



Tracking Off Heap Memory in the JVM

Nick Palmer (2018)



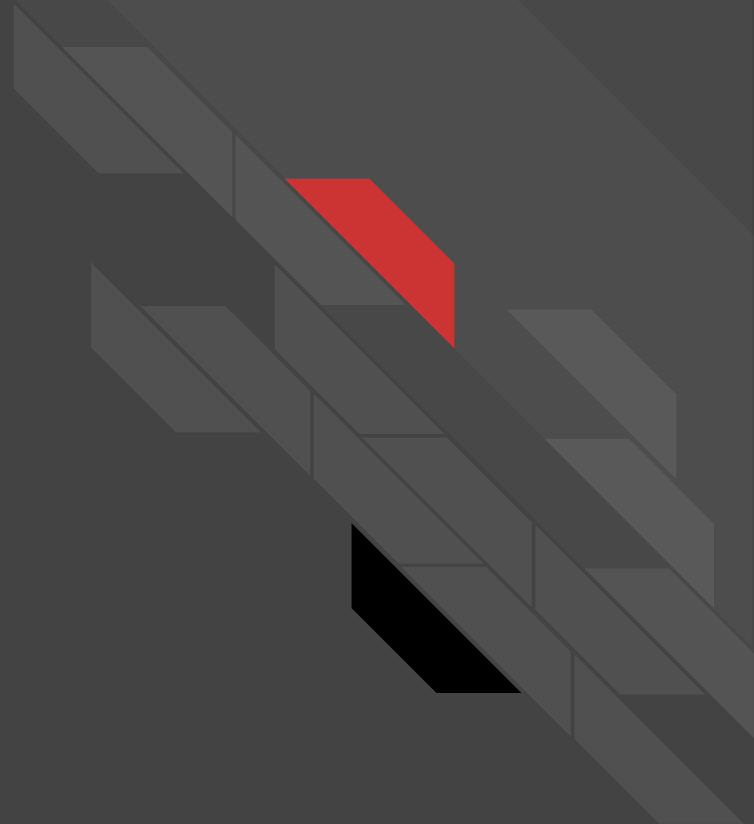
Disclaimer

I'm talking about Java 8 here.

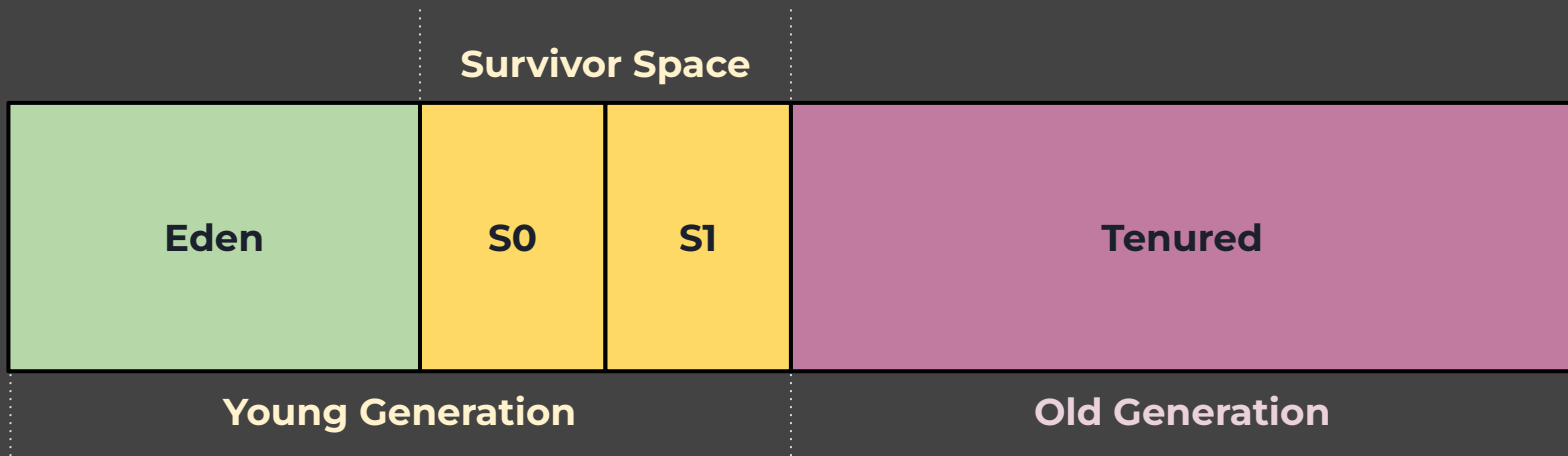
HotSpot only. G1 collector only. Probably not be accurate for anything else.

Your mileage may vary...

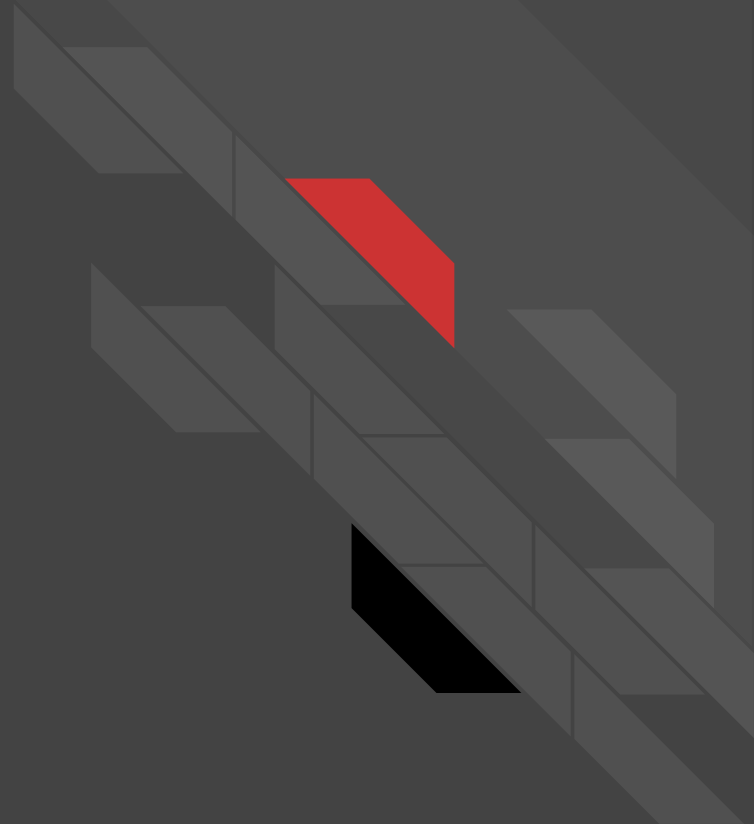
What do we mean by
JVM Heap?



