Memory Footprint of a Java Process

Zhihu: tkanng

GitHub: tkanng

Email: tkanng@gmail.com

Typical Questions

- Why is Java using much more memory than heap size?
- How to keep the memory used by a Java process under control?
- How to detect the memory usage of off-heap?
- Why does NMT (Native Memory Tracking) report more / less committed memory than the linux process resident set size?
- ……

Memory Footprint of a Java Process

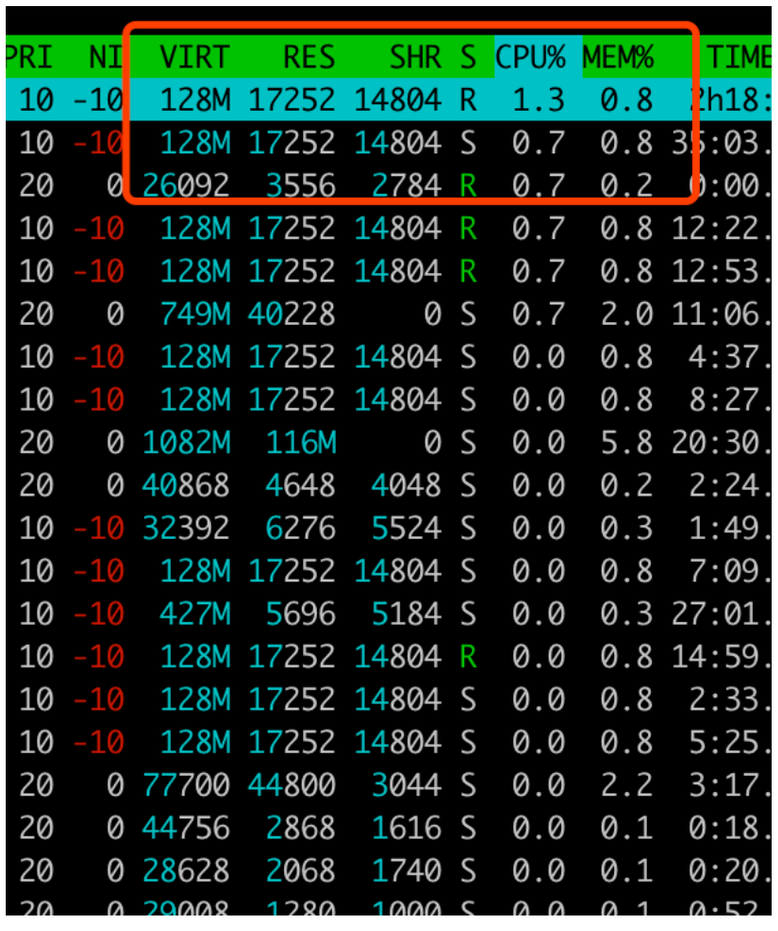
- Linux Memory Management
- Memory Footprint of a Java process
- Tools to detect memory usage of a Java process
- Examples

Linux Memory Management

Linux Memory Management

- RSS(RES): Resident Set Size

- VIRT: Virtual memory used by the task. (It matters for 32-bit OS, cause it's maximum virtual memory is just 4G)



Linux Memory Management

- Allocate physical memory only if it is touched.

Demo program 1: allocate memory without using it.

```
#include <stdio.h>
#include <stdlib.h>

int main (void) {
        int n = 0;

        while (1) {
                if (malloc(1<<20) == NULL) {
                        printf("malloc failure after %d MiB\n", n);
                        return 0;
                }
                printf ("got %d MiB\n", ++n);
        }
}
```

Demo program 2: allocate memory and actually touch it all.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main (void) {
        int n = 0;
        char *p;

        while (1) {
                if ((p = malloc(1<<20)) == NULL) {
                        printf("malloc failure after %d MiB\n", n);
                        return 0;
                }
                memset (p, 0, (1<<20));
                printf ("got %d MiB\n", ++n);
        }
}
```

https://www.win.tue.nl/~aeb/linux/lk/lk-9.html

Typically, the first demo program will get a very large amount of memory before `malloc()` returns `NULL`. The second demo program will get a much smaller amount of memory, now that earlier obtained memory is actually used.

Linux Memory Management

- Allocate physical memory only if it is touched.

Demo program 3: first allocate, and use later.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define N       10000

int main (void) {
        int i, n = 0;
        char *pp[N];

        for (n = 0; n < N; n++) {
                pp[n] = malloc(1<<20);
                if (pp[n] == NULL)
                        break;
        }
        printf("malloc failure after %d MiB\n", n);

        for (i = 0; i < n; i++) {
                memset (pp[i], 0, (1<<20));
                printf("%d\n", i+1);
        }

        return 0;
}
```

For example:

