LINUX MEMORY EXPLAINED

accentureoperations

FRITS HOOGLAND

ENVIRONMENT

Virtual machine: VirtualBox 5.1.28 r117968

Operating system: Oracle Linux 7.3
Kernel: 4.1.12-94.5.9.el7uek.x86_64
Oracle database: 12.2.0.1.170814

Amount of memory: 4GB

BOOT

An intel based system first starts it's firmware, which is either BIOS or UEFI.

The firmware tries to boot. When it boots from disk:

- **BIOS: scans disks for a MBR, which loads a boot loader: grub / grub2.**
- **UEFI: scans disk for GUID partition table, read system partition, which executes boot loader: grub2.**

Grub(2) loads the linux kernel into memory:

```
# dmesg | grep Command
[    0.000000] Command line: BOOT_IMAGE=/
vmlinuz-4.1.12-94.3.4.el7uek.x86_64 root=/dev/mapper/ol-root ro
crashkernel=auto rd.lvm.lv=ol/root rd.lvm.lv=ol/swap rhgb quiet numa=off
transparent_hugepage=never
```

BOOT

Of course this uses some memory:

```
# dmesg | grep Memory
[    0.000000] Memory: 3743704K/4193848K available (7417K kernel code,
1565K rwdata, 3812K rodata, 1888K init, 3228K bss, 433760K reserved,
16384K cma-reserved)
```

This means the kernel left approximately 3.6G of the 4.0G of PHYSICAL memory for use after loading.

This can be verified by looking at /proc/meminfo:

```
# head -1 /proc/meminfo
MemTotal:        3781460 kB
```

VIRTUAL MEMORY

Linux is an operating system that provides virtual memory.

- This means the addresses the processes see, are virtual memory addresses which are unique to that process.
  - This is called an ‘address space’.
  - All memory allocations are only visible to the process that allocated these.
- The translation from virtual memory addresses to physical addresses is done by hardware in the CPU, using what is called the ‘memory management unit’ or MMU.
  - This is why specialistic memory management, like hugepages, is not only dependent on operating system support, but also on hardware support.
- The management of physical memory is done by the operating system.
  - This is why swapping occurs by an operating system process.

ADDRESS SPACE

A linux 64 bit process address space consists of the full 2^64 addresses:

```
# cat /proc/self/maps
00400000-0040b000 r-xp 00000000 f9:00 67398280                           /usr/bin/cat
0060b000-0060c000 r--p 0000b000 f9:00 67398280                           /usr/bin/cat
0060c000-0060d000 rw-p 0000c000 f9:00 67398280                           /usr/bin/cat
023c4000-023e5000 rw-p 00000000 00:00 0                                  [heap]
…lot's of other output…
7ffc4880e000-7ffc4882f000 rw-p 00000000 00:00 0                          [stack]
7ffc48840000-7ffc48842000 r--p 00000000 00:00 0                          [vvar]
7ffc48842000-7ffc48844000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]

# echo "obase=16;2^64-1" | bc
FFFFFFFFFFFFFFFF
```

That’s 16 exa bytes! For every process!

PHYSICAL MEMORY LIMITATIONS

Please mind the 16 exabytes is virtual memory!

Physical memory is limited by multiple factors.

First factor is physical implementation of memory addressing on the CPU:

```
# grep address\ size /proc/cpuinfo
address sizes      : 46 bits physical, 48 bits virtual
address sizes      : 46 bits physical, 48 bits virtual
address sizes      : 46 bits physical, 48 bits virtual
address sizes      : 46 bits physical, 48 bits virtual
```

This is saying the linux kernel detects a CPU with 46 bits for addressing memory.

```
# echo "2^46" | bc
70368744177664
```

That's 64 terabyte. Per CPU socket.

PHYSICAL MEMORY LIMITATIONS

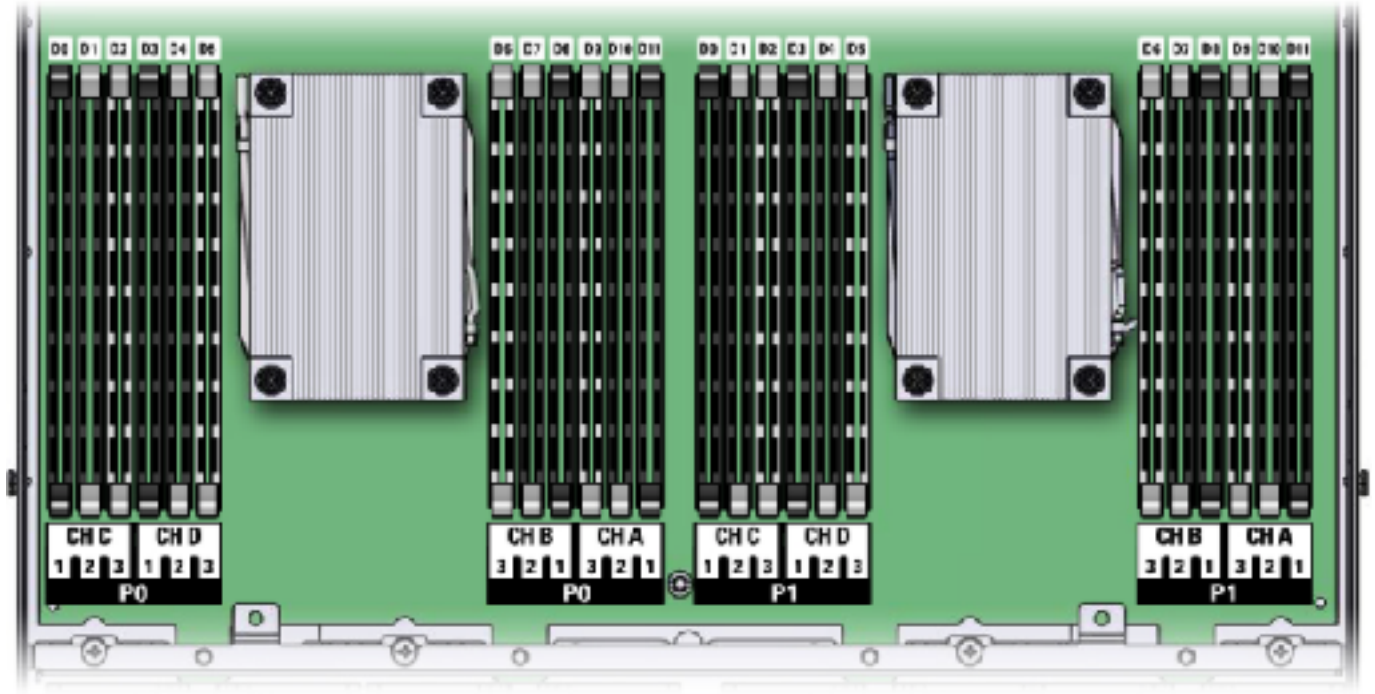
64 terabyte is a lot. In a (typical) 2 socket setup, that is 128 terabyte of memory potentially addressable.

