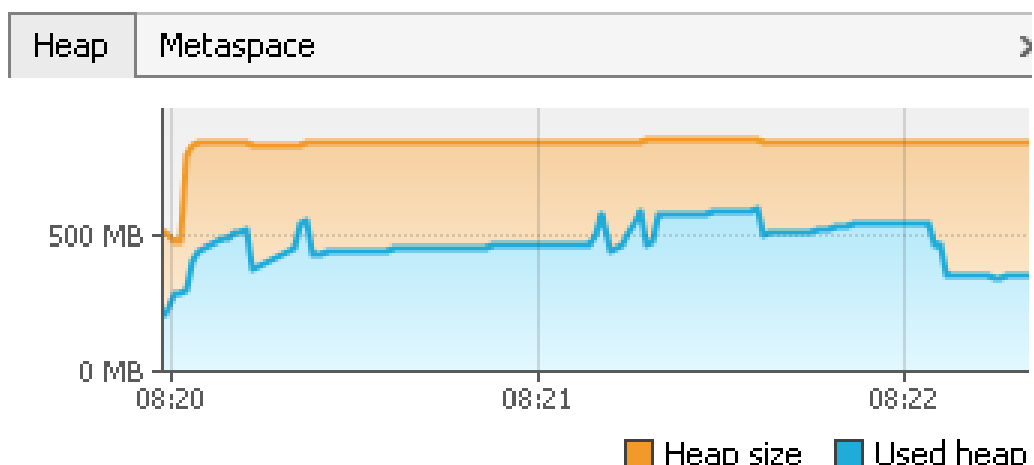


One way of reducing the heap size of a system with limited resources may be to play around with the -XX:MaxHeapFreeRatio variable. This is usually set to 70, and is the maximum percentage of the heap that is free before the GC shrinks it. Setting it to a lower value, and you will see in eg the jvisualvm profiler that a smaller heap size is usually used for your program.

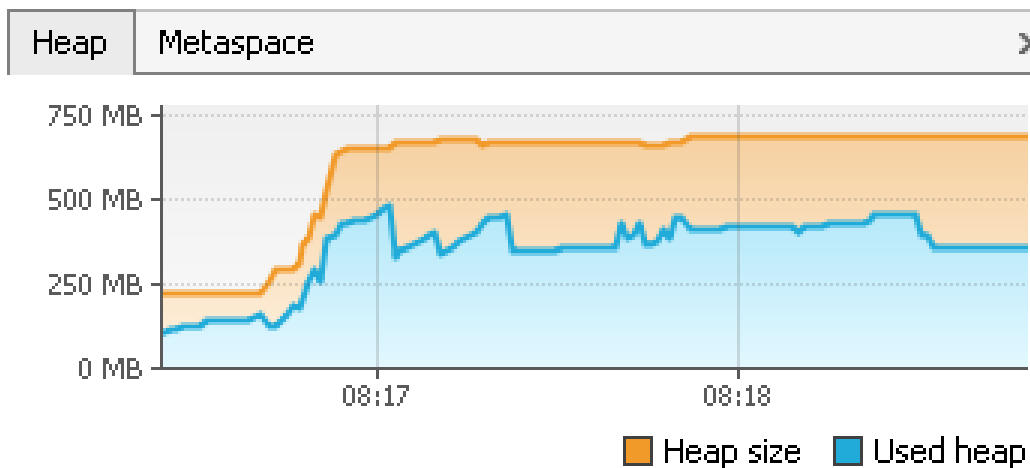
EDIT: To set small values for -XX:MaxHeapFreeRatio you must also set -XX:MinHeapFreeRatio Eg

```
java -XX:MinHeapFreeRatio=10 -XX:MaxHeapFreeRatio=25 H
```

EDIT2: Added an example for a real application that starts and does the same task, one with default parameters and one with 10 and 25 as parameters. I didn't notice any real speed difference, although java in theory should use more time to increase the heap in the latter example.



At the end, max heap is 905, used heap is 378



At the end, max heap is 722, used heap is 378

This actually have some inpact, as our application runs on a remote desktop server, and many users may run it at once.

Share Improve this answer

edited Jan 22, 2015 at 7:35

Follow

answered Jan 16, 2015 at 7:21



runholen

960 ● 11 ● 24



4



Just a thought, but you may check the influence of [a](#) `ulimit -v` option.

That is not an actual solution since it would limit address space available for *a//* process, but that would allow you to check the behavior of your application with a limited virtual memory.

Share Improve this answer

answered Feb 18, 2009 at 14:51

Follow



VonC

1.3m ● 558 ● 4.7k ● 5.6k

That is exactly what my problem is. My Heap is set to 64M, but linux reserves 204MB. If I set the ulimit below 204, the application doesn't run at all. – [Mario Ortégón](#) Feb 18, 2009 at 14:57

Interesting: setting the ulimit might have unintended side-effect for other processes, explaining why the application is not able to run. – [VonC](#) Feb 18, 2009 at 14:59

The problem seems to be that Java requires to reserve this bigger amount of Virtual memory even though that it won't use it. In windows the virtual memory used and the Xmx setting are rather closer. – [Mario Ortégón](#) Feb 18, 2009 at 15:00

Did you try it with a JRockit JVM ? – [VonC](#) Feb 18, 2009 at 15:07

Since the memory allocation of the JVM is the sum of the Heap Allocation and of the Perm Size (the first can be fixed using the -Xms and -Xmx options), did you try some settings with -XX:PermSize and -XX:MaxPermSize (default from 32MB to 64MB depending on JVM version) ? – [VonC](#) Feb 18, 2009 at 15:10



0

No, you can't configure memory amount needed by VM. However, note that this is virtual memory, not resident, so it just stays there without harm if not actually used.



Alernatively, you can try some other JVM then Sun one, with smaller memory footprint, but I can't advise here.





Share Improve this answer

answered Feb 18, 2009 at 14:28

Follow



Marko

31.3k ● 18 ● 78 ● 108



Sun's java 1.4 has the following arguments to control memory size:

0



-Xmsn Specify the initial size, in bytes, of the memory allocation pool. This value must be a multiple of 1024 greater than 1MB. Append the letter k or K to indicate kilobytes, or m or M to indicate megabytes. The default value is 2MB.

Examples:

```
-Xms6291456  
-Xms6144k  
-Xms6m
```

-Xmxn Specify the maximum size, in bytes, of the memory allocation pool. This value must a multiple of 1024 greater than 2MB. Append the letter k or K to indicate kilobytes, or m or M to indicate megabytes. The default value is 64MB.

Examples:

```
-Xmx83886080  
-Xmx81920k  
-Xmx80m
```

<http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/java.html>

Java 5 and 6 have some more. See

<http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp>

Share Improve this answer

answered Feb 18, 2009 at 14:35

Follow



Paul Tomblin

183k ● 59 ● 323 ● 410

-
- 1 The problem I have is not with the Heap Size, but with the amount of Virtual Memory that is assigned by Linux
– **Mario Ortégón** Feb 18, 2009 at 14:51

Read kdgregory's explanation. Reducing the heap size, "New size", and the other configurable parameters will reduce the amount of REAL memory the jvm takes. – **Paul Tomblin** Feb 18, 2009 at 15:41 ✎

He may have a legitimate problem. Some applications (like one I wrote) mmap a 1 GB file and some systems only have 2 GB of virtual memory, some of which gets filled with shared libraries. And if this is the problem he should definitely disable DSO randomization. There is an option in /proc. – **Zan Lynx** Jun 11, 2010 at 4:32