- On an 8 MiB machine without swap running 1.2.11:
  demo1: 274 MiB, demo2: 4 MiB, demo3: 270 / oom after 1 MiB: `Killed.`
- Idem, with 32 MiB swap:
  demo1: 1528 MiB, demo2: 36 MiB, demo3: 1528 / oom after 23 MiB: `Killed.`
- On a 32 MiB machine without swap running 2.0.34:
  demo1: 1919 MiB, demo2: 11 MiB, demo3: 1919 / oom after 4 MiB: `Bus error.`
- Idem, with 62 MiB swap:
  demo1: 1919 MiB, demo2: 81 MiB, demo3: 1919 / oom after 74 MiB: The machine hangs. After several seconds: `Out of memory for bash.` `Out of memory for crond. Bus error.`
- On a 256 MiB machine without swap running 2.6.8.1:
  demo1: 2933 MiB, demo2: after 98 MiB: `Killed.` Also: `Out of Memory: Killed process 17384 (java_vm).` demo3: 2933 / oom after 135 MiB: `Killed.`
- Idem, with 539 MiB swap:
  demo1: 2933 MiB, demo2: after 635 MiB: `Killed.` demo3: oom after 624 MiB: `Killed.`

https://www.win.tue.nl/~aeb/linux/lk/lk-9.html

The third program will get the same large amount as the first program, and then is killed when it wants to use its memory. (On a well-functioning system, like Solaris, the three demo programs obtain the same amount of memory and do not crash but see `malloc()` return `NULL`.)

Linux Memory Management

```
/proc/[pid]/status
       Provides much of the information in /proc/[pid]/stat and /proc/[pid]/statm in a format that's easier for humans to parse.  Here's an example:

           $ cat /proc/$$/status
           Name:   bash
           State:  S (sleeping)
           Tgid:   3515
           Pid:    3515
           PPid:   3452
           TracerPid:      0
           Uid:    1000    1000    1000    1000
           Gid:    100     100     100     100
           FDSize: 256
           Groups: 16 33 100
           VmPeak:     9136 kB
           VmSize:     7896 kB
           VmLck:         0 kB
           VmPin:         0 kB
           VmHWM:      7572 kB
           VmRSS:      6316 kB
           VmData:     5224 kB
           VmStk:        88 kB
           VmExe:       572 kB
           VmLib:      1708 kB
           VmPMD:         4 kB
           VmPTE:        20 kB
           VmSwap:        0 kB
           Threads:        1
           SigQ:   0/3067
           SigPnd: 0000000000000000
           ShdPnd: 0000000000000000
           SigBlk: 0000000000010000
           SigIgn: 0000000000384004
           SigCgt: 000000004b813efb
           CapInh: 0000000000000000
           CapPrm: 0000000000000000
           CapEff: 0000000000000000
           CapBnd: ffffffffffffffff
           CapAmb:   0000000000000000
           Seccomp:        0
           Cpus_allowed:   00000001
           Cpus_allowed_list:      0
           Mems_allowed:   1
           Mems_allowed_list:      0
           voluntary_ctxt_switches:        150
           nonvoluntary_ctxt_switches:     545
al page proc(5) line 889 (press h for help or q to quit)
```

Linux Memory Management

```
/proc/[pid]/maps
       A file containing the currently mapped memory regions and their access permissions.  See mmap(2) for some further information about memory mappings.

       The format of the file is:

address           perms offset  dev   inode       pathname
00400000-00452000 r-xp 00000000 08:02 173521      /usr/bin/dbus-daemon
00651000-00652000 r--p 00051000 08:02 173521      /usr/bin/dbus-daemon
00652000-00655000 rw-p 00052000 08:02 173521      /usr/bin/dbus-daemon
00e03000-00e24000 rw-p 00000000 00:00 0           [heap]
00e24000-011f7000 rw-p 00000000 00:00 0           [heap]
...
35b1800000-35b1820000 r-xp 00000000 08:02 135522  /usr/lib64/ld-2.15.so
35b1a1f000-35b1a20000 r--p 0001f000 08:02 135522  /usr/lib64/ld-2.15.so
35b1a20000-35b1a21000 rw-p 00020000 08:02 135522  /usr/lib64/ld-2.15.so
35b1a21000-35b1a22000 rw-p 00000000 00:00 0
35b1c00000-35b1dac000 r-xp 00000000 08:02 135870  /usr/lib64/libc-2.15.so
35b1dac000-35b1fac000 ---p 001ac000 08:02 135870  /usr/lib64/libc-2.15.so
35b1fac000-35b1fb0000 r--p 001ac000 08:02 135870  /usr/lib64/libc-2.15.so
35b1fb0000-35b1fb2000 rw-p 001b0000 08:02 135870  /usr/lib64/libc-2.15.so
...
f2c6ff8c000-7f2c7078c000 rw-p 00000000 00:00 0    [stack:986]
...
7fffb2c0d000-7fffb2c2e000 rw-p 00000000 00:00 0   [stack]
7fffb2d48000-7fffb2d49000 r-xp 00000000 00:00 0   [vdso]

       The address field is the address space in the process that the mapping occupies.  The perms field is a set of permissions:

            r = read
            w = write
            x = execute
            s = shared
            p = private (copy on write)

       The offset field is the offset into the file/whatever; dev is the device (major:minor); inode is the inode on that device.  0 indicates that no inode is associated with the memory region, as would be the case with  BSS
       (uninitialized data).

       The pathname field will usually be the file that is backing the mapping.  For ELF files, you can easily coordinate with the offset field by looking at the Offset field in the ELF program headers (readelf -l).

       There are additional helpful pseudo-paths:

            [stack]
                  The initial process's (also known as the main thread's) stack.

            [stack:<tid>] (since Linux 3.4)
                  A thread's stack (where the <tid> is a thread ID).  It corresponds to the /proc/[pid]/task/[tid]/ path.

            [vdso] The virtual dynamically linked shared object.

            [heap] The process's heap.

       If  the pathname field is blank, this is an anonymous mapping as obtained via the mmap(2) function.  There is no easy way to coordinate this back to a process's source, short of running it through gdb(1), strace(1), or
       similar.

       Under Linux 2.0, there is no field giving pathname.
```