The second limitation is the actual major limitation on physical memory.

That is the on-die memory controller supported physical memory types and configuration, and the physical hardware motherboard setup.

PHYSICAL MEMORY LIMITATIONS

This is the DIMM and processor physical layout from the X6-2 service manual:



It has got 12 DIMM slots per socket, and the manual describes it supports DIMMs up to 64G in size. This means that the maximal amount of physical memory is: 64*12*2=1536G = 1.5T!

MEMORY: PHYSICAL AND VIRTUAL

Okay. So far we got:

Every linux process has its own address space of 16 exabytes of addressable virtual memory.

A typical current Intel Xeon CPU has 46 bits physically available for memory addressing, which means every CPU can theoretically address 64T.

A system is limited in physical memory by memory module size, number and populating rules dictated by the on die memory manager(s) (type of memory, number of channels and number of slots per channel)*, and the physical memory motherboard configuration.

On an Oracle X6-2 system this is maximally 1.5T. Same for X7-2.

EXECUTABLE MEMORY USAGE

What happens when an executable is executed in a shell on linux?

Common thinking is:

- **The process is cloned.**
- **Loads the executable.**
- **Executes it.**
- **Returns the return code to the parent process.**

Memory needed = executable size.

Which very roughly is what happens, but there's a lot more to it than that!
…and memory size that is needed is commonly NOT executable size!

EXECUTABLE MEMORY USAGE

A linux ELF executable is not entirely loaded into memory and then executed.

For the ELF executable itself, some memory regions are allocated:

```
# less /proc/self/maps
00400000-00421000 r-xp 00000000 f9:00 67316594                           /usr/bin/less
00620000-00621000 r--p 00020000 f9:00 67316594                           /usr/bin/less
00621000-00625000 rw-p 00021000 f9:00 67316594                           /usr/bin/less
00625000-0062a000 rw-p 00000000 00:00 0
0255d000-0257e000 rw-p 00000000 00:00 0                                  [heap]
7f5149f59000-7f5150482000 r--p 00000000 f9:00 587351                     /usr/lib/locale/locale-archive
7f5150482000-7f5150639000 r-xp 00000000 f9:00 134334316                  /usr/lib64/libc-2.17.so
7f5150639000-7f5150838000 ---p 001b7000 f9:00 134334316                  /usr/lib64/libc-2.17.so
…etc…
```


EXECUTABLE MEMORY USAGE

```
00400000-00421000 r-xp 00000000 f9:00 67316594                           /usr/bin/less
```

Code segment. Contains code to execute, is obviously readonly and executable.

```
00620000-00621000 r--p 00020000 f9:00 67316594                           /usr/bin/less
```

Rodata segment. Contains readonly variables, obviously is readonly too.

```
00621000-00625000 rw-p 00021000 f9:00 67316594                           /usr/bin/less
```

Data segment. Contains variables, obviously NOT readonly.

```
00625000-0062a000 rw-p 00000000 00:00 0
```

BSS segment. Contains statically allocated variables that are zero-valued initially.

These are possible segments.

There can be more and others.

Code and data segments seem to be always used.

EXECUTABLE MEMORY USAGE

```
00400000-00421000 r-xp 00000000 f9:00 67316594                           /usr/bin/less
00620000-00621000 r--p 00020000 f9:00 67316594                           /usr/bin/less
00621000-00625000 rw-p 00021000 f9:00 67316594                           /usr/bin/less
00625000-0062a000 rw-p 00000000 00:00 0
```

421000 - 400000 = 21000

621000 - 620000 = 1000

625000 - 621000 = 4000 +

0x26000 = 155648 = 152K

```
# du -sh $(which less)
156K                /bin/less
```

accentureoperations

EXECUTABLE MEMORY USAGE

A way to look at the total size of a process is looking at the VSZ/VSS or virtual set size:

```
# ps -o pid,vsz,cmd -C less
  PID    VSZ CMD
19477 110256 less /proc/self/maps
```

(VSZ is in KB)

However, this tells the virtual set size is 110MB!

That is very much different from 152K of the ‘less’ executable?!

EXECUTABLE MEMORY USAGE

There's more stuff visible in the address space!

Let's execute less /proc/self/maps again:

```
00400000-00421000 r-xp 00000000 f9:00 67316594                           /usr/bin/less
00620000-00621000 r--p 00020000 f9:00 67316594                           /usr/bin/less
00621000-00625000 rw-p 00021000 f9:00 67316594                           /usr/bin/less
00625000-0062a000 rw-p 00000000 00:00 0
0255d000-0257e000 rw-p 00000000 00:00 0                                  [heap]
7f5149f59000-7f5150482000 r--p 00000000 f9:00 587351                     /usr/lib/locale/locale-archive
7f5150482000-7f5150639000 r-xp 00000000 f9:00 134334316                  /usr/lib64/libc-2.17.so
7f5150639000-7f5150838000 ---p 001b7000 f9:00 134334316                  /usr/lib64/libc-2.17.so
7f5150838000-7f515083c000 r--p 001b6000 f9:00 134334316                  /usr/lib64/libc-2.17.so
7f515083c000-7f515083e000 rw-p 001ba000 f9:00 134334316                  /usr/lib64/libc-2.17.so
7f515083e000-7f5150843000 rw-p 00000000 00:00 0
7f5150843000-7f5150868000 r-xp 00000000 f9:00 134567523                  /usr/lib64/libtinfo.so.5.9
7f5150868000-7f5150a67000 ---p 00025000 f9:00 134567523                  /usr/lib64/libtinfo.so.5.9
7f5150a67000-7f5150a6b000 r--p 00024000 f9:00 134567523                  /usr/lib64/libtinfo.so.5.9
7f5150a6b000-7f5150a6c000 rw-p 00028000 f9:00 134567523                  /usr/lib64/libtinfo.so.5.9
…etc…
```


EXECUTABLE MEMORY USAGE

The ‘less’ executable is a dynamic linked executable.

