Out of Memory in Containers

Java Heap (JVM) vs. Native Heap (malloc)

Martijn Vermeulen

adyen

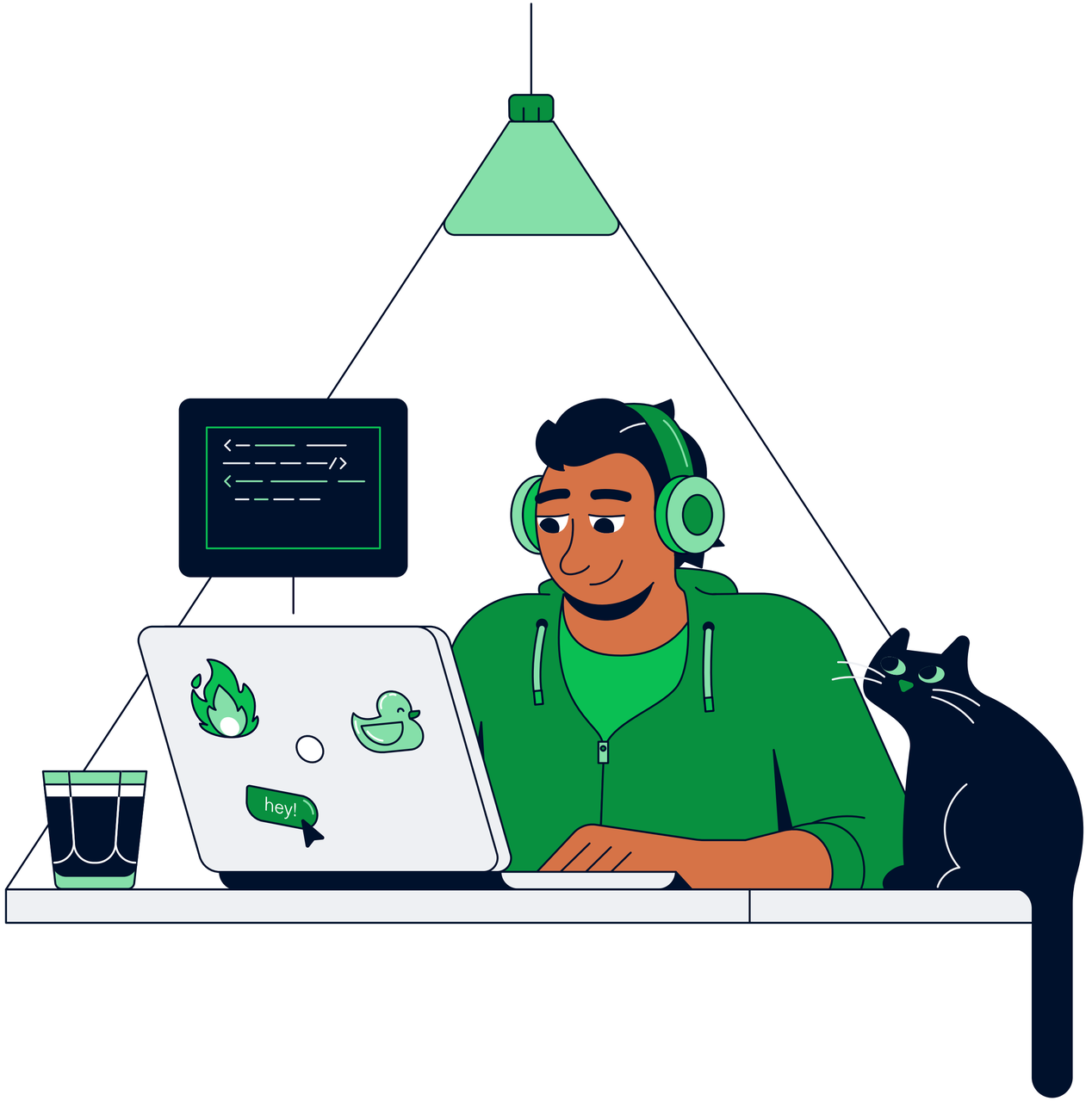
engineered
for ambition

Introduction

adyen

engineered
for ambition

About Adyen



About me

Containers & JVM memory challenges

adyen

engineered
for ambition

Upside of containers:
Isolation

Downside of containers:
Isolation

Upside of the JVM:
Does a lot of memory management for you

Downside of the JVM:
Does a lot of memory management for you

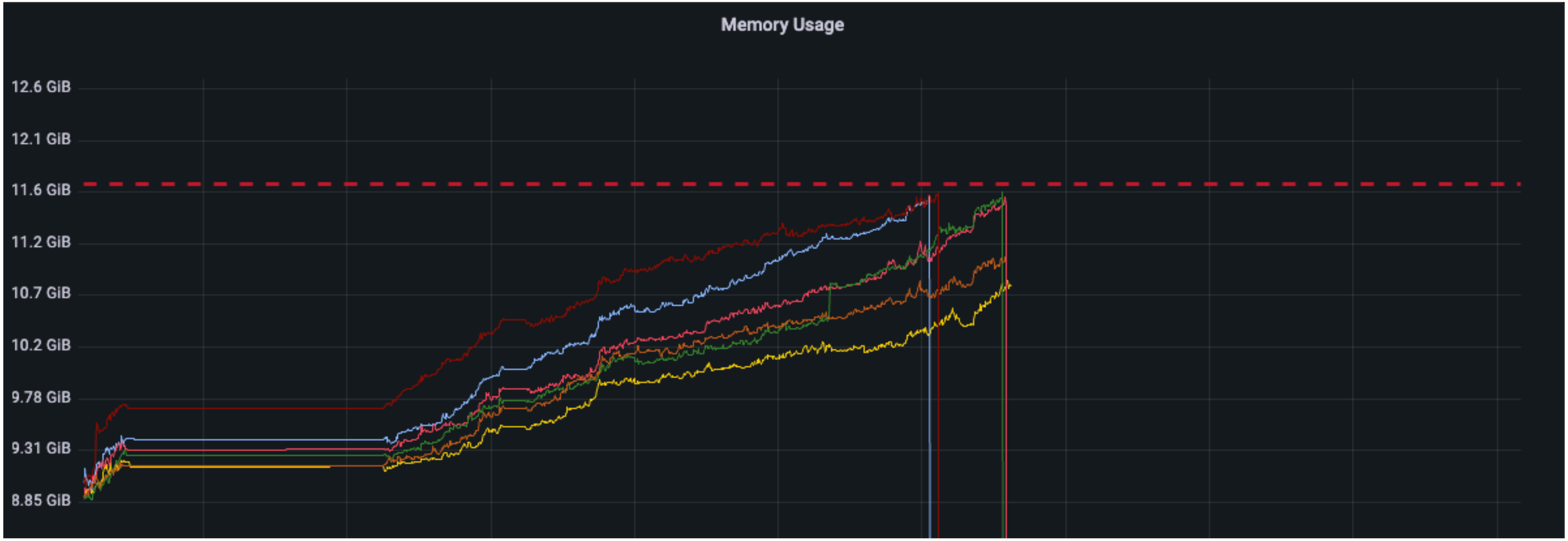
Investigating real issues

adyen

engineered
for ambition

The problem

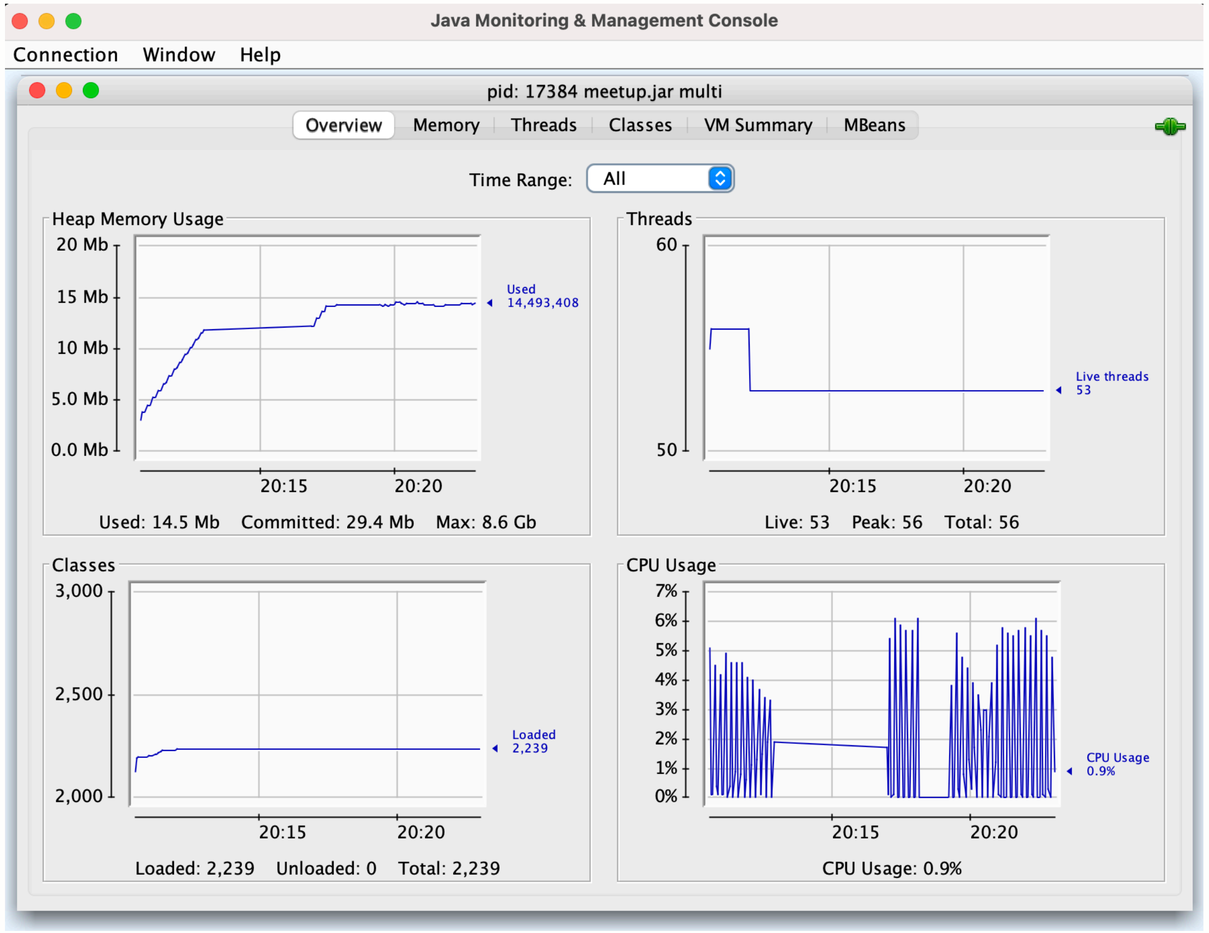
The memory usage

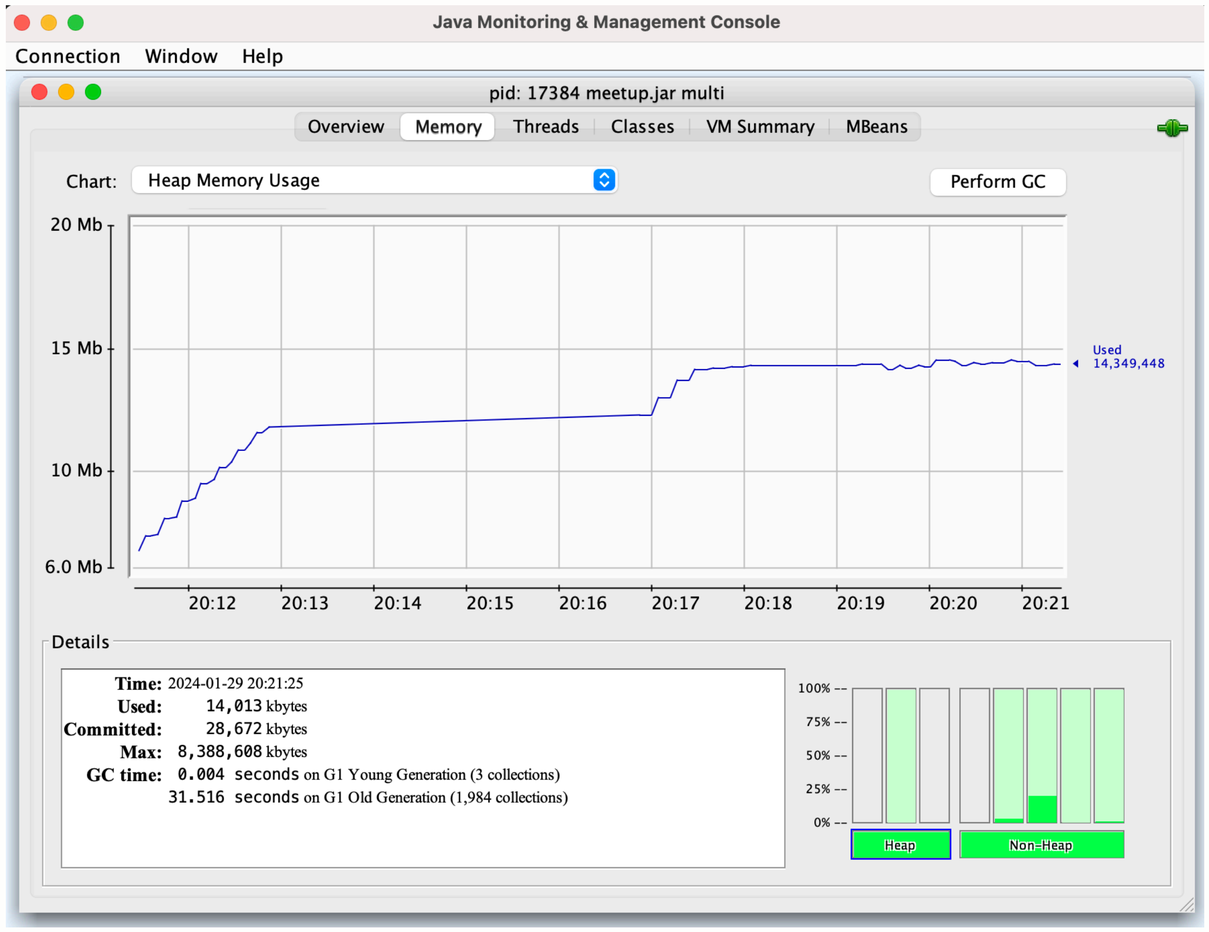


The investigation

- Heap ?
- Threads ?
- Class ?
- Code cache?
- Other processes?

The investigation
JConsole





The investigation
jcmd, jstat, jmap & MAT

- jcmd <pid> GC.heap_info
- jstat -gc <pid>
- jmap -> Eclipse MAT

The investigation
Native Memory Tracking

`summary` / `detail`

Detecting memory leaks
with baseline -> diff

Overhead

The investigation
Native Memory Tracking

Total: reserved, committed

- **malloc**
- **mmap**

- Java Heap
- Class
- Thread
- Code
- GC
- Compiler
- Internal
- Other
- Symbol
- Native Memory Tracking
- Shared class space
- Arena Chunk
- Module
- Safepoint
- Synchronization
- Serviceability
- Metaspace
- String Deduplication
- Object Monitors

Native memory tracking

Summary

```
sh-4.4$ jcmd 1 VM.native_memory summary
1:

Native Memory Tracking:

(Omitting categories weighting less than 1KB)

Total: reserved=2487217KB, committed=81637KB
       malloc: 13057KB #7182
       mmap:   reserved=2474160KB, committed=68580KB

-                 Java Heap (reserved=1048576KB, committed=1024
                            (mmap: reserved=1048576KB, committe

-                     Class (reserved=1048688KB, committed=240K
                            (classes #842)
                            (  instance classes #720, array cla
                            (malloc=112KB #1317)
                            (mmap: reserved=1048576KB, committe
                            (  Metadata:   )
                            (    reserved=8192KB, committed=704
                            (    used=624KB)
                            (    waste=80KB =11.38%)
                            (  Class space:)
                            (    reserved=1048576KB, committed=
                            (    used=36KB)
                            (    waste=92KB =71.73%)
```

Native memory tracking

Summary

```
Shared class space (reserved=12288KB, committed=12116KB)
                   (mmap: reserved=12288KB, committed=12116KB

Arena Chunk (reserved=2329KB, committed=2329KB)
            (malloc=2329KB)

Module (reserved=14KB, committed=14KB)
       (malloc=14KB #402)

Safepoint (reserved=8KB, committed=8KB)
          (mmap: reserved=8KB, committed=8KB)

Synchronization (reserved=35KB, committed=35KB)
                (malloc=35KB #474)

Serviceability (reserved=1KB, committed=1KB)
               (malloc=1KB #6)

Metaspace (reserved=8203KB, committed=715KB)
          (malloc=11KB #9)
          (mmap: reserved=8192KB, committed=704KB)

String Deduplication (reserved=1KB, committed=1KB)
                     (malloc=1KB #8)

Object Monitors (reserved=2KB, committed=2KB)
                (malloc=2KB #8)
```

Native memory tracking

Baseline

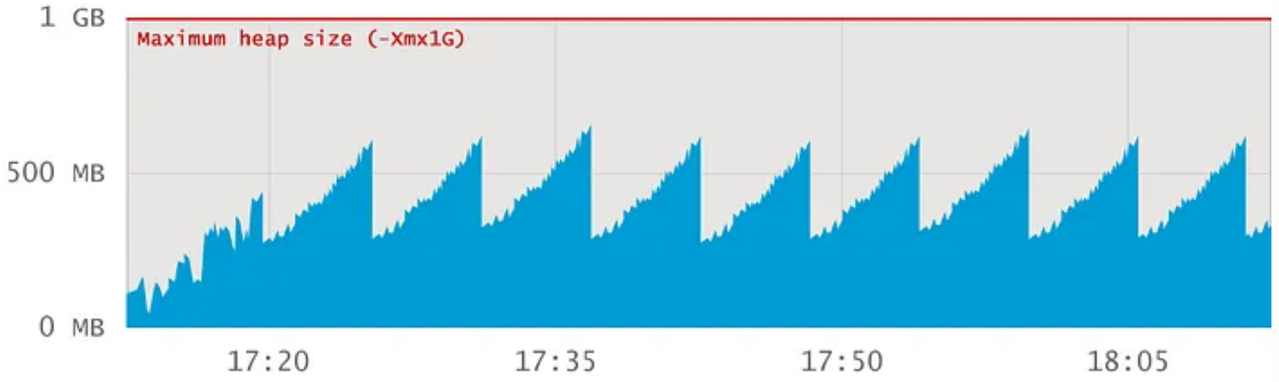
vs.

Diff (compared to the baseline)

```
sh-4.4$ jcmd 1 VM.native_memory baseline
1:
Baseline succeeded
sh-4.4$ jcmd 1 VM.native_memory summary.diff
1:

Native Memory Tracking:

(Omitting categories weighting less than 1KB)

Total: reserved=2469091KB +675KB, committed=76171KB +675KB

-                 Java Heap (reserved=1048576KB, committed=1
                            (mmap: reserved=1048576KB, commi

-                     Class (reserved=1048676KB, committed=2
                            (classes #791)
                            (  instance classes #673, array
                            (malloc=100KB #1086 +4)
                            (mmap: reserved=1048576KB, commi
```

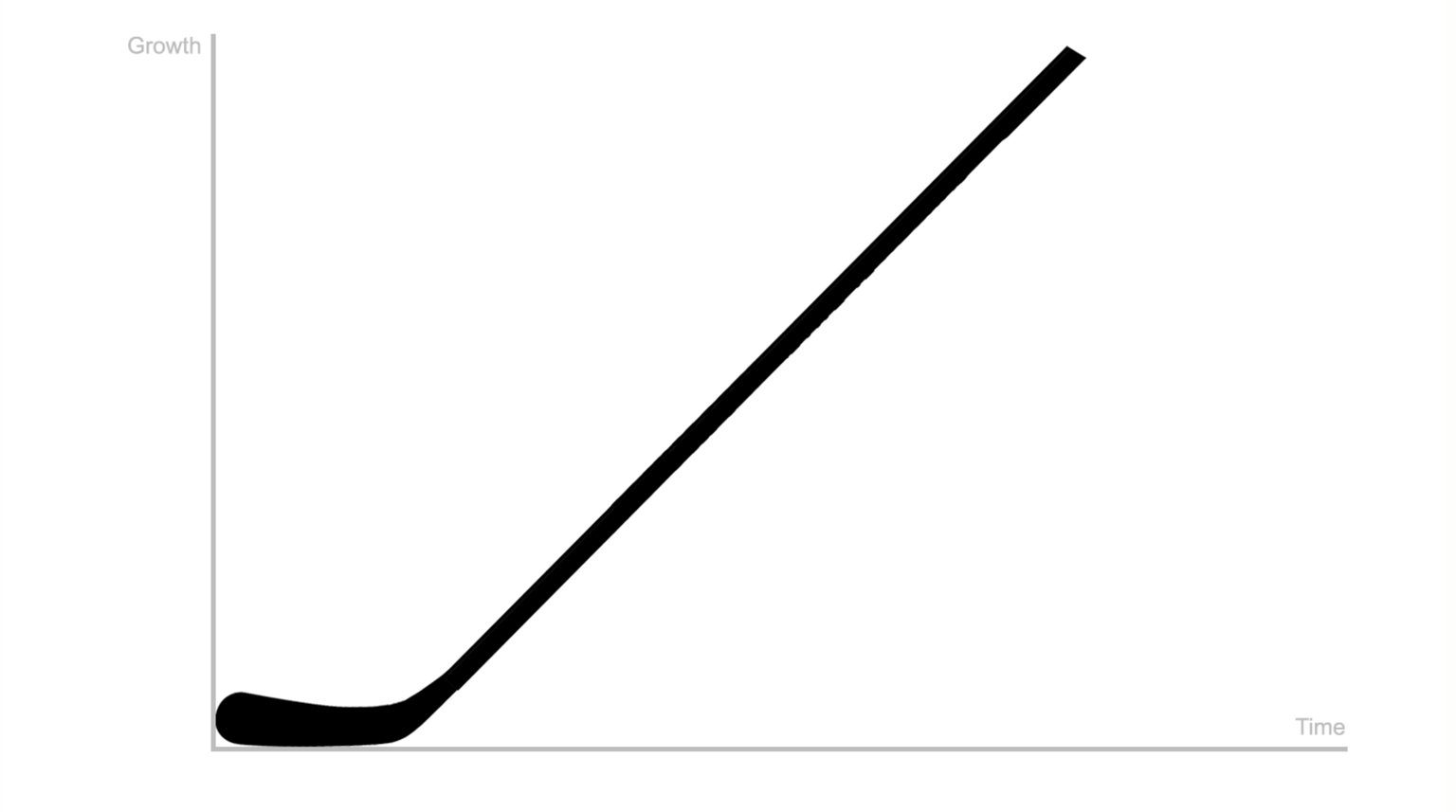
The investigation
JVM related memory usage

https://medium.com/medical-myths-and-models/can-you-shock-a-heart-that-has-flatlined-d156a12a898c



https://medium.com/@plumbr/memory-leaks-fallacies-and-misconce-8c79594a3986

The investigation
top & docker stats



https://www.ecosystemsandentrepreneurs.com/blog/2020/1/17/the-hockey-stick-growth-slide-is-broken-heres-a-better-one

The investigation

- Heap ? ❌
- Threads ? ❌
- Class ? ❌
- Code cache? ❌
- Other processes?

The investigation

- Heap ? ❌
- Threads ? ❌
- Class ? ❌
- Code cache? ❌
- Other processes? ❌

The culprit?

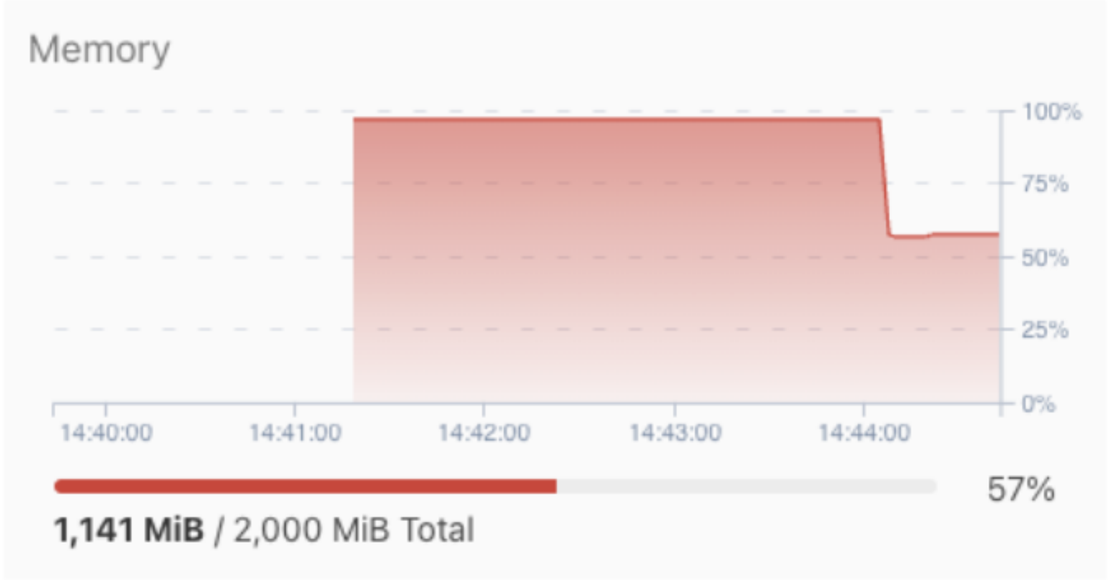
- Trying stuff...
- Google

https://tenor.com/view/typing-cat-busy-writing-chatting-gif-15923952

The culprit!

- System.trim_native_heap?
- Native heap overhead! ✅

```
bash-4.4$ jcmd 1 System.trim_native_heap
1:
Attempting trim...
Done.
Virtual size before: 10757268k, after: 10757140k, (-128k)
RSS before: 2009392k, after: 1191544k, (-817848k)
Swap before: 0k, after: 0k, (0k)
```



The culprit!

- **Native heap overhead?**

Surprised Engineer Stock Image
https://www.shutterstock.com/search/surprised-engineer

Native heap / malloc

“But Martijn, we are at a Java Meetup?”

I know!

JVM & Native heap

Java heap

- Java heap is allocated using mmap (mostly)
  - Max size is preallocated as one contiguous area
- JVM manages this memory directly

JVM & Native heap

Native heap

- The C way
- Allocated by using the underlying malloc and free

JVM & Native heap

Native heap

- GUI
- Buffers for data compression routines
- Compiled code generated by JIT Compiler
- Threads to map to Java threads
- Malloc allocations by application JNI code

JVM & Native heap
Malloc in JNI code

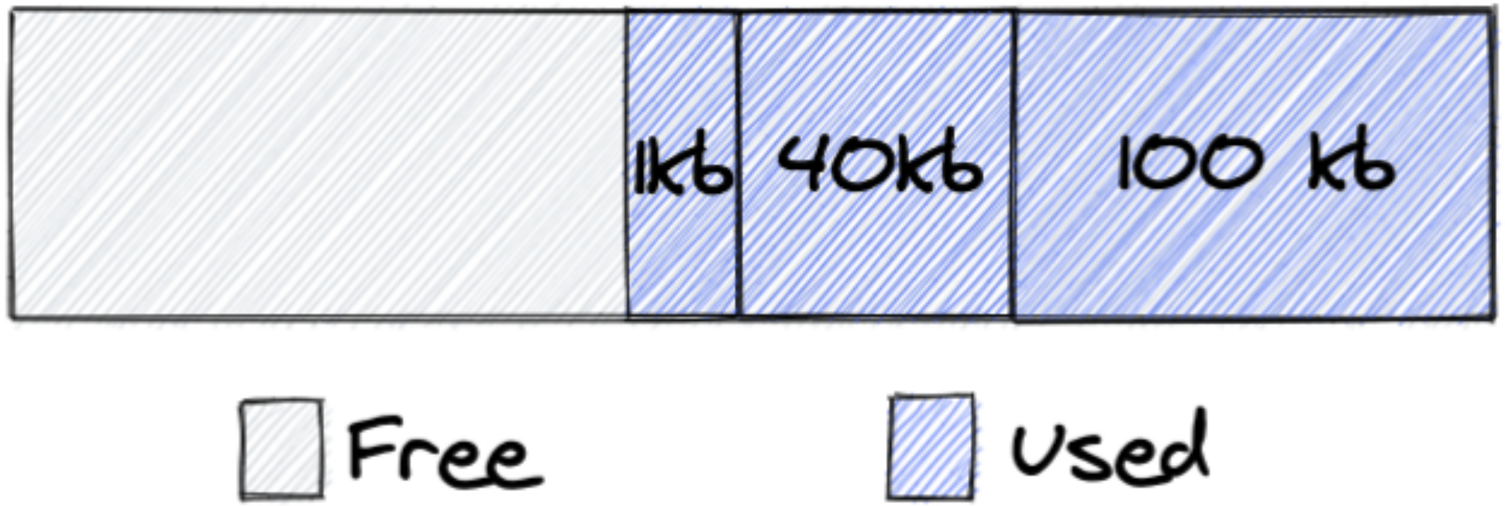
- java.nio.DirectByteBuffer
- sun.misc.Unsafe
- Any native code potentially

malloc & free

- malloc
  - allocate x bytes of memory and return address
- free
  - frees the memory space at given address

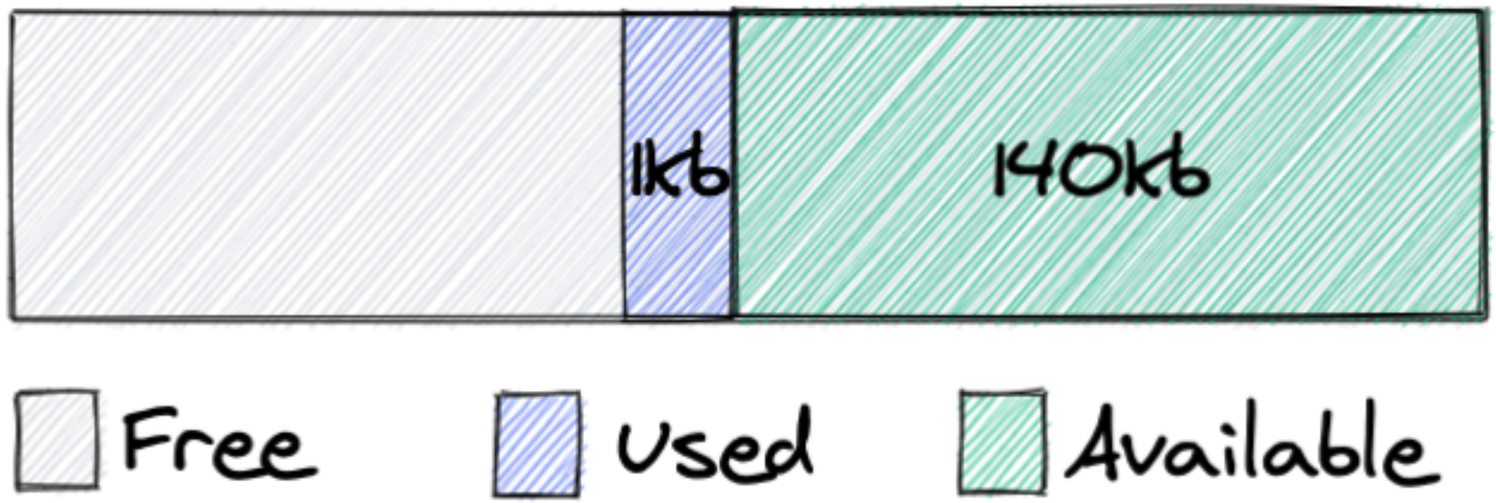
malloc & free

- malloc(100kb) -> addr1
- malloc(40kb) -> addr2
- malloc(1kb) -> addr3



malloc & free

- free(addr1)
- free(addr2)



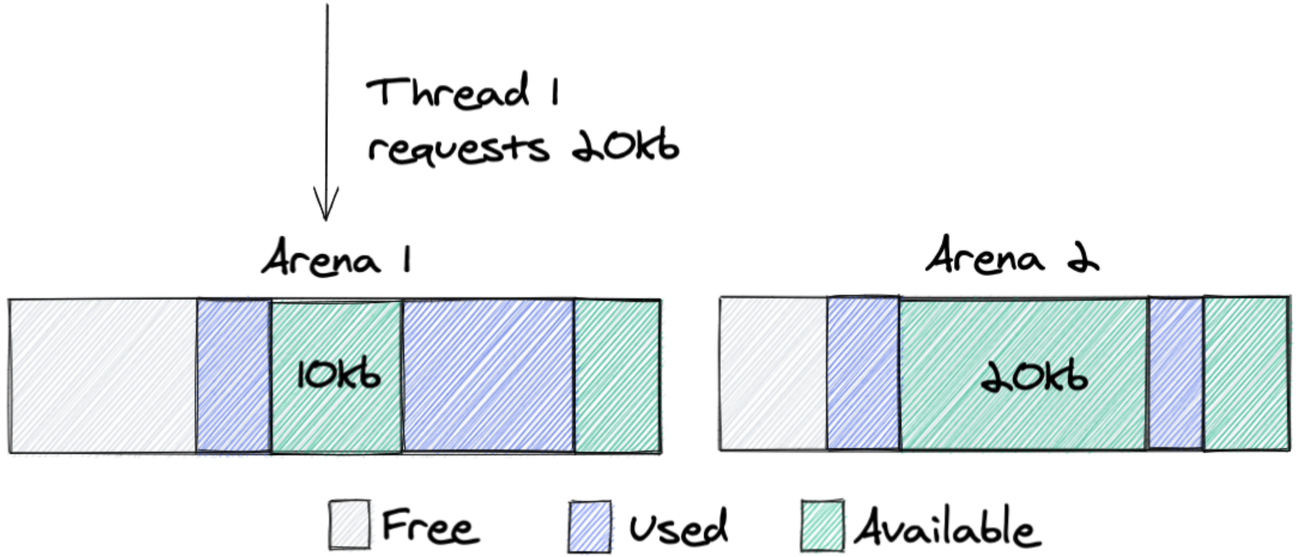
glibc

- GNU Project's implementation of the C standard library
- Default implementation of many Linux distros

glibc
malloc

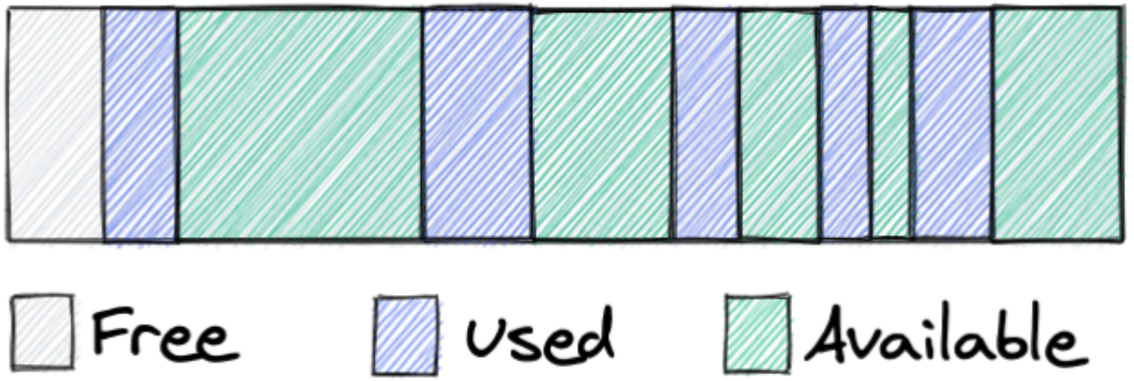
- Uses multiple ‘arena’s to reduce locking
- One arena can contain multiple ‘heaps’ underneath
- By default, maximum of 8 * # cores arenas

glibc
malloc



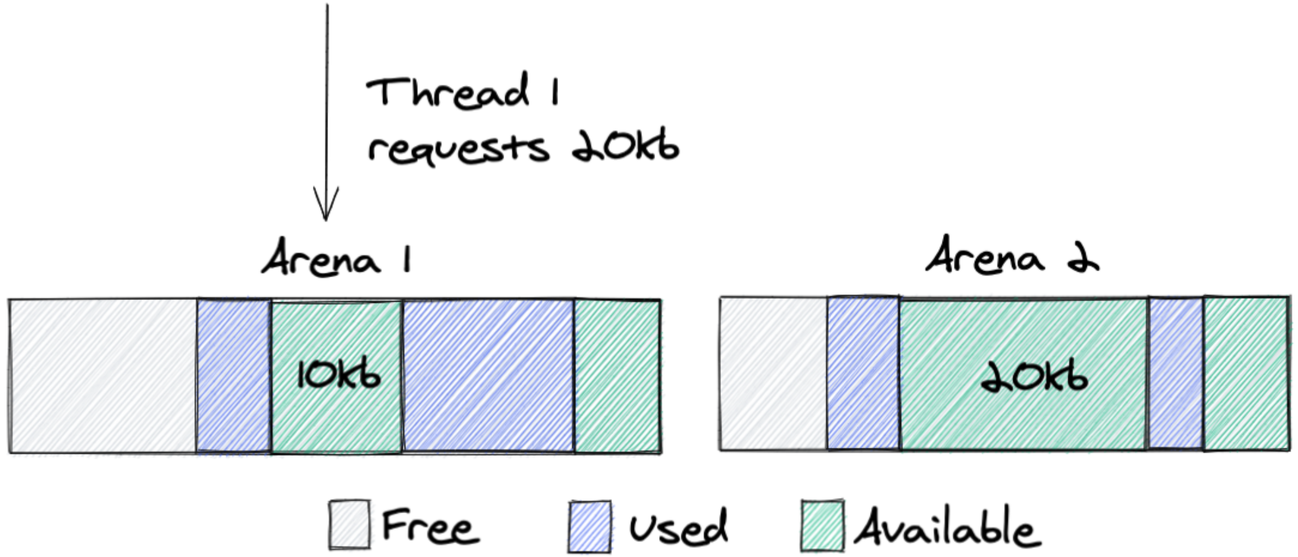
https://blog.cloudflare.com/the-effect-of-switching-to-tcmalloc-on-rocksdb-memory-use

Memory fragmentation



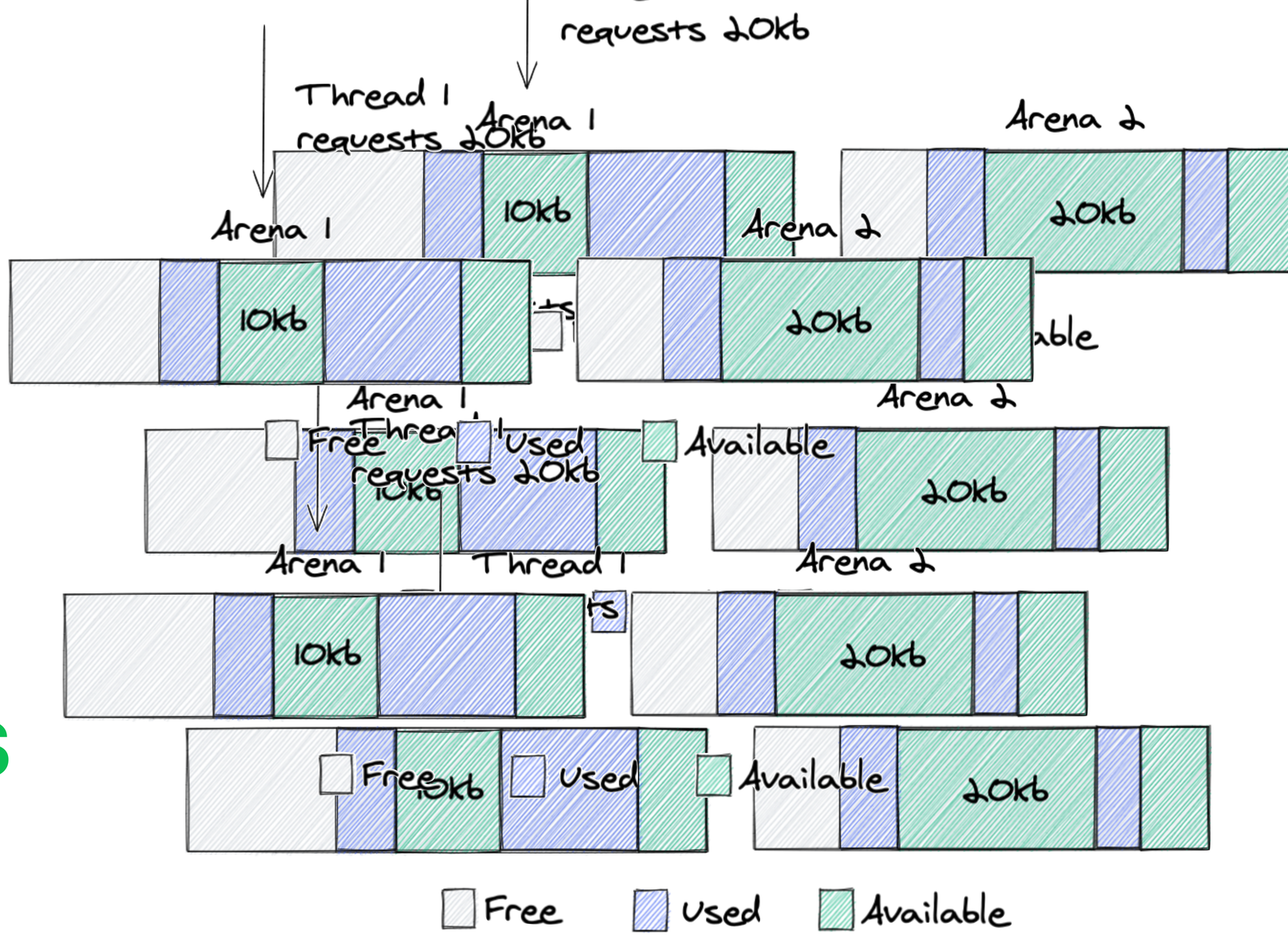
https://blog.cloudflare.com/the-effect-of-switching-to-tcmalloc-on-rocksdb-memory-use

glibc
malloc



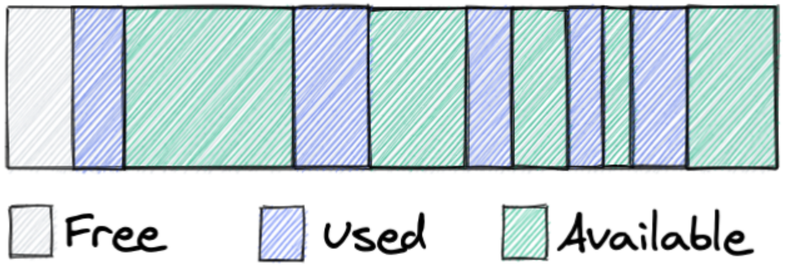
https://blog.cloudflare.com/the-effect-of-switching-to-tcmalloc-on-rocksdb-memory-use

glibc
malloc
+
Many threads

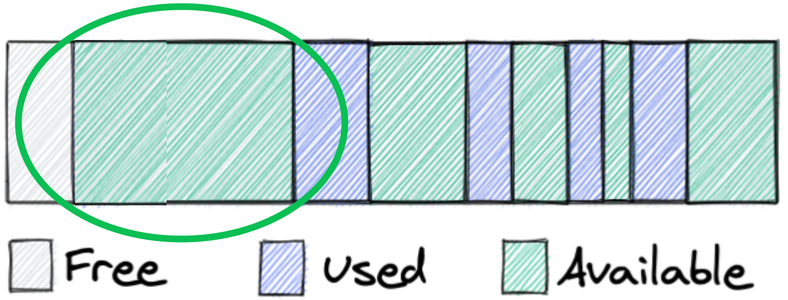


https://blog.cloudflare.com/the-effect-of-switching-to-tcmalloc-on-rocksdb-memory-use

glibc
malloc
+
Releasing memory



Will not release back



Will potentially release back

Solving the growing memory issue

adyen

engineered
for ambition

glibc
Memory Allocation Tunables

- MALLOC_ARENA_MAX
- M_MMAP_THRESHOLD
- MALLOC_TRIM_THRESHOLD_
- Amongst others

https://www.gnu.org/software/libc/manual/2.28/html_node/Memory-Allocation-Tunables.html

glibc
Memory Allocation Tunables

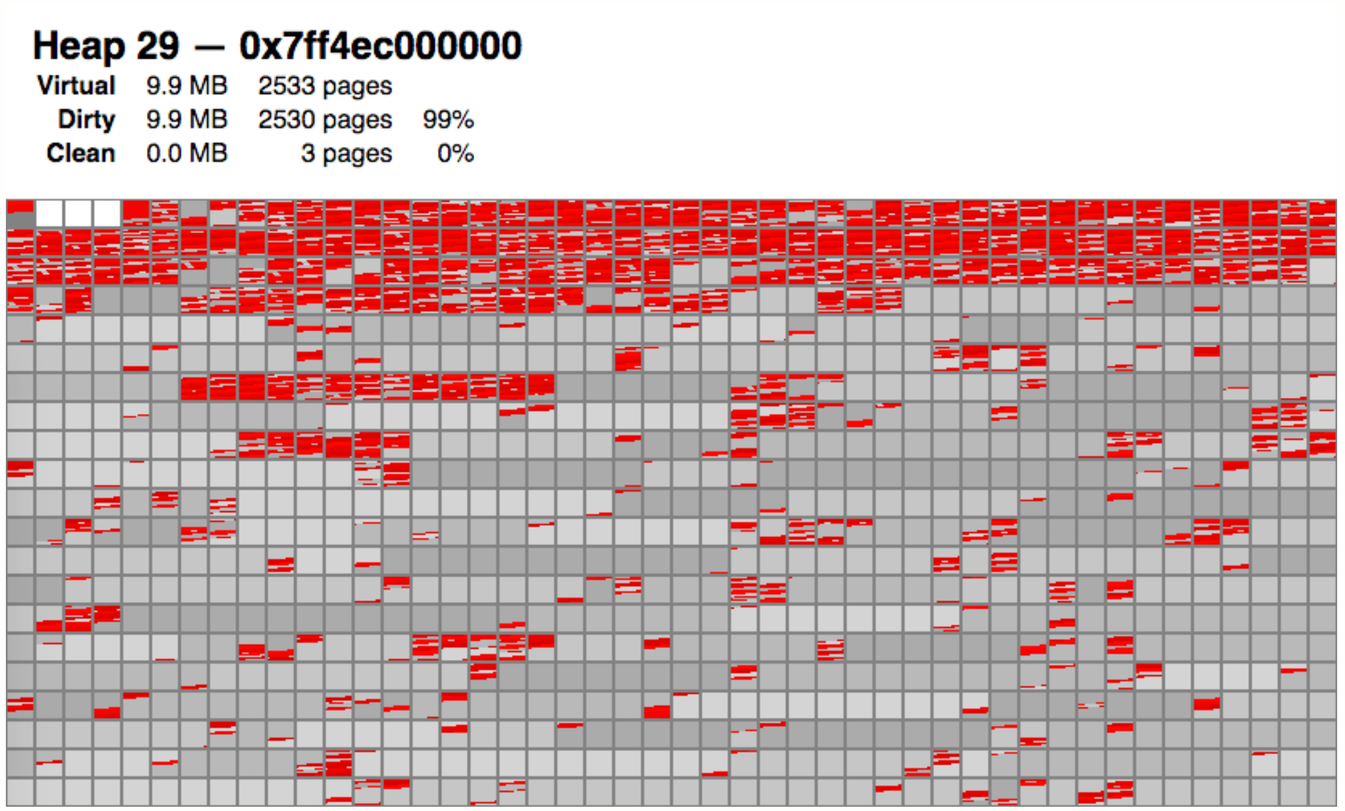
- MALLOC_ARENA_MAX
  - The most used tunable, set to 2-4
  - Widely accepted as ‘the’ fix (next to switching malloc implementation)

https://www.gnu.org/software/libc/manual/2.28/html_node/Memory-Allocation-Tunables.html

glibc
malloc_trim

- From the manual:
  - The malloc_trim() function attempts to release free memory at the top of the heap.
- The manual is wrong!

glibc
malloc_trim

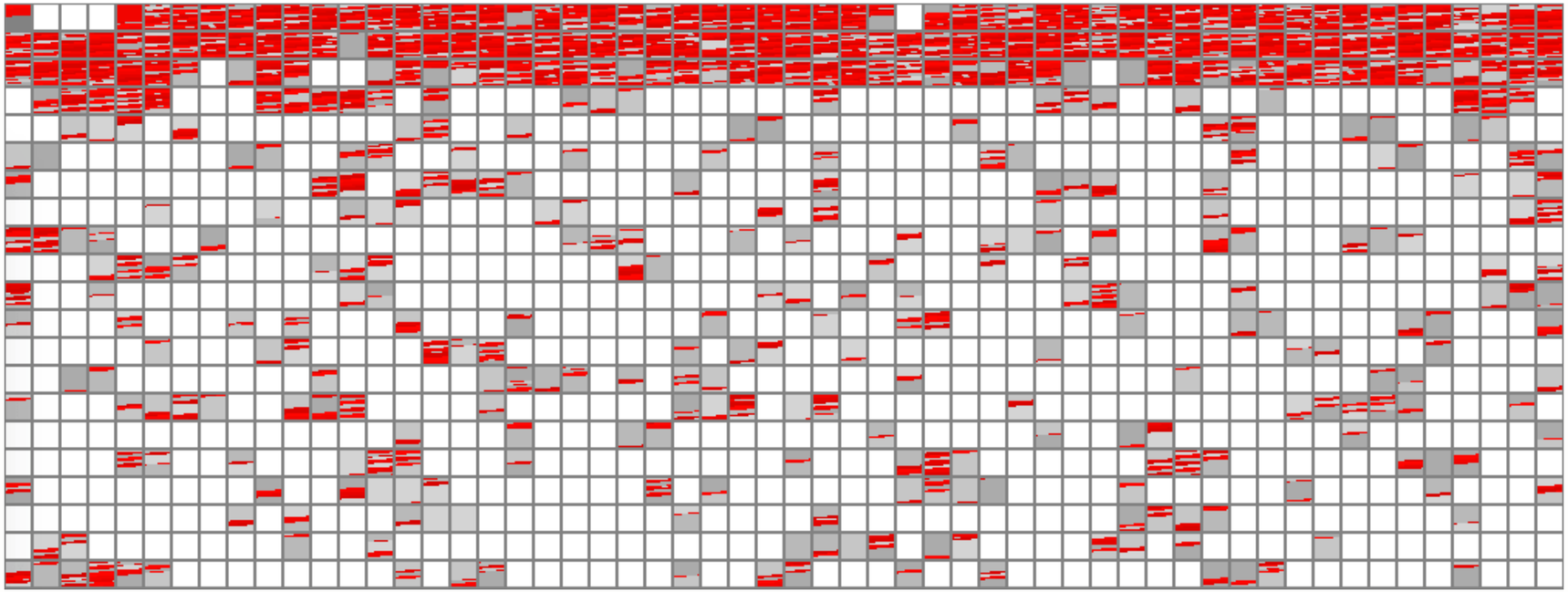


Hongli Lai - https://www.joyfulbikeshedding.com/blog/2019-03-14-what-causes-ruby-memory-bloat.html

glibc
malloc_trim

Heap 10 — 0x7f10b0000000

Virtual	10.7 MB	2750 pages	
Dirty	2.7 MB	689 pages	25%
Clean	8.1 MB	2061 pages	74%

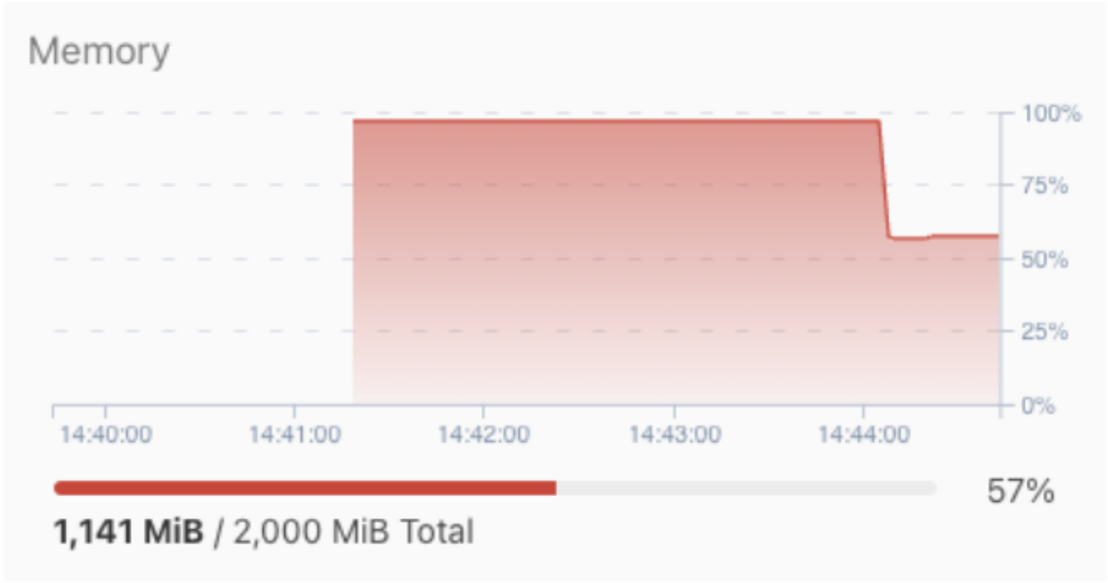


Hongli Lai - https://www.joyfulbikeshedding.com/blog/2019-03-14-what-causes-ruby-memory-bloat.html

glibc
malloc_trim

```
bash-4.4$ jcmd 1 System.trim_native_heap
1:
Attempting trim...
Done.
Virtual size before: 10757268k, after: 10757140k, (-128k)
RSS before: 2009392k, after: 1191544k, (-817848k)
Swap before: 0k, after: 0k, (0k)
```

System.trim_native_heap



glibc
malloc_trim

Overhead!

Does a lot of system calls, which is expensive

Other malloc implementations

- jemalloc
- mimalloc
- tcmalloc
- Many others...
- Your own?

- Using LD_PRELOAD

Solving the growing memory issue

- Always test properly for your application!
- Depending on the nature, performance might be affected

Demo time

https://github.com/MartijnVermeulen96/MeetupMemory

adyen

engineered
for ambition

The end
Discussion time (/beers)

Some references:

malloc(3) Linux manual page: https://man7.org/linux/man-pages/man3/malloc.3.html

glibc Memory Allocation Tunables: https://www.gnu.org/software/libc/manual/html_node/Memory-Allocation-Tunables.html

Heroku on tuning glibc malloc, setting MALLOC_ARENA_MAX to 2: https://devcenter.heroku.com/articles/tuning-glibc-memory-behavior

Picnic blogpost on malloc: https://blog.picnic.nl/quest-to-the-os-java-native-memory-5d3ef68ffc0a

LinkedIn switching to Jemalloc: https://www.linkedin.com/blog/engineering/infrastructure/taming-memory-fragmentation-in-venice-with-jemalloc

PR on openjdk “JVM should trim the native heap”: https://github.com/openjdk/jdk/pull/14781

IBM mentioning setting MALLOC_ARENA_MAX to 4: https://www.ibm.com/docs/en/SSFHA8_1.2.0/troubleshooting_faq/malloc_tuning.html

Some references:

Mimalloc: https://github.com/microsoft/mimalloc

Jemalloc: https://github.com/jemalloc/jemalloc

Tcmalloc: https://github.com/google/tcmalloc

StackOverflow for LD_PRELOAD trick: https://stackoverflow.com/questions/426230/what-is-the-ld-preload-trick

Youtube video - Memory footprint of a Java process by Andrei Pangin: https://www.youtube.com/watch?v=c755fFv1Rnk