Linux Memory Management

- /proc/[pid]/mem: This file can be used to access the pages of a process's memory through open(2), read(2), and lseek(2).

```
# open and read mem
try:
    mem_file = open(mem_filename, 'rb+')
except IOError as e:
    print("[ERROR] Can not open file {}:".format(mem_filename))
    print("        I/O error({}): {}".format(e.errno, e.strerror))
    maps_file.close()
    exit(1)

# read heap
mem_file.seek(addr_start)
heap = mem_file.read(addr_end - addr_start)

# find string
try:
    i = heap.index(bytes(search_string, "ASCII"))
except Exception:
    print("Can't find '{}'".format(search_string))
    maps_file.close()
    mem_file.close()
    exit(0)
print("[*] Found '{}' at {:x}".format(search_string, i))

# write the new string
print("[*] Writing '{}' at {:x}".format(write_string, addr_start + i))
mem_file.seek(addr_start + i)
mem_file.write(bytes(write_string, "ASCII"))

# close files
maps_file.close()
mem_file.close()

# there is only one heap in our example
break
```

https://blog.holbertonschool.com/hack-the-virtual-memory-c-strings-proc/

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

/**
 * main - uses strdup to create a new string, loops forever-ever
 *
 * Return: EXIT_FAILURE if malloc failed. Other never returns
 */
int main(void)
{
     char *s;
     unsigned long int i;

     s = strdup("Holberton");
     if (s == NULL)
     {
          fprintf(stderr, "Can't allocate mem with malloc\n");
          return (EXIT_FAILURE);
     }
     i = 0;
     while (s)
     {
          printf("[%lu] %s (%p)\n", i, s, (void *)s);
          sleep(1);
          i++;
     }
     return (EXIT_SUCCESS);
```

```
.
676] Holberton (0x10ff010)
677] Holberton (0x10ff010)
678] Holberton (0x10ff010)
679] Holberton (0x10ff010)
680] Holberton (0x10ff010)
681] Holberton (0x10ff010)
682] Fun w vm! (0x10ff010)
683] Fun w vm! (0x10ff010)
684] Fun w vm! (0x10ff010)
685] Fun w vm! (0x10ff010)
...
```

Memory Footprint of a Java process

Memory Footprint of a Java process

- Native Memory Tracking

  -XX:NativeMemoryTracking=detail / summary
  -XX:+UnlockDiagnosticVMOptions
  -XX:+PrintNMTStatistics

NMT 中committed，大致可以认为是RSS对应的值。

NMT 只能Track到JVM自身的内存分配情况，比如：Heap内存分配，direct byte buffer等；不能track到jni里直接调用malloc时的内存分配，这里最典型的就是ZipStream的场景。

一般情况下NMT Track出来的 committed 内存值既可能比RSS值大，也可能比RSS小，主要原因是：
- 比真实RSS小： NMT 只能Track JVM自身的内

```
Total: reserved=3535846KB, committed=2261270KB
-                 Java Heap (reserved=1257472KB, committed=1257472KB)
                            (mmap: reserved=1257472KB, committed=1257472KB)

-                     Class (reserved=1087227KB, committed=42491KB)
                            (classes #6025)
                            (malloc=5883KB #7385)
                            (mmap: reserved=1081344KB, committed=36608KB)

-                    Thread (reserved=39133KB, committed=39133KB)
                            (thread #38)
                            (stack: reserved=38964KB, committed=38964KB)
                            (malloc=125KB #196)
                            (arena=45KB #76)

-                      Code (reserved=252732KB, committed=22892KB)
                            (malloc=3132KB #5230)
                            (mmap: reserved=249600KB, committed=19760KB)

-                        GC (reserved=52265KB, committed=52265KB)
                            (malloc=5773KB #151)
                            (mmap: reserved=46492KB, committed=46492KB)

-                  Compiler (reserved=172KB, committed=172KB)
                            (malloc=41KB #225)
                            (arena=131KB #3)

-                  Internal (reserved=835091KB, committed=835091KB)
                            (malloc=835059KB #15987)
                            (mmap: reserved=32KB, committed=32KB)

-                    Symbol (reserved=9562KB, committed=9562KB)
                            (malloc=7891KB #74545)
                            (arena=1670KB #1)

-    Native Memory Tracking (reserved=1813KB, committed=1813KB)
                            (malloc=153KB #2414)
                            (tracking overhead=1661KB)