Typical View of HotSpot G1 Heap



Okay, so what's
“**off-heap**”?

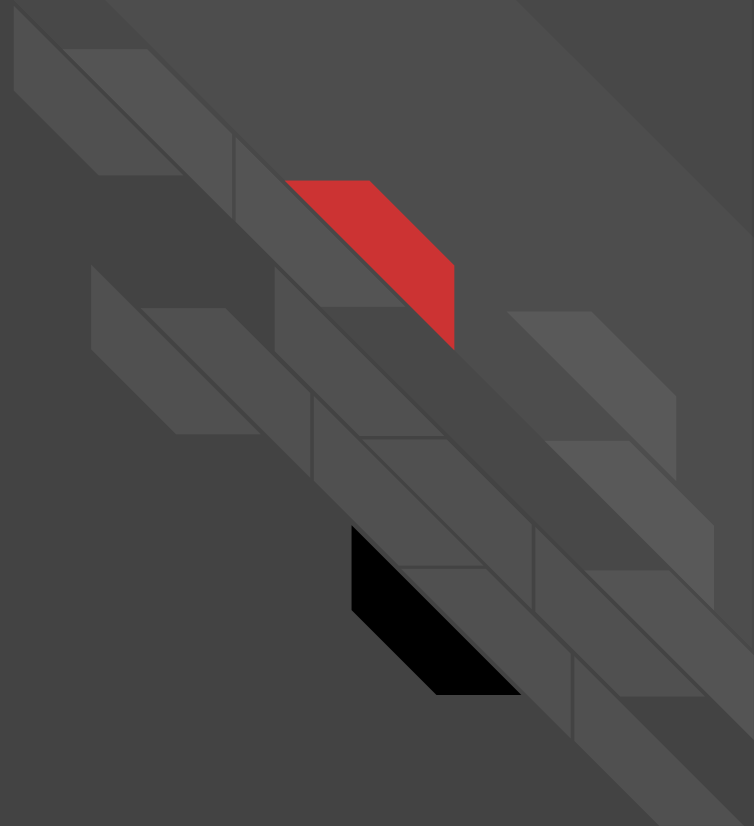




JVM Off-Heap

- Metaspace
 - Class metadata
- Threads (frame stack here)
 - Every thread you make gets allocated a stack for method frames & local variables etc.
- JVM itself, native stack & heap
 - The JVM is a program, it uses memory too!

Is that all?





Off-Piste Off-Heap

- Direct buffers
 - Byte Buffers you can allocate outside the heap
- Zip streams
 - HotSpot JVM implements the zip code using native code, while zipping and not closed memory will be allocated off-heap
- JNI/JNA libraries
 - Here you're on your own in terms of memory management, could be doing anything

How much memory am I using?

``htop``




PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
10259	palmer	20	0	17.1G	1263M	33032	S	69.5	2.0	2:35.74	/mnt/scratch/beantcache/jdk/1.8.0_172-1/bin/java -Xms512m -Xmx1g
28660	palmer	20	0	247M	65456	8632	R	43.1	0.1	12h22:31	python support/bigfeedback/src/main/py/littlefeedback.py
8702	palmer	20	0	10.1G	497M	37296	S	43.1	0.8	0:59.82	java -Dvisualvm.display.name=fault-monitoring-svc -Dcom.sun.management
6664	palmer	20	0	9.7G	375M	37032	S	39.9	0.6	0:25.99	java -Dvisualvm.display.name=broker-archiver -Dcom.sun.management
9347	palmer	20	0	8802M	357M	37164	S	33.8	0.6	0:50.34	java -Dvisualvm.display.name=global-admin-svc -Dcom.sun.management
6698	palmer	20	0	10.2G	463M	37252	S	32.0	0.7	0:53.52	java -Dvisualvm.display.name=member-portal-internal -Dcom.sun.management

Check out the process information pseudo-filesystem

See ``man proc`` for more info

``less /proc/8702/status``



```
VmPeak: 10647712 kB
VmSize: 10584224 kB
VmLck: 0 kB
VmPin: 0 kB
VmHWM: 543952 kB
VmRSS: 525884 kB
RssAnon: 488588 kB
RssFile: 37264 kB
RssShmem: 32 kB
VmData: 817416 kB
VmStk: 144 kB
VmExe: 4 kB
VmLib: 24532 kB
```



Example

Test application running on HotSpot JVM 1.8 running on Linux 4.17 (x86_64)

Heap size set using -Xmx to 32MB

Roughly 2MB of JARs.

It starts a few threads and uses one 30KB native library via JNI.

Finger in the air estimate for max memory use: nothing over 75MB?

```
palmern@gemini OffHeapLeakExample-1.0-SNAPSHOT (master)$ ./bin/OffHeapLeakExample
17:39:14.512 [main] INFO uk.co.palmr.offheapleakexample.Main - Starting application...
17:39:14.532 [main] INFO uk.co.palmr.offheapleakexample.Main - ...done
17:39:14.532 [main] INFO uk.co.palmr.offheapleakexample.Main - Pausing for 20 seconds...
```