This can be seen by using the ‘ldd’ executable: shared library (loader) dependencies:

```
# ldd $(which less)
                linux-vdso.so.1 =>  (0x00007ffecc5bd000)
                libtinfo.so.5 => /lib64/libtinfo.so.5 (0x00007fe0db553000)
                libc.so.6 => /lib64/libc.so.6 (0x00007fe0db192000)
                /lib64/ld-linux-x86-64.so.2 (0x000055fde3a3b000)
```

This means the less executable requires four libraries to run!

- **linux-vdso.so.1** — **Meta-library, injected by the kernel to solve slow sys calls.**
- **libtinfo.so.5** — **Terminal info library.**
- **libc.so.6** — **General c library.**
- **ld-linux-x86-64.so.2** — **Program interpreter for loading libraries.**

EXECUTABLE MEMORY USAGE

Okay that's:

/bin/less 156K

/lib64/libtinfo.so.5.9 168K

/lib64/libc-2.17.so 2.1 M

/lib64/ld-2.17.so 156K +

2.580M versus VSZ: 110'256K !!

Turns out there is the following entry in maps:

```
7f5149f59000-7f5150482000 r--p 00000000 f9:00 587351                     /usr/lib/locale/locale-archive
```

```
# du -sh /usr/lib/locale/locale-archive
102M            /usr/lib/locale/locale-archive
```

Ah! 2.5M (exec+libs) + 102M (locale-archive) + some other bits in maps is approx. 110'245K!

accenture operations

VIRTUAL SET SIZE

That is VSZ, virtual set size, explained:

- **The total size of all memory allocations in an address space.**

Now that you know about virtual set size: forget about it.

It’s a useless statistic for determining memory usage, it says nothing… at all!

VIRTUAL SET SIZE

- Lazy loading and allocation.
  - Only the bare minimum of a file is read into the address space.
- Page fault
  - Accessing a page of a memory allocation that hasn't been read before in the address space causes the MMU to trigger an interrupt which calls the kernel 'page fault' routine.
    - 'fault' means the page is not yet available.
  - Causing this page to be read into the process' address space.
  - Commonly referred to as 'page in'.

There is a statistic that shows the amount of pages actually available in the address space.

This is what is called 'resident set size' or RSS.

RESIDENT SET SIZE

Let's look at the less command with 'ps' including the RSS column:

```
# ps -o pid,vsz,rss,cmd -C less
  PID    VSZ   RSS CMD
21908 110256  1940 less /proc/self/maps
```

That is 1940K versus 110256K !!!!!

Do you see how much work has been prevented by using lazy loading???

Actually, you can see how much is resident per memory allocation by looking at the 'smaps' file in proc.

RESIDENT SET SIZE

```
# less /proc/21908/smaps
…
7fc57c21c000-7fc582745000 r--p 00000000 f9:00 587351                     /usr/lib/locale/locale-archive
Size:              103588 kB
Rss:                  436 kB
Pss:                   48 kB
Shared_Clean:         436 kB
Shared_Dirty:           0 kB
…
```

Only a tiny portion of /usr/lib/locale/locale-archive is actually used (Size versus Rss).

RESIDENT SET SIZE

So, now we got virtual set size (VSZ or sometimes VSS) that is the full size of files and memory allocations, which only very seldom gets truly allocated. It is not a useful statistic for determining memory usage.

For being able to understand what is truly used in an address space we got resident set size (RSS). Does RSS show the actual memory used by a process?

No.

Linux has another trick up its sleeve.

Any page that is in the address space of a process, is shared with all of its child processes. By using a technique called ‘copy on write’ (COW), changes to a page make that page unique for that process.

PROPORTIONAL SET SIZE

So, we apparently got this fantastic page sharing mechanism.
But is there any way to see how and if this is being used?

Yes. Linux keeps a statistic in ‘smaps’ that is called ‘Pss’: proportional set size.

The PSS value is calculated in the following way:

- **The sum of the size of every resident page,**
- **For which the size of every page is divided by the number of processes it is shared with.**

PROPORTIONAL SET SIZE

Let's look at the 'smaps' file for the less command again, at the 'locale-archive' allocation:

```
# less /proc/21908/smaps
…
7fc57c21c000-7fc582745000 r--p 00000000 f9:00 587351                     /usr/lib/locale/locale-archive
Size:             103588 kB
Rss:                 436 kB
Pss:                  48 kB
Shared_Clean:        436 kB
Shared_Dirty:          0 kB
…
```

The full size is 103588K, of which only 436K is resident in the process space, which is shared with other processes, because proportional only 48K is used!

ORACLE MEMORY USAGE

Of course linux applies these techniques to running Oracle too:

```
# grep -e \- -e ^Size -e ^Rss -e ^Pss /proc/$(pidof ora_vktm_test)/smaps
00400000-13ac8000 r-xp 00000000 f9:02 214249537                          /u01/app/12.2.0.1/grid/bin/oracle
Size:             318240 kB
Rss:               43192 kB
Pss:                1916 kB
13cc7000-13ead000 r--p 136c7000 f9:02 214249537                          /u01/app/12.2.0.1/grid/bin/oracle
Size:               1944 kB
Rss:                 884 kB
Pss:                  37 kB
13ead000-13ef3000 rw-p 138ad000 f9:02 214249537                          /u01/app/12.2.0.1/grid/bin/oracle
Size:               280 kB
Rss:                 24 kB
Pss:                 24 kB
…etc…
```


LINUX SHARED MEMORY

How about shared memory?

There is no ‘owner’ of shared memory, it is used by every process it is shared with.

Every processes will page in whatever it actually needs:

System V shared memory:

```
60c00000-96800000 rw-s 00000000 00:05 163844                             /SYSV00000000 (deleted)
Size:             880640 kB
Rss:                 884 kB
Pss:                 124 kB
```

But does shared memory need to exist BEFORE processes can use it?