-               Arena Chunk (reserved=378KB, committed=378KB)
                            (malloc=378KB)
```

Memory Footprint of a Java process

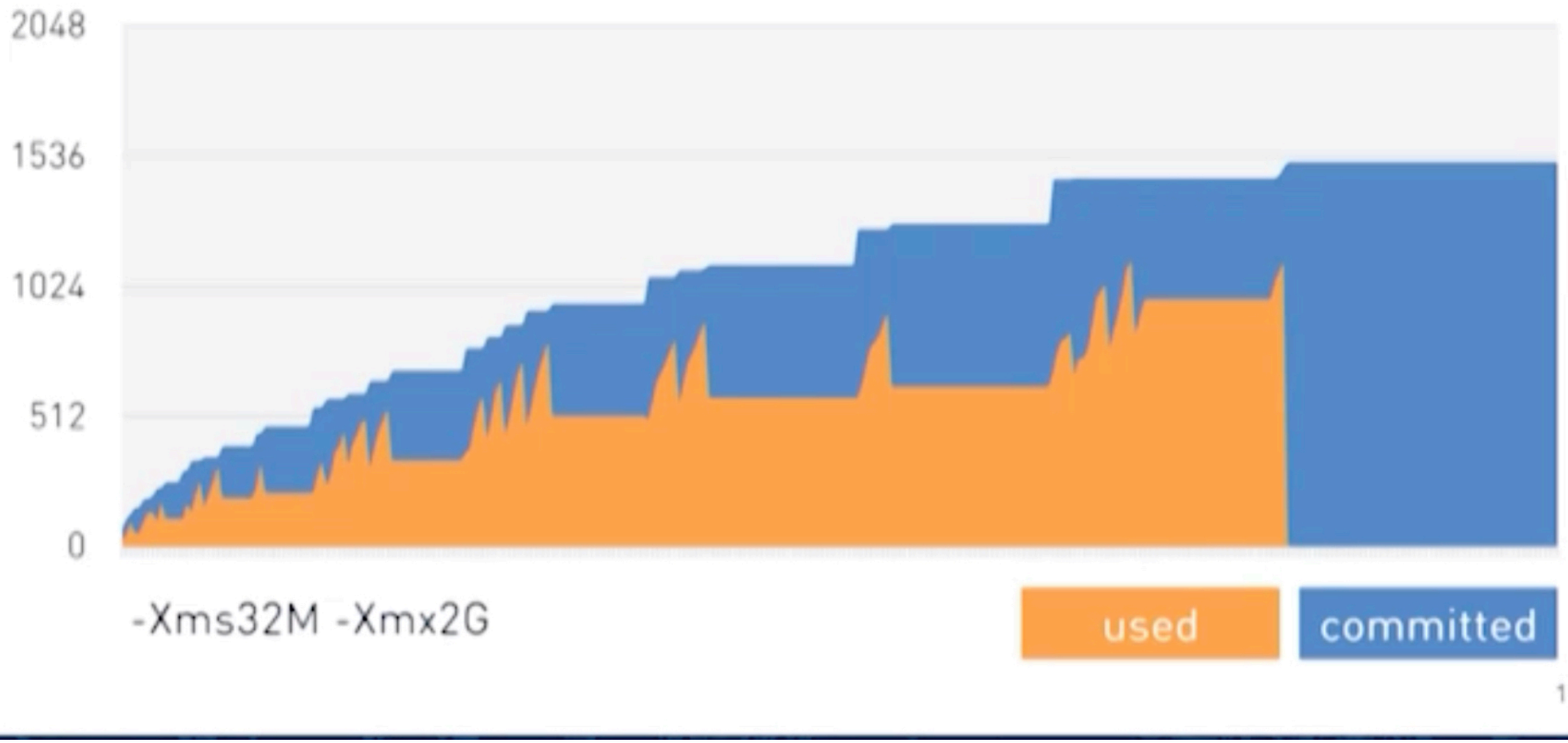
- Heap
  - Xms: Initial Heap size, NOT minimal
  - Xmx: Max Heap size
  - XX:+AlwaysPretouch

Pre-touch the Java heap during JVM initialization. Every page of the heap is thus demand-zeroed during initialization rather than incrementally during application execution.【会

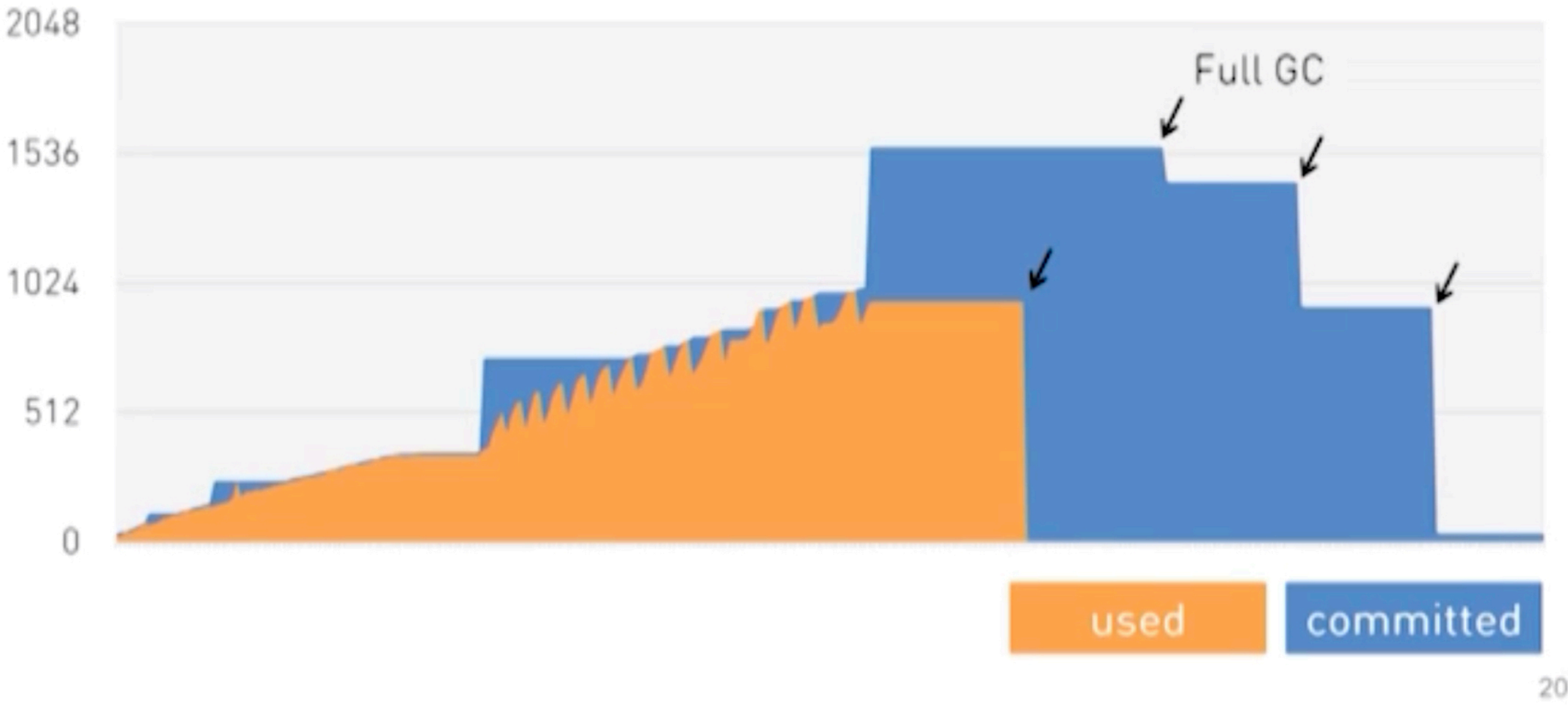
Memory Footprint of a Java process

- Heap
  - -XX:+UseAdaptiveSizePolicy(On by default)