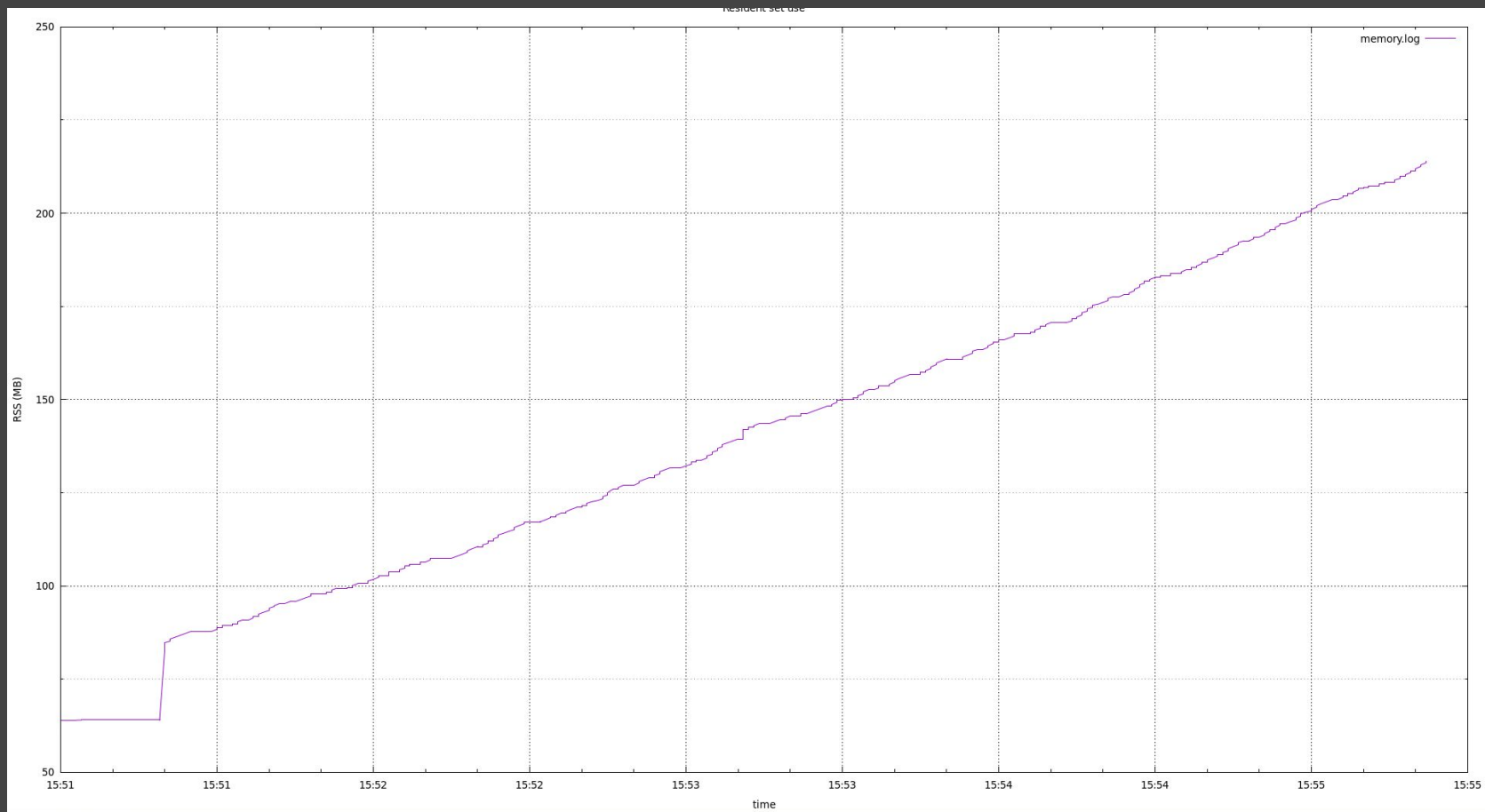
```
palmern@gemini OffHeapLeakExample-1.0-SNAPSHOT (master)$ ./bin/log_memory.sh
Application found with PID=17710
2018-08-18_17:45:22 67284 kB
2018-08-18_17:45:23 67284 kB
2018-08-18_17:45:23 67284 kB
2018-08-18_17:45:24 67284 kB
```

```
palmern@gemini OffHeapLeakExample-1.0-SNAPSHOT (master)$ jcmd 17710 VM.native_memory baseline
17710:
Baseline succeeded
```

Example - Starting

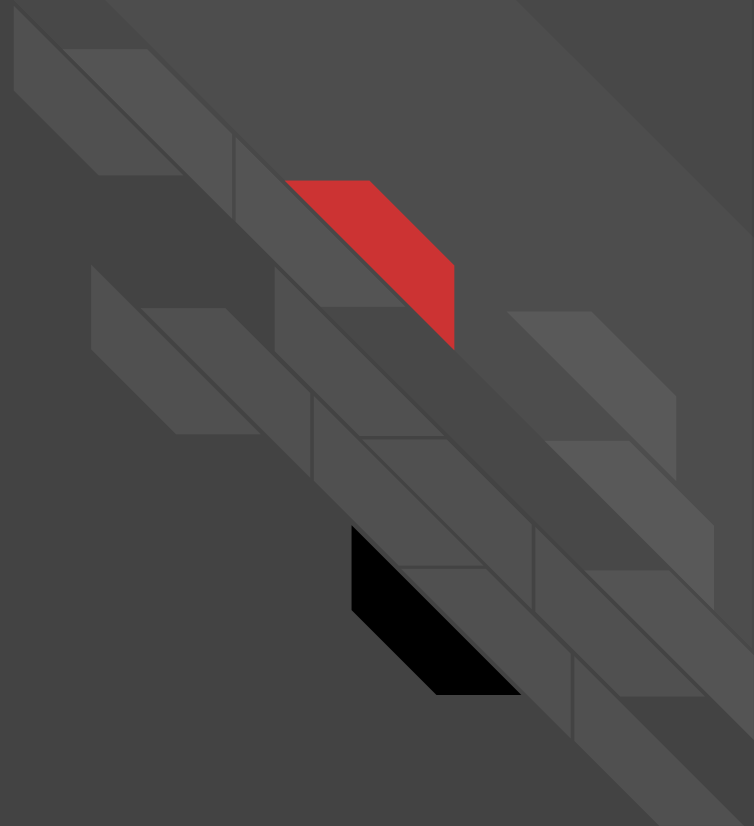


And now, we wait...



Example - Plotting RSS

Okay, so where do I
start?





JMAP

- Jmap command - Java memory map
- Prints shared object memory maps or heap memory details
- Useful to get an idea of where the JVM thinks memory has been allocated
- <https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr014.html>

```
palmern@gemini OffHeapLeakExample-1.0-SNAPSHOT (master)$ jmap -heap 27444
Attaching to process ID 27444, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.172-b11

using thread-local object allocation.
Parallel GC with 20 thread(s)

Heap Configuration:
  MinHeapFreeRatio      = 0
  MaxHeapFreeRatio      = 100
  MaxHeapSize           = 33554432 (32.0MB)
  NewSize               = 11010048 (10.5MB)
  MaxNewSize            = 11010048 (10.5MB)
  OldSize               = 22544384 (21.5MB)
  NewRatio              = 2
  SurvivorRatio         = 8
  MetaspaceSize         = 21807104 (20.796875MB)
  CompressedClassSpaceSize = 1073741824 (1024.0MB)
  MaxMetaspaceSize      = 1759218604415 MB
  G1HeapRegionSize      = 0 (0.0MB)

Heap Usage:
PS Young Generation
Eden Space:
  capacity = 8912896 (8.5MB)
  used     = 6555672 (6.251976013183594MB)
  free     = 2357224 (2.2480239868164062MB)
  73.55265897863052% used
From Space:
  capacity = 1048576 (1.0MB)
  used     = 1024960 (0.97747802734375MB)
  free     = 23616 (0.02252197265625MB)
  97.747802734375% used
To Space:
  capacity = 1048576 (1.0MB)
  used     = 0 (0.0MB)
  free     = 1048576 (1.0MB)
  0.0% used
PS Old Generation
  capacity = 22544384 (21.5MB)
  used     = 7114688 (6.78509521484375MB)
  free     = 15429696 (14.71490478515625MB)
  31.558582394622093% used

3467 interned Strings occupying 2367232 bytes.
```