LINUX SHARED MEMORY

No:

```
# ipcs -mu

------ Shared Memory Status --------
segments allocated 4
pages allocated 256005
pages resident  83055
pages swapped   0
Swap performance: 0 attempts           0 successes
```

The same lazy allocation mechanism applies to shared memory too.
(pages allocated versus pages resident)

ORACLE SHARED MEMORY PAGING

Actually, there are a few settings that influences the way Oracle uses shared memory.

_touch_sga_pages_during_allocation

During startup the bequeathing process touches SGA pages, which allocates them.

Default value: false.

(Setting it to true doesn't seem to do anything on my test systems)

pre_page_sga

In the nomount phase a process (sa00) is spawned by the database that touches SGA pages, and vanishes after it's done.

Default value: false*.

This means that by default Oracle allocates SGA in a lazy manner too!

HUGE / LARGE PAGES

Hugepages is a joint operating system and hardware (MMU) feature to increase the memory administration unit (‘page’) size.

Regular page size is 4K.

2MB page size support is dependent on cpu flag ‘pse’:

```
# grep pse /proc/cpuinfo
flags                           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush mmx fxsr sse sse2 ht syscall nx rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc pni
pclmulqdq ssse3 cx16 sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx rdrand hypervisor lahf_lm abm
```

1GB page size support is dependent on cpu flag ‘pdpe1gb’:

```
# grep pdpe1gb /proc/cpuinfo
flags                           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon
pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2
ssse3 cx16 xtpr pdcm dca sse4_1 sse4_2 x2apic popcnt aes xsave avx lahf_lm arat epb xsaveopt pln pts dts
tpr_shadow vnmi flexpriority ept vpid
```

HUGE / LARGE PAGES

Why use hugepages?

- **Page addresses are cached in a memory area that is called ‘transaction lookaside buffer’ or TLB.**
- **Having fewer entries improves TLB cache hit ratio, and reduces CPU usage.**
- **Hugepages are non-swappable.**
- **Hugepages reduces per process page address administration size: Pagetable size.**

Hugepages memory creation requires physical memory that has not been used by 4K pages (‘fragmented’) previously.

Memory that is configured for hugepages, is not usable for regular 4K allocations ! ! ! !

Even if none of the hugepages are used by any process.

HUGE / LARGE PAGES

```
# grep ^HugePage /proc/meminfo
HugePages_Total:    1126
HugePages_Free:      784
HugePages_Rsvd:      159
HugePages_Surp:        0
```

Used 1126 (tot) - 784 (free) = 342 Used

Reserved = 159 Reserved

Real free 784 (free) - 159 (rsvd) = 625 Free +

1126 Total

accentureoperations

HUGE / LARGE PAGES

The reporting for hugepages memory usage by ipcs is ‘kind of’ broken:

```
# ipcs -mu
------ Shared Memory Status --------
segments allocated 4
pages allocated 256517
pages resident  175105
```

256517 * 4K = 1002M.

In other words: ipcs -mu reports the hugepages as regular (4K) pages!

HUGE / LARGE PAGES

The reporting for hugepages memory allocations in /proc/PID/smaps is ‘kind of’ broken too:

```
# less /proc/$(pidof ora_pmon_test)/smaps
60400000-96400000 rw-s 00000000 00:0e 360449                             /SYSV00000000 (deleted)
Size:             884736 kB
Rss:                   0 kB
Pss:                   0 kB
Shared_Clean:          0 kB
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:         0 kB
Referenced:            0 kB
Anonymous:             0 kB
AnonHugePages:         0 kB
Swap:                  0 kB
KernelPageSize:     2048 kB
MMUPageSize:        2048 kB
```

This is the variable SGA shared memory alloc. Rss says we allocated nothing. Not true.

accenture operations

ANONYMOUS ALLOCATIONS

Outside of the code, data and rodata allocations mapped from a file into memory, there's memory allocations NOT backed by a file.

```
00621000-00625000 rw-p 00021000 f9:00 67316594                           /usr/bin/less
00625000-0062a000 rw-p 00000000 00:00 0
0255d000-0257e000 rw-p 00000000 00:00 0                                  [heap]
7f5149f59000-7f5150482000 r--p 00000000 f9:00 587351                     /usr/lib/locale/locale-archive
7f5150482000-7f5150639000 r-xp 00000000 f9:00 134334316                  /usr/lib64/libc-2.17.so
7f5150639000-7f5150838000 ---p 001b7000 f9:00 134334316                  /usr/lib64/libc-2.17.so
7f5150838000-7f515083c000 r--p 001b6000 f9:00 134334316                  /usr/lib64/libc-2.17.so
7f515083c000-7f515083e000 rw-p 001ba000 f9:00 134334316                  /usr/lib64/libc-2.17.so
7f515083e000-7f5150843000 rw-p 00000000 00:00 0
7f5150843000-7f5150868000 r-xp 00000000 f9:00 134567523                  /usr/lib64/libtinfo.so.5.9
```

The allocations above are BSS segments.

Memory explicitly allocated using malloc() shows up identical, also being anonymous allocations.

The PGA of Oracle processes is allocated in anonymous allocations.

MEMORY USAGE CALCULATION

Is there a simple way to calculate actual, total, memory usage for Oracle on Linux?

Answer: no, not really.

VSZ: has nothing to do with actual usage.

RSS: describes memory allocation for a process, the pages are allocated, but pages can be shared, so RSS will hugely oversubscribe. Unless RSS = PSS.

PSS: describes proportional memory usage, only if all processes PSS sizes are calculated together, at the same time, it will tell actual allocation (this is VERY fluent information!).

For process private memory, PSS is the way to go. But for shared memory, you can't use a process centric method. Memory could be allocated by a process that quit.

MEMORY USAGE CALCULATION

The only way to realistically say something about linux memory usage, is using these statistics in /proc/meminfo:

- **MemFree: idle memory.**
- **KernelStack, Slab: kernel memory.**
- **Buffers: linux kernel buffers for IO.**
- **SwapCached: cache for swapped pages.**
- **Cached (- Shmem): linux page cache and memory mapped files.**
- **PageTables: virtual to physical memory addresses administration.**
- **Shmem: small pages shared memory allocation.**
- **AnonPages: all anonymous memory allocations.**
- **Hugepages_Total - Hugepages_Free: Used hugepages (number in pages)**
- **Hugepages_Rsvd: Logically allocated, not yet physically allocated hugepages (number in pages)**
- **Hugepages_Free-Hugepages_Rsvd: true free hugepages (number in pages)**

DEMO: LIVE MEMORY USAGE

If you want to do the same yourself:

1. Install node_exporter (https://github.com/prometheus/node_exporter)

- **This is the fetcher ('exporter') of the statistics.**

2. Install prometheus (https://github.com/prometheus/prometheus)

- **This is the persistence layer for the statistics.**

3. Install grafana (http://grafana.com)

- **This is the presentation/visualisation layer.**

4. Setup prometheus to fetch data from node_exporter (https://resin.io/blog/monitoring-linux-stats-with-prometheus-io/)

- **node_exporter → prometheus.**

5. Add prometheus as a grafana source (https://prometheus.io/docs/visualization/grafana/)

- **prometheus → grafana**

6. Import grafana dashboard id: 2747

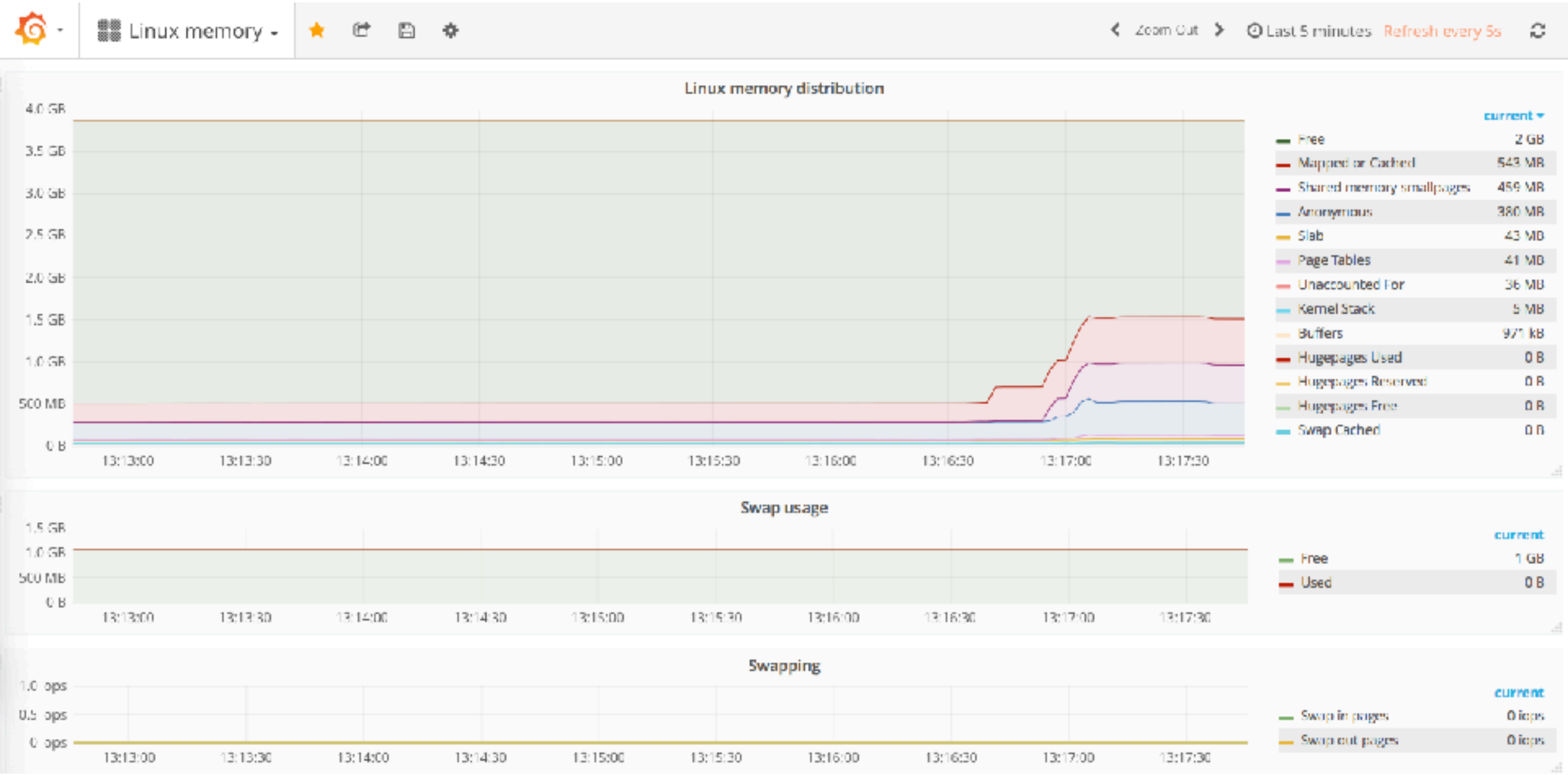
(https://fritshoogland.wordpress.com/2017/07/31/installation-overview-of-node_exporter-prometheus-and-grafana/)

DEMO: LIVE MEMORY USAGE

Now let's startup the Oracle database.

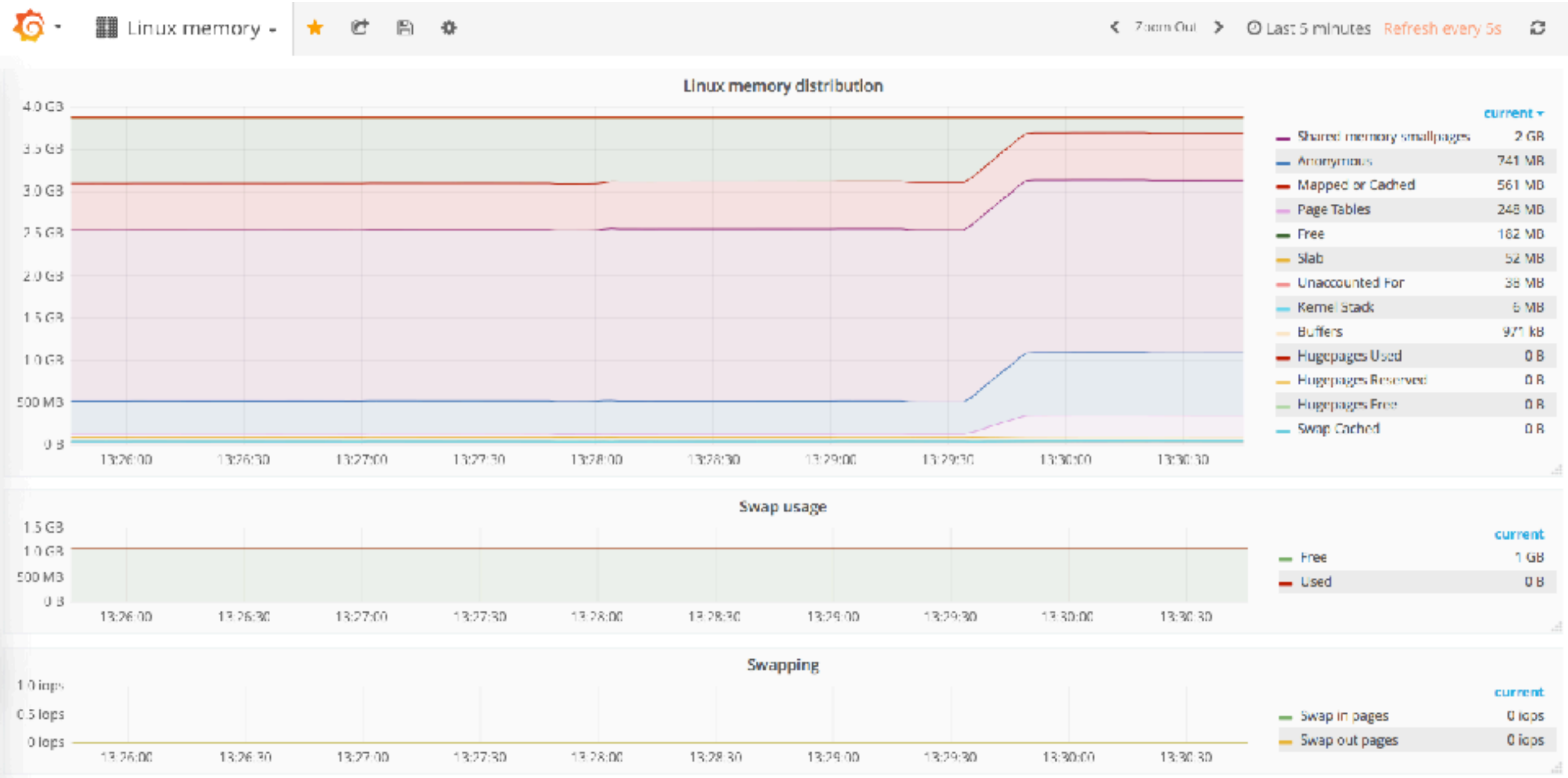
Important settings:

- **sga_target = 2G**
- **pga_aggregate_target = 500M***
- **filesystemio_options = setall**



accentureoperations

 |



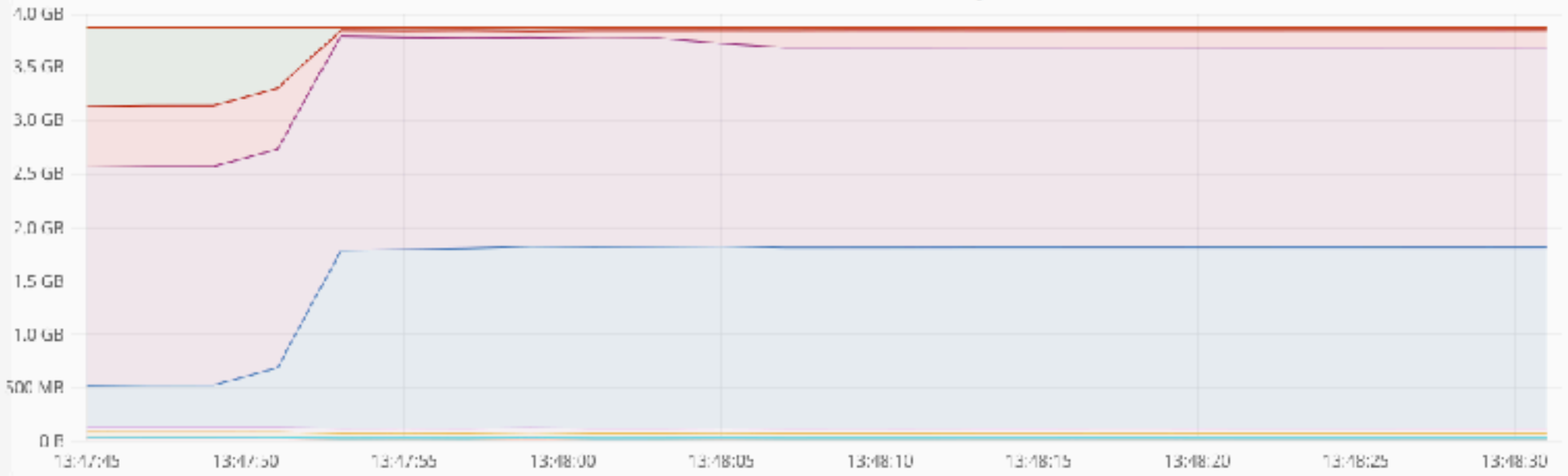
accentureoperations

 |

Linux memory

Zoom Out

Linux memory distribution

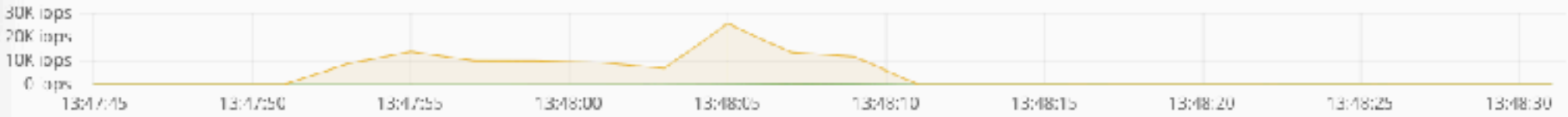


	current
Shared memory smallpages	2 GB
Anonymous	2 GB
Mapped or Cached	157 MB
Page Tables	47 MB
Slab	43 MB
Free	27 MB
Unaccounted For	27 MB
Kernel Stack	4 MB
Swap Cached	967 kB
Hugepages Used	0 B
Hugepages Reserved	0 B
Hugepages Free	0 B
Buffers	0 B

Swap usage

	current
Free	858 MB
Used	215 MB

Swapping



	current
Swap in pages	1 iops
Swap out pages	0 iops

accentureoperations

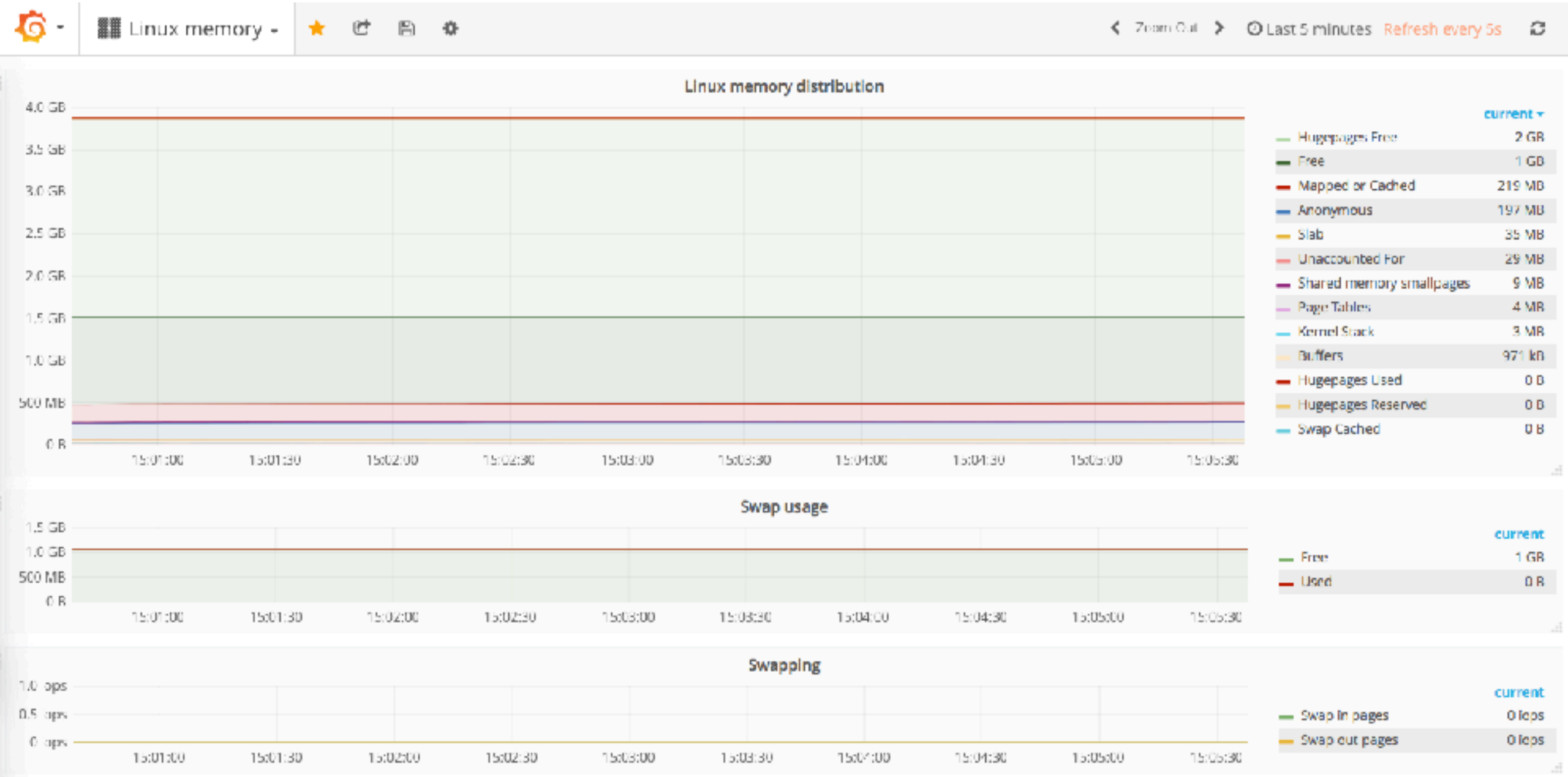
 |

accentureoperations

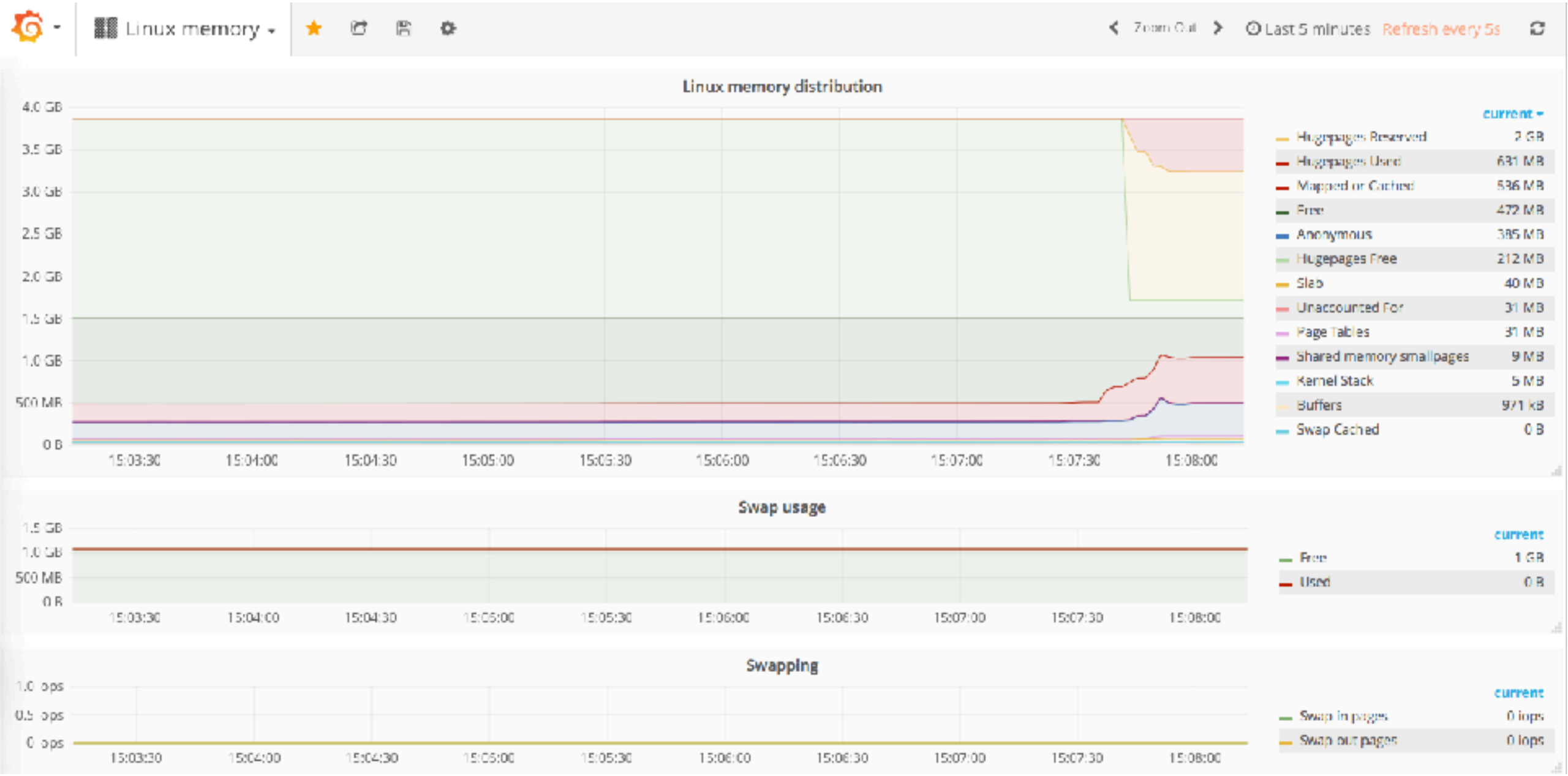
 |

DEMO: LIVE MEMORY USAGE

I didn’t mention huge pages (yet)! Let’s see how that looks like:

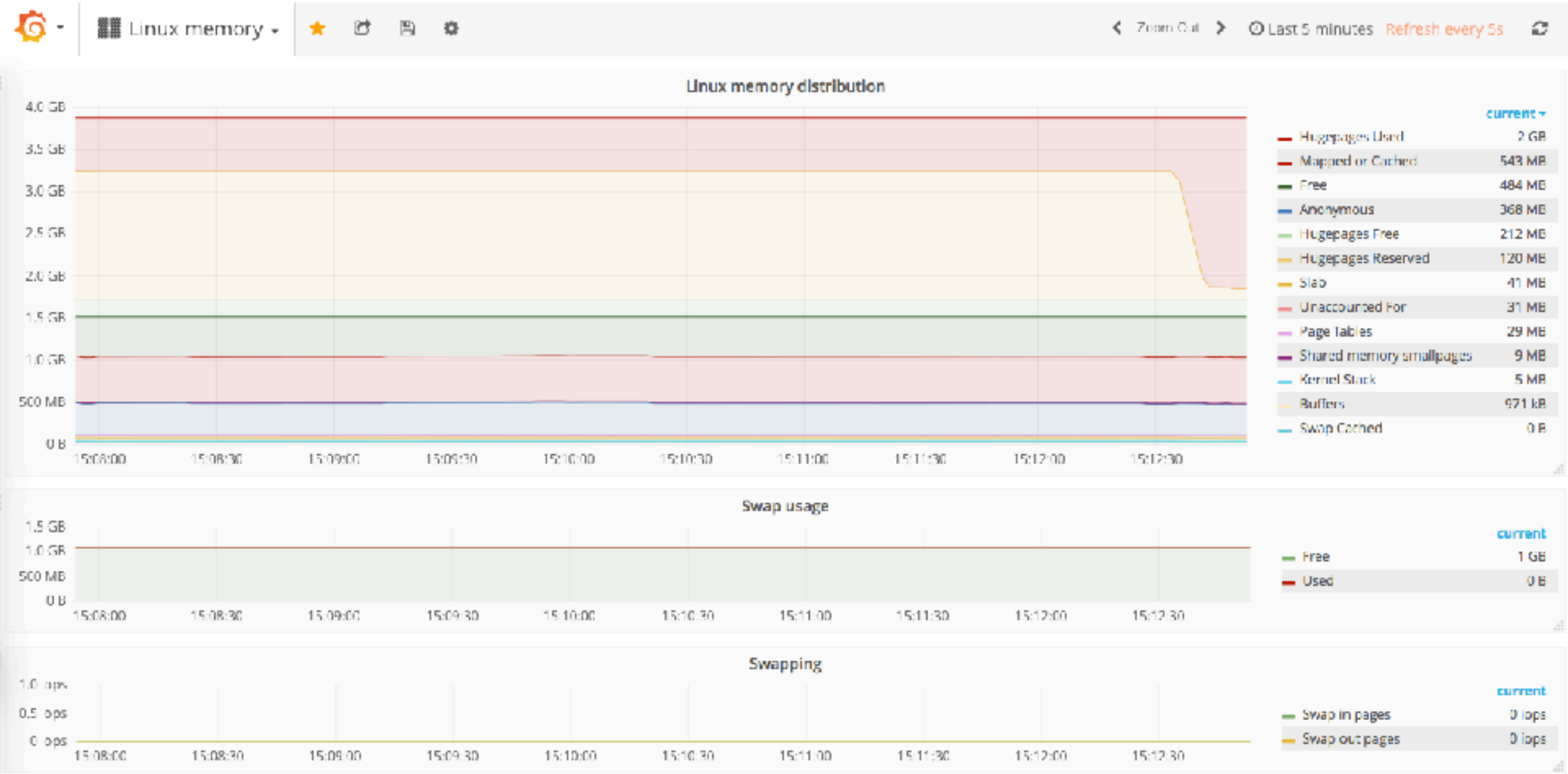