  - -XX:MinHeapFreeRatio=40(expansion)
  - -XX:MaxHeapFreeRatio=70(shrinking)

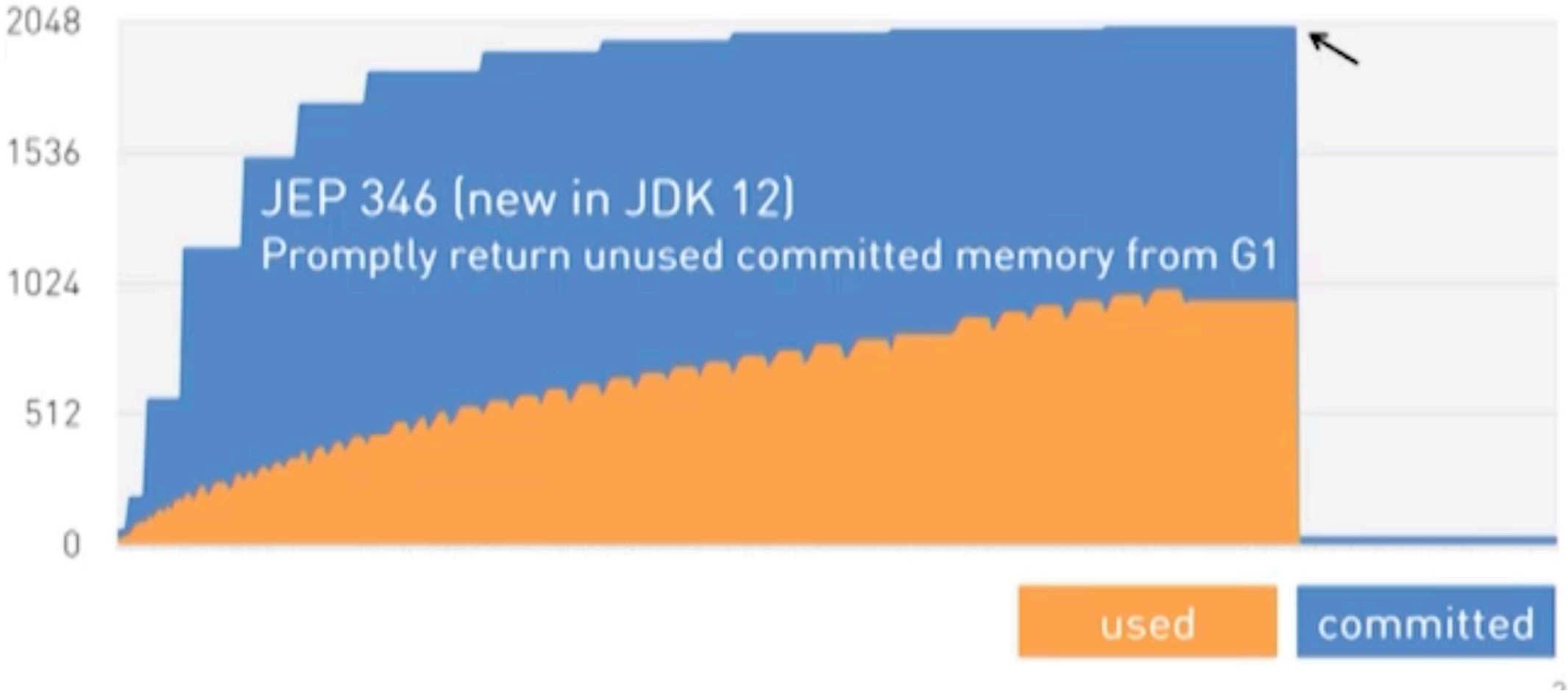
Uncommit: Parallel



Uncommit: CMS



Uncommit: G1


2048
1536
1024
512
0
JEP 346 (new in JDK 12)
Promptly return unused committed memory from G1
used
committed

Memory Footprint of a Java process

- Metaspace:
  - Class: More classes are loaded, more metaspace is used.
  - Methods
  - Annotations
  - ....

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.

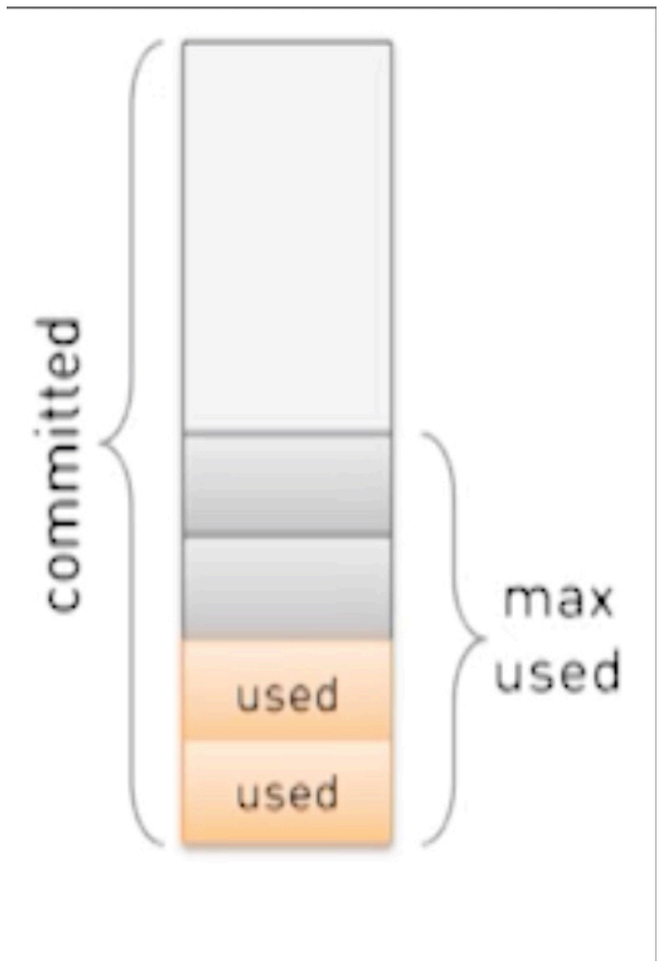
- JVM Opts:
  - -XX:MaxMetaspaceSize (unlimited by default)
  - -XX:MetaspaceSize=64M