jmap -heap <example-pid>

Places looked

Heap ✓

Off-heap

Metaspace

Direct Buffers

Zip Steams

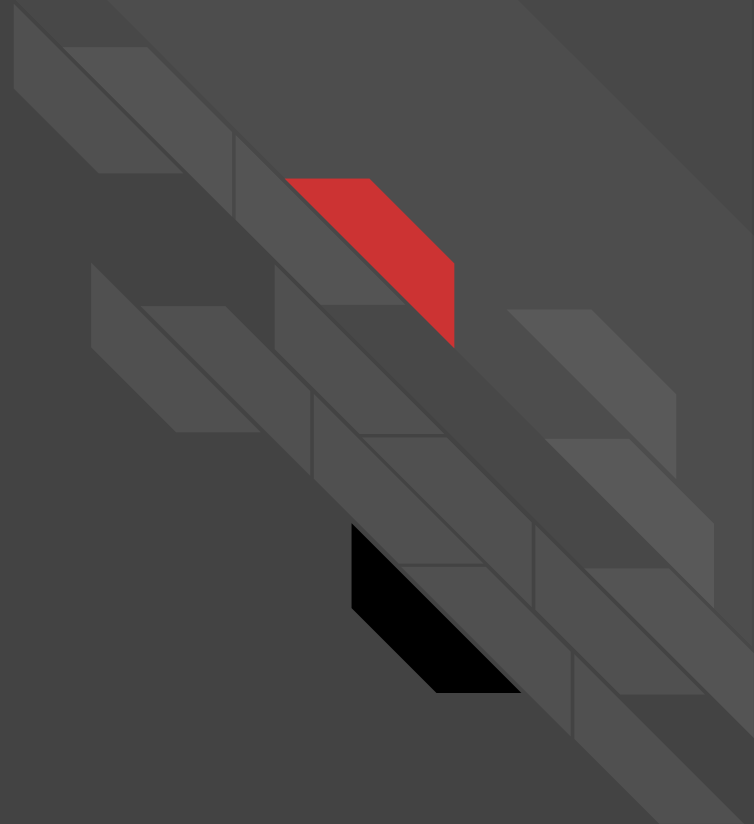
Threads

JVM

JNI/JNA



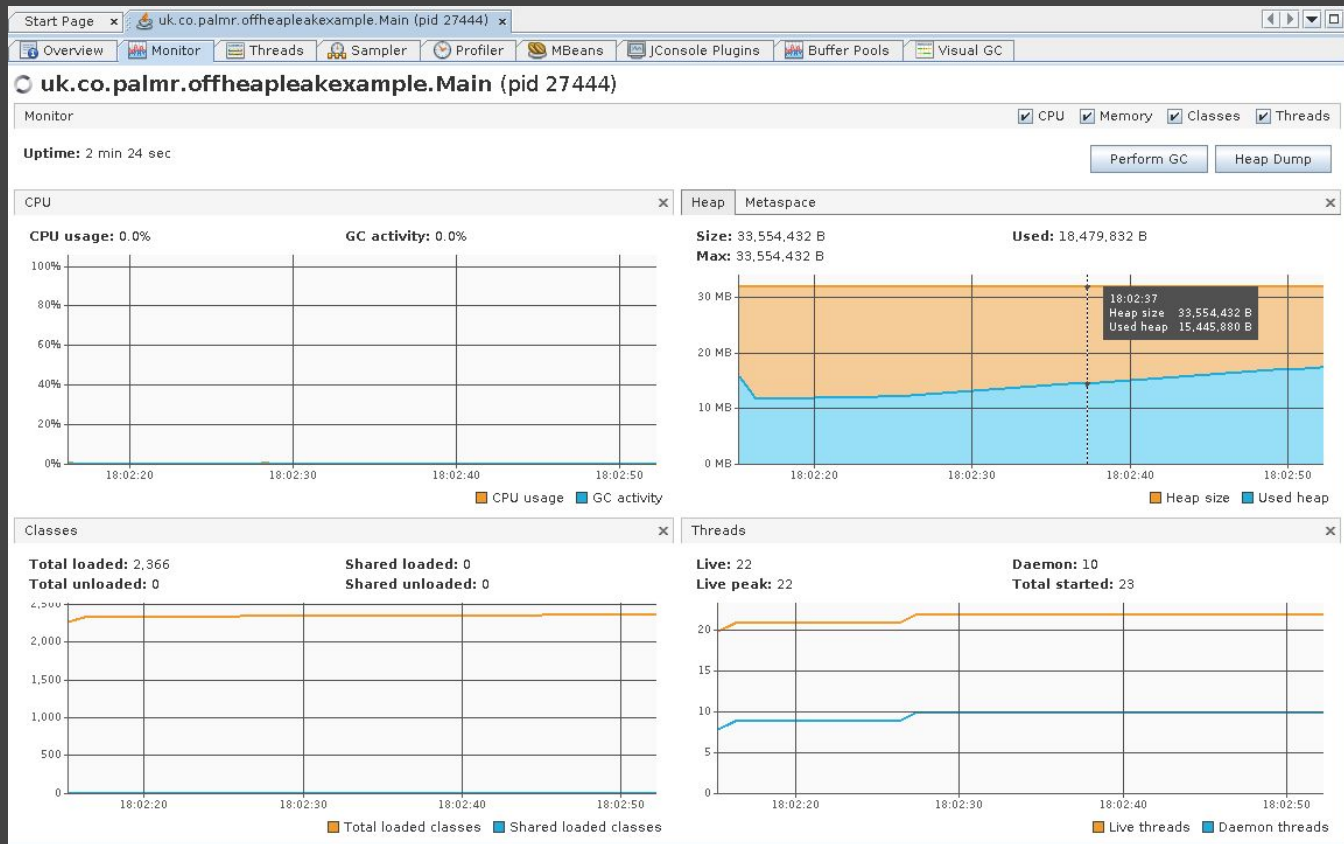
Time to roll up your
sleeves



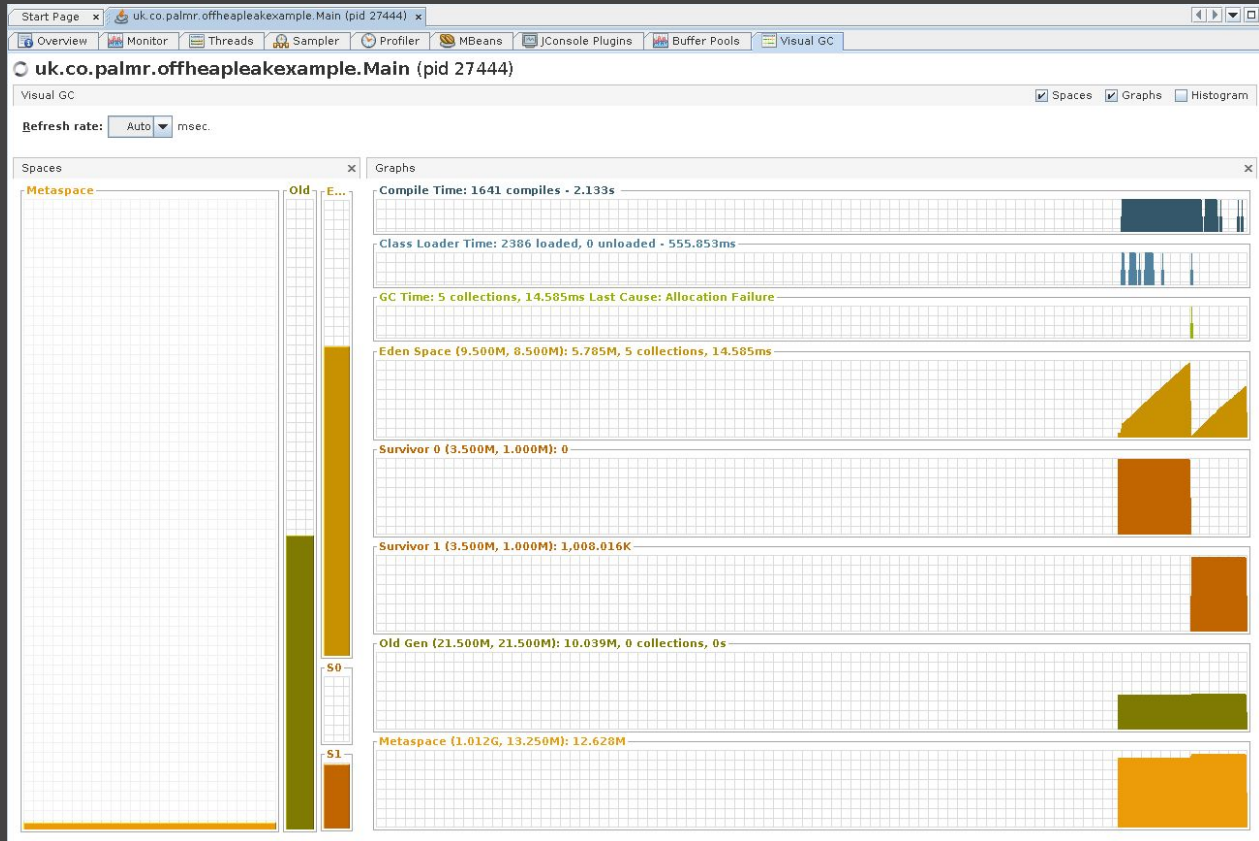


JVisualVM

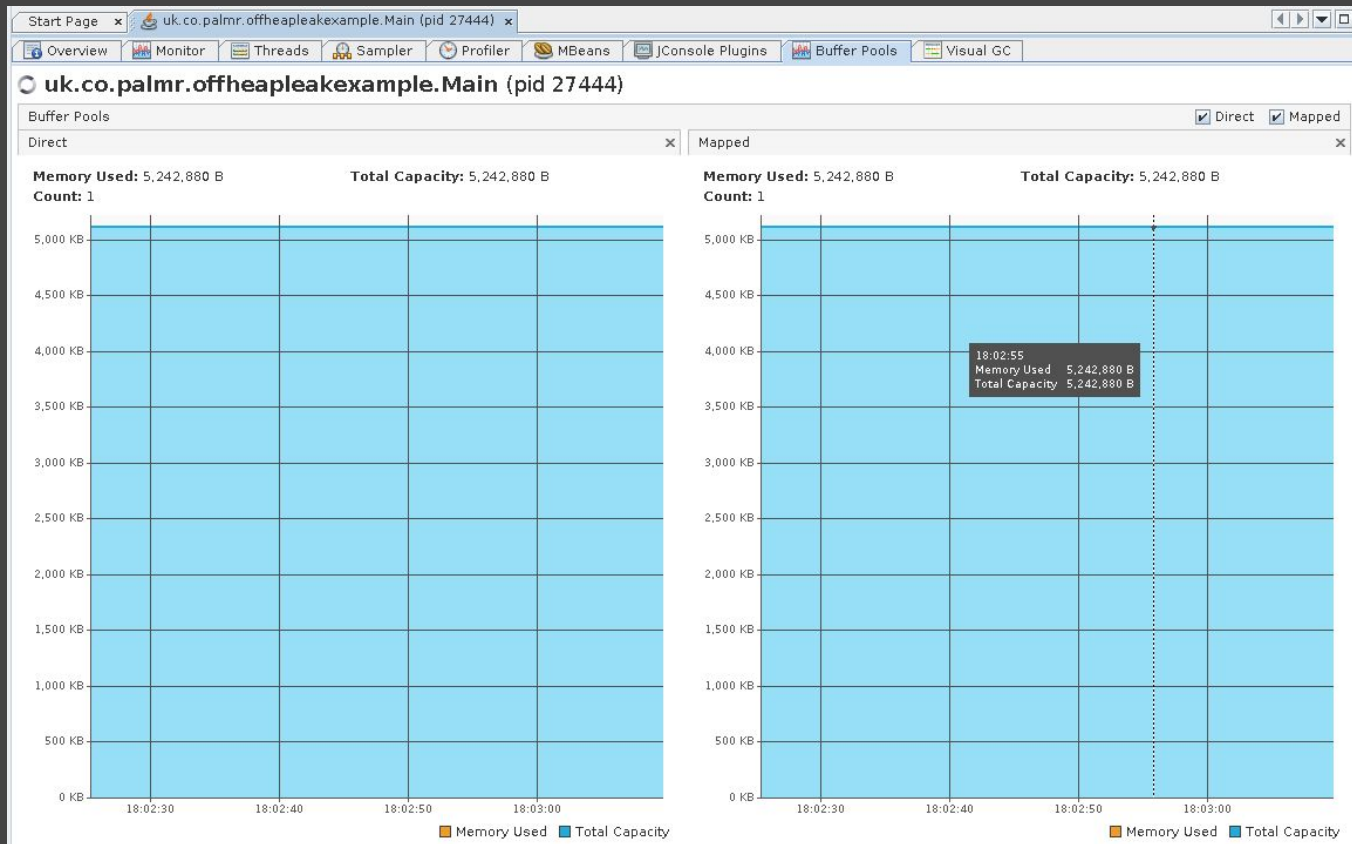
- Java VisualVM is a GUI for monitoring the JVM
- You can see graphs for CPU and memory use
- Plugins let you see GC information, access MBeans and see off-heap buffer sizes
- Take a heap dump and explore it, looking for large or oddly long-lived objects
 - With a heap dump you can look for inflate/deflate instances which might be indicative of unclosed zip streams
- Alternatives tools are available: jconsole, mission control, yourkit, etc.
- <https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jvisualvm.html>