```
2400.241: [GC concurrent-root-region-scan-start]
2400.241: [Full GC (Metadata GC Threshold) 2400.252: [GC concurrent-root-region-sc
2400.252: [GC concurrent-mark-start]
 1151M->603M(4356M), 2.6980537 secs]
   [Eden: 0.0B(2558.0M)->0.0B(2613.0M) Survivors: 55.0M->0.0B Heap: 1151.7M(4356.0
 [Times: user=3.92 sys=0.00, real=2.70 secs]
```

Memory Footprint of a Java process

```
-                    Thread (reserved=39133KB, committed=39133KB)
                            (thread #38)
                            (stack: reserved=38964KB, committed=38964KB)
                            (malloc=125KB #196)
                            (arena=45KB #76)
```

- Thread
  - Xss
  - Committed!=Resident



https://github.com/apangin/jstackmem/blob/master/jstackmem.py

Memory Footprint of a Java process

```
Native Memory Tracking:

Total: reserved=4023326KB  committed=2762382KB
-                 Java Heap (reserved=1331200KB, committed=1331200KB)
                            (mmap: reserved=1331200KB, committed=1331200KB)

-                     Class (reserved=1108143KB, committed=64559KB)
                            (classes #8621)
                            (malloc=6319KB #17371)
                            (mmap: reserved=1101824KB, committed=58240KB)

-                    Thread (reserved=1190668KB, committed=1190668KB)
                            (thread #1154)
                            (stack: reserved=1185284KB, committed=1185284KB)
                            (malloc=3809KB #5771)
                            (arena=1575KB #2306)

-                      Code (reserved=255744KB, committed=38384KB)
                            (malloc=6144KB #8858)
                            (mmap: reserved=249600KB, committed=32240KB)

-                        GC (reserved=54995KB, committed=54995KB)
                            (malloc=5775KB #217)
                            (mmap: reserved=49220KB, committed=49220KB)

-                  Compiler (reserved=267KB, committed=267KB)
                            (malloc=137KB #333)
                            (arena=131KB #3)

-                  Internal (reserved=65106KB, committed=65106KB)
                            (malloc=65074KB #29652)
                            (mmap: reserved=32KB, committed=32KB)

-                    Symbol (reserved=13622KB, committed=13622KB)
                            (malloc=12016KB #128199)
                            (arena=1606KB #1)
```

This shows 2.7GB of committed memory, including 1.3GB of allocated heap and almost 1.2GB of allocated thread stacks (using many threads).

However, when running `ps ax -o pid,rss | grep <mypid>` or `top` it shows only 1.6GB of `RES/rss` resident memory. Checking swap says none in use:

```
free -m
             total       used       free     shared    buffers     cached
Mem:        129180      99348      29831          0       2689      73024
-/+ buffers/cache:      23633     105546
Swap:        15624          0      15624
```

Why does the JVM indicate 2.7GB memory is committed when only 1.6GB is resident? Where did the rest go?

https://stackoverflow.com/questions/31173374/why-does-a-jvm-report-more-committed-memory-than-the-linux-process-resident-set

Memory Footprint of a Java process

- Code：
  - Contains dynamically generated code: JIT-compiled methods, interpreter and run-time stubs. Its size is limited by -XX:ReservedCodeCacheSize (240M by default).

```
CodeCache: size=245760Kb used=1108Kb max_used=1108Kb free=244651Kb
 bounds [0x0000000114d8e000, 0x0000000114ffe000, 0x0000000123d8e000]
 total_blobs=269 nmethods=28 adapters=155
 compilation: enabled

Process finished with exit code 0
```

  - Turn off -XX:-TieredCompilation to reduce the amount of compiled code and thus the Code Cache usage.

  - -XX:+PrintCodeCache

Memory Footprint of a Java process

- Garbage Collector:
  - Mark Bitmap
  - Mark Stack (for traversing object graph)
  - Remembered Set

GC memory overhead

GC	Overhead, MB	%
Serial	7	0,3%
Shenandoah	38	1,9%
CMS	76	3,7%
Parallel	90	4,4%
G1	166	8,1%
Z	238	11,6%

OpenJDK 13 | Heap size = 2 GB

Memory Footprint of a Java process

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

Memory Footprint of a Java process

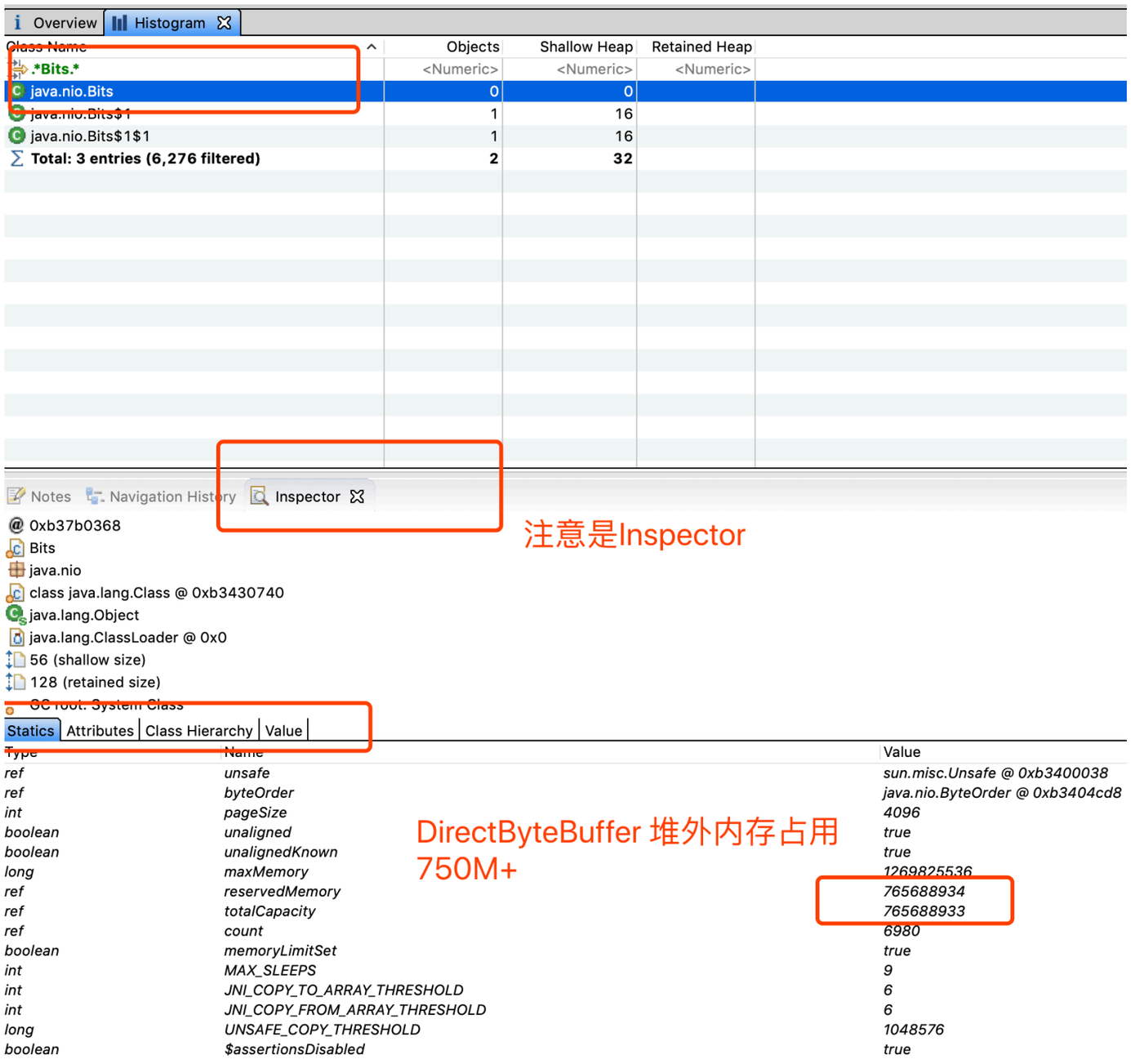
```
201
202   -                 Internal (reserved=835091KB, committed=835091KB)
203                            (malloc=835059KB #15987)
204                            (mmap: reserved=32KB, committed=32KB)
205
206   -                   Symbol (reserved=9562KB, committed=9562KB)
207                            (malloc=7891KB #74545)
208                            (arena=1670KB #1)
209
```

- Internal
  - Direct Byte Buffer

```
1374      [0x00007f6cdd49697e] JavaMain+0x9e
1375  Details:
1376
1377  [0x00007f6cdc95f0 3] Unsafe_AllocateMemory+0xc3
1378  [0x00007f6cc5f5c3 8]
1379                            (malloc=800998KB #7495)
1380
1381  [0x00007f6cdc95f0 3] Unsafe_AllocateMemory+0xc3
1382  [0x00007f6cc50186 7]
1383                            (malloc=27002KB #244)
1384
1385  [0x00007f6cdc8574e5] ArrayAllocator<StarTask, (MemoryType)1>::allocate(unsigned long
1386  [0x00007f6cdc855c10] PSPromotionManager::PSPromotionManager()+0x1f0
1387  [0x00007f6cdc855e5d] PSPromotionManager::initialize()+0x13d
1388  [0x00007f6cdc95a724] universe_post_init()+0x8a4
1389                            (malloc=5120KB #5)
```

Memory Footprint of a Java process

- Internal
  - Direct Byte Buffer



Memory Footprint of a Java process

- Symbol：

```
-                     Symbol (reserved=1464KB, committed=1464KB)
                            (malloc=944KB #161)
                            (arena=520KB #1)
```

- SymbolTable: names signatures
- StringTable: interned strings
- -XX:+PrintStringTableStatistics

```
SymbolTable statistics:
Number of buckets       :     20011 =    160088 bytes, avg   8.000
Number of entries       :     12431 =    298344 bytes, avg  24.000
Number of literals      :     12431 =    477440 bytes, avg  38.407
Total footprint         :           =    935872 bytes
Average bucket size     :     0.621
Variance of bucket size :     0.620
Std. dev. of bucket size:     0.787
Maximum bucket size     :         6
StringTable statistics:
Number of buckets       :     60013 =    480104 bytes, avg   8.000
Number of entries       :       877 =     21048 bytes, avg  24.000
Number of literals      :       877 =     59120 bytes, avg  67.412
Total footprint         :           =    560272 bytes
Average bucket size     :     0.015
Variance of bucket size :     0.015
Std. dev. of bucket size:     0.121
Maximum bucket size     :         2
```

Memory Footprint of a Java process

- Others
  - JNI malloc
  - ZipInputStream
  - ……

Memory Footprint of a Java process

- Why is Java using much more memory than heap size?
    - Off-Heap
- How to keep the memory used by a Java process under control?
    - No way!
- How to detect the memory usage of off-heap?
    - pmap, strace,gdb….
- Why does NMT (Native Memory Tracking) report more / less committed memory than the linux process resident set size?
    - Less: JNI malloc
    - More: Thread (Committed != Resident)
- …….

Tools to detect memory usage of a Java process

Tools to detect memory usage of a Java process

- Heap
  - -XX:+PrintFlagsFinal
  - jinfo
  - jmap
  - MAT
  - …..

Tools to detect memory usage of a Java process

- Off-Heap
  - NMT
  - pmap
  - gdb
  - strace
  - async-profiler
  - …..

Examples

Examples

- https://zhuanlan.zhihu.com/p/54048271
- https://zhuanlan.zhihu.com/p/60976273

Thanks!