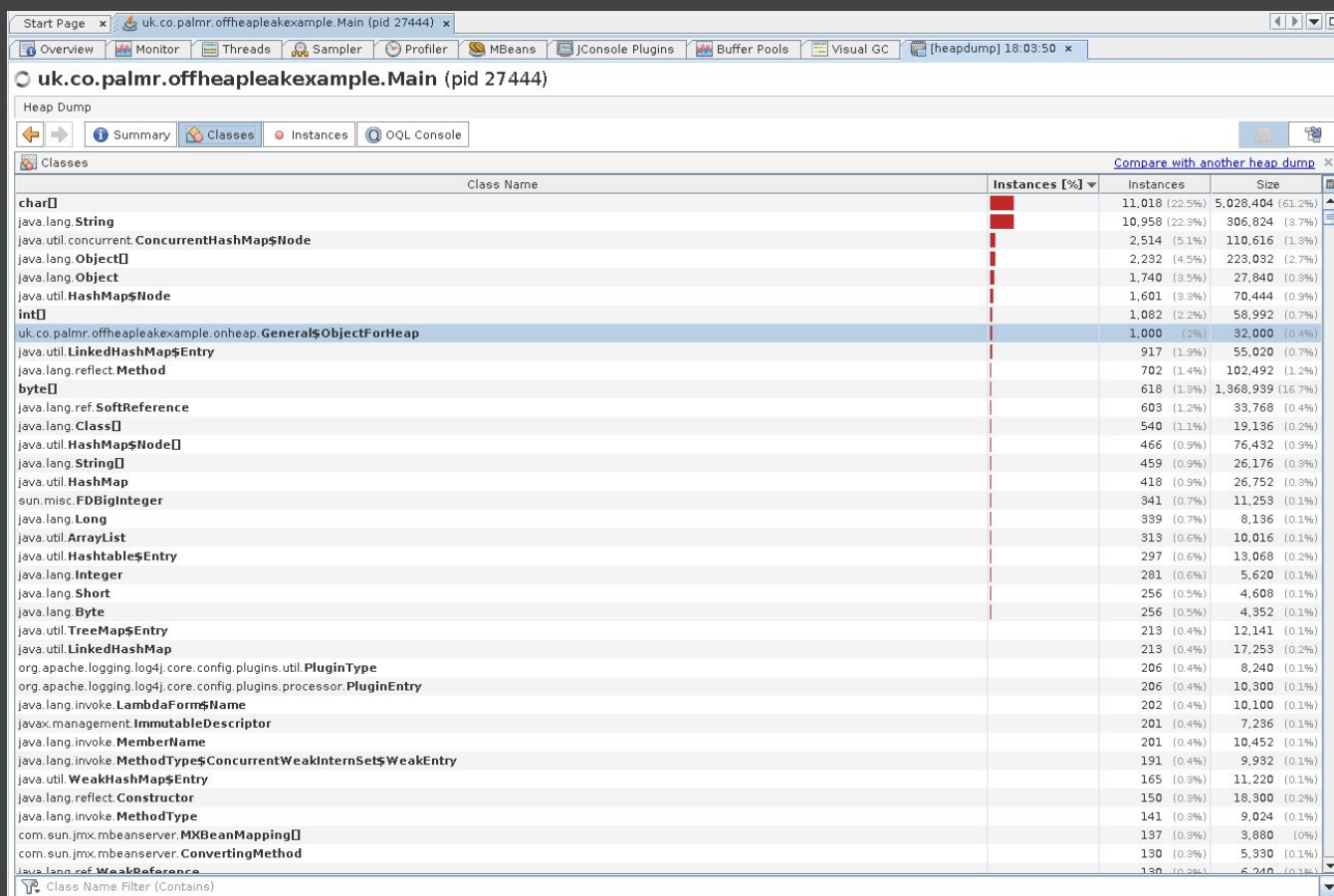
VisualVM: Monitor tab



VisualVM: Visual GC plugin



VisualVM: Buffer pools plugin



Jvisualvm heap dump

Places looked

Heap ✓

Off-heap

Metaspace ✓

Direct Buffers ✓

Zip Steams ✓

Threads

JVM

JNI/JNA



Not there? Time to step it
up a notch





Native Memory Tracking

- NMT is a Java HotSpot VM feature to track internal memory usage of the VM
 - Has a summary and detail mode
 - Has to be enabled via flag at application startup
 - `-XX:NativeMemoryTracking=summary` | `-XX:NativeMemoryTracking=detail`
 - Can get snapshots or diffs from a baseline using the jcmd utility
 - Enabling NMT will result in a 5-10 percent JVM performance drop and memory usage for NMT adds 2 machine words to all malloc memory as malloc header. NMT memory usage is also tracked by NMT.
-
- <https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr007.html>
 - <https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr006.html>

```
palmern@gemini OffHeapLeakExample-1.0-SNAPSHOT (master)$ jcmd 30237 VM.native_memory summary.diff
30237:
```

Native Memory Tracking:

Total: reserved=1472628KB +16978KB, committed=180292KB +18582KB

```
-      Java Heap (reserved=32768KB, committed=32768KB)
              (mmap: reserved=32768KB, committed=32768KB)
-
              Class (reserved=1080638KB +53KB, committed=32702KB +1205KB)
              (classes #1957 +209)
              (malloc=21822KB +53KB #1603 +530)
              (mmap: reserved=1058816KB, committed=10880KB +1152KB)
-
              Thread (reserved=52509KB +11357KB, committed=52509KB +11357KB)
              (thread #52 +11)
              (stack: reserved=52288KB +11308KB, committed=52288KB +11308KB)
              (malloc=161KB +36KB #277 +55)
              (arena=60KB +13 #90 +22)
-
              Code (reserved=250104KB +102KB, committed=5704KB +554KB)
              (malloc=504KB +102KB #1378 +205)
              (mmap: reserved=249600KB, committed=5200KB +452KB)
-
              GC (reserved=25423KB, committed=25423KB)
              (malloc=24219KB #172)
              (mmap: reserved=1204KB, committed=1204KB)
-
              Compiler (reserved=137KB -5KB, committed=137KB -5KB)
              (malloc=6KB -5KB #75 -1)
              (arena=131KB #15)
-
              Internal (reserved=27174KB +5201KB, committed=27174KB +5201KB)
              (malloc=27142KB +5201KB #3635 +550)
              (mmap: reserved=32KB, committed=32KB)
-
              Symbol (reserved=3205KB +182KB, committed=3205KB +182KB)
              (malloc=2110KB +86KB #10100 +206)
              (arena=1095KB +96 #1)
-
      Native Memory Tracking (reserved=440KB +55KB, committed=440KB +55KB)
              (malloc=138KB +25KB #1941 +338)
              (tracking overhead=302KB +30KB)
-
      Arena Chunk (reserved=229KB +33KB, committed=229KB +33KB)
              (malloc=229KB +33KB)
```

jcmd <example-pid> VM.native_memory summary.diff

Places looked

Heap ✓

Off-heap

Metaspace ✓

Direct Buffers ✓

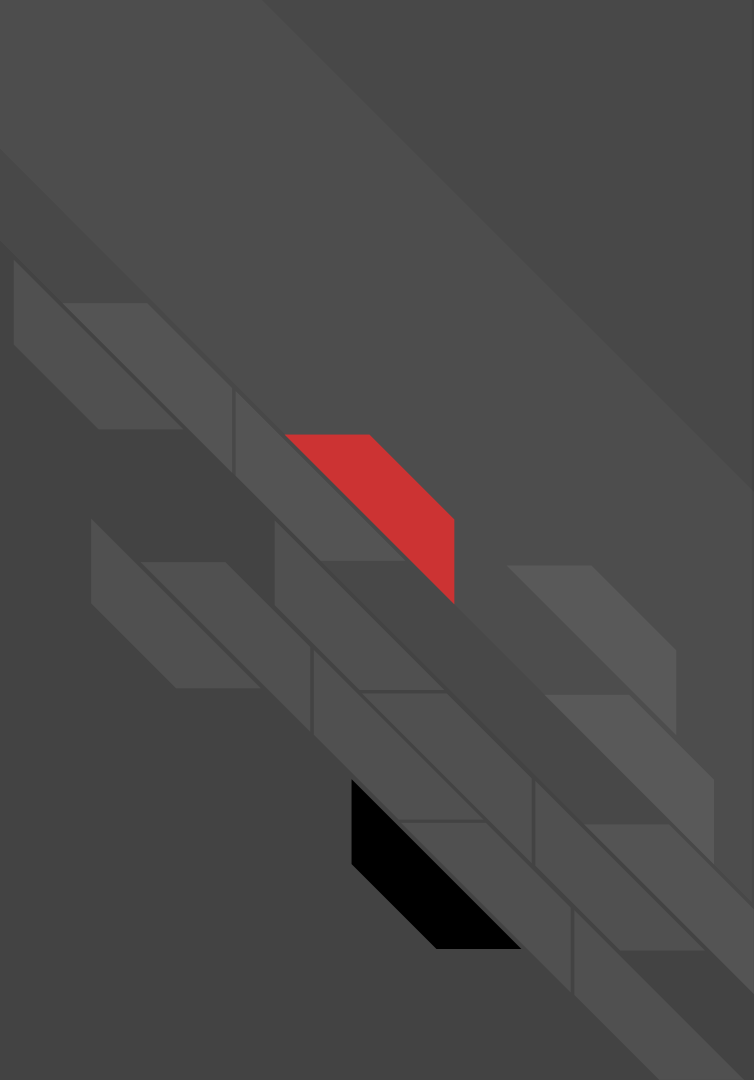
Zip Steams ✓

Threads ✓

JVM ✓ ← kinda

JNI/JNA

Fairly sure it's JNI/JNA,
but how to prove it?

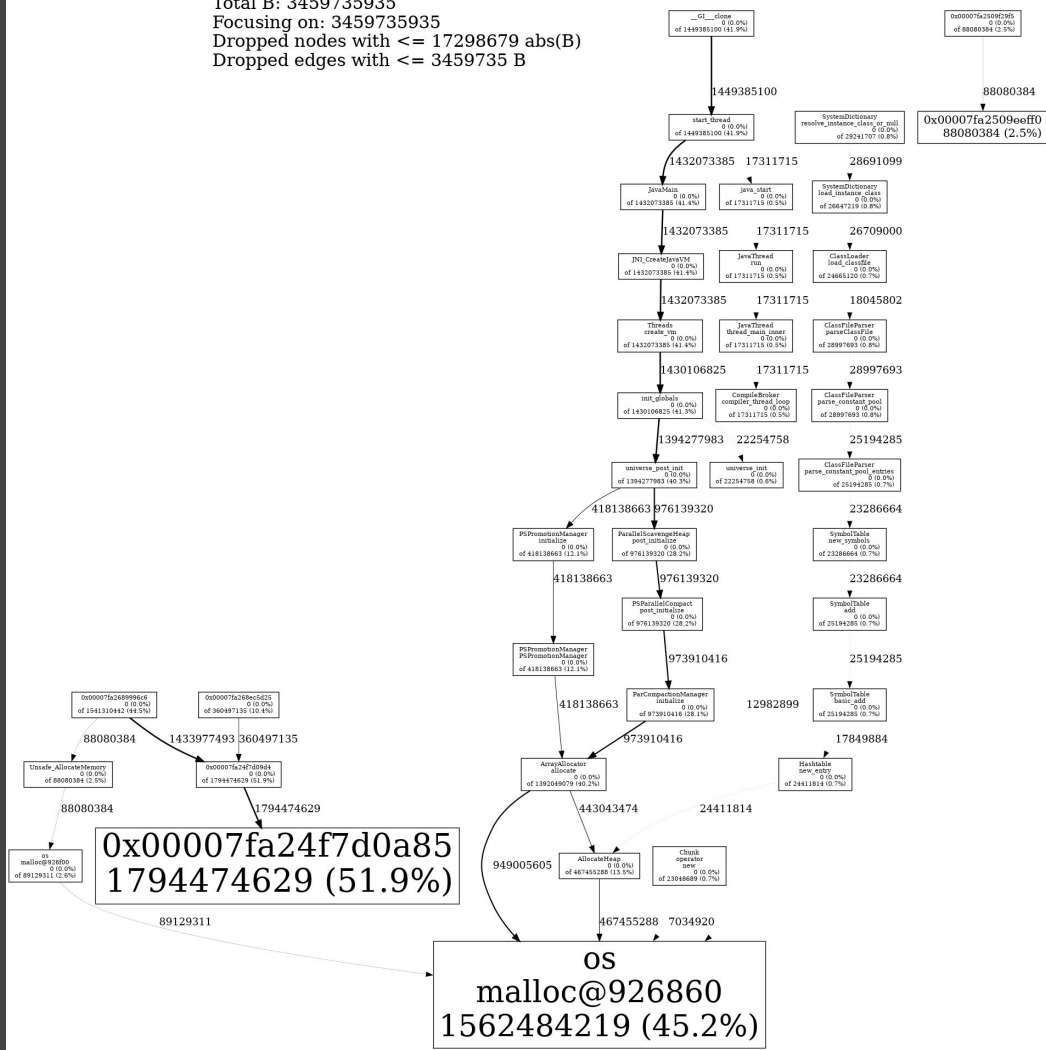




JEMALLOC

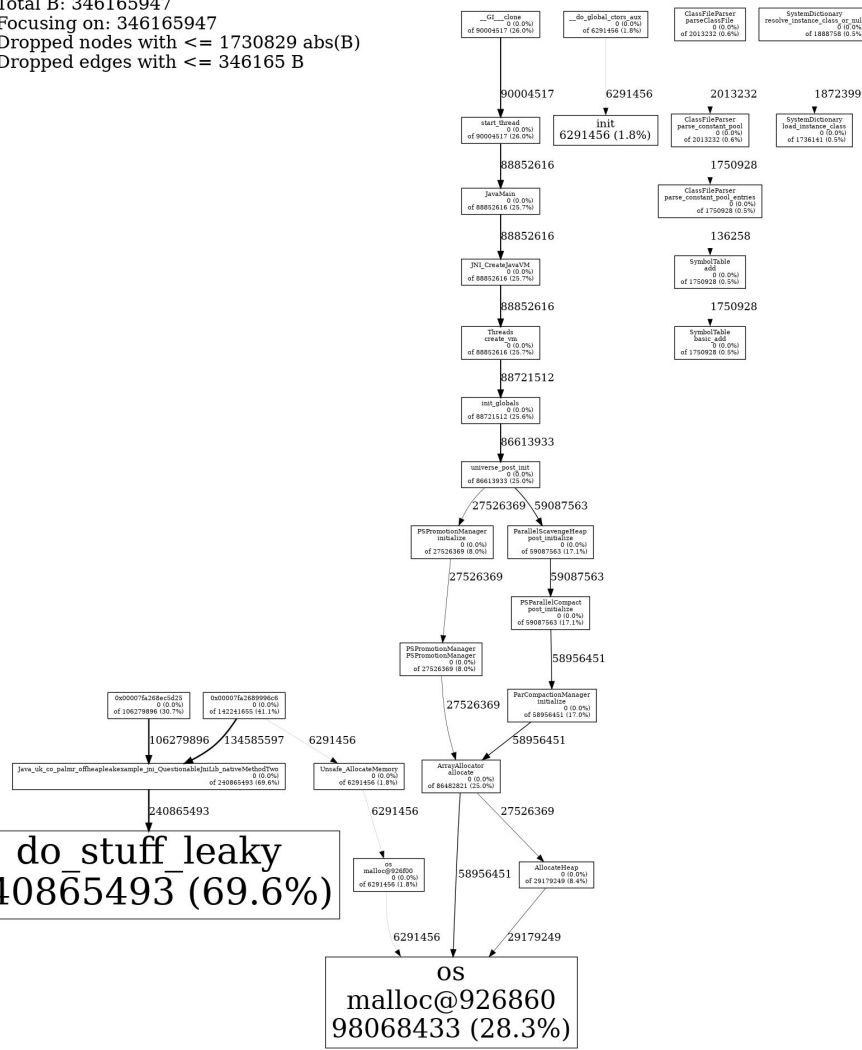
- General purpose malloc implementation from FreeBSD
 - Has profiling options (not on by default, may need to compile yourself)
 - Can be loaded instead of default linux glibc malloc via environment variable
 - Config also passed via environment variable
 - Use jemalloc profiling, memory profile dump every 30 mB, allocation sample every 128 kB
 - `export`
`MALLOC_CONF=prof:true,lg_prof_interval:25,lg_prof_sample:17`
 - Note: interval/sample values are in log base 2 for some reason?
 - Shows what stack traces led to calling malloc
-
- <http://jemalloc.net/>
 - <https://linux.die.net/man/3/jemalloc>
 - <https://github.com/jemalloc/jemalloc/wiki/Use-Case%3A-Leak-Checking>

/home/palmern/projects/buck/opt/jdk/bin/java
 Total B: 3459735935
 Focusing on: 3459735935
 Dropped nodes with <= 17298679 abs(B)
 Dropped edges with <= 3459735 B



jeprof all heap samples

```
/home/palmern/projects/buck-all/opt/jdk/bin/java
Total B: 346165947
Focusing on: 346165947
Dropped nodes with <= 1730829 abs(B)
Dropped edges with <= 346165 B
```



jeprof last heap sample


```
void do_stuff_leaky(char* string)
{
    char* str = (char*) malloc(allocation_amount);
    memset(str, 0xff, allocation_amount);
    strcpy(str, string);
    printf("String = %s, Address = %p\n", str, (void *) str);
    fflush(stdout);

    // Whoops!
    // free(str);
}
```



Whoops!

Conclusion

- Despite being a Java dev with GC lending you a hand, you can still get burned by strange memory leak issues
- There's more tooling in the JVM than just looking at heap stats and dumps
- If you are writing native code, don't forget to free!

Example code: <https://github.com/Palmr/java-off-heap-leak-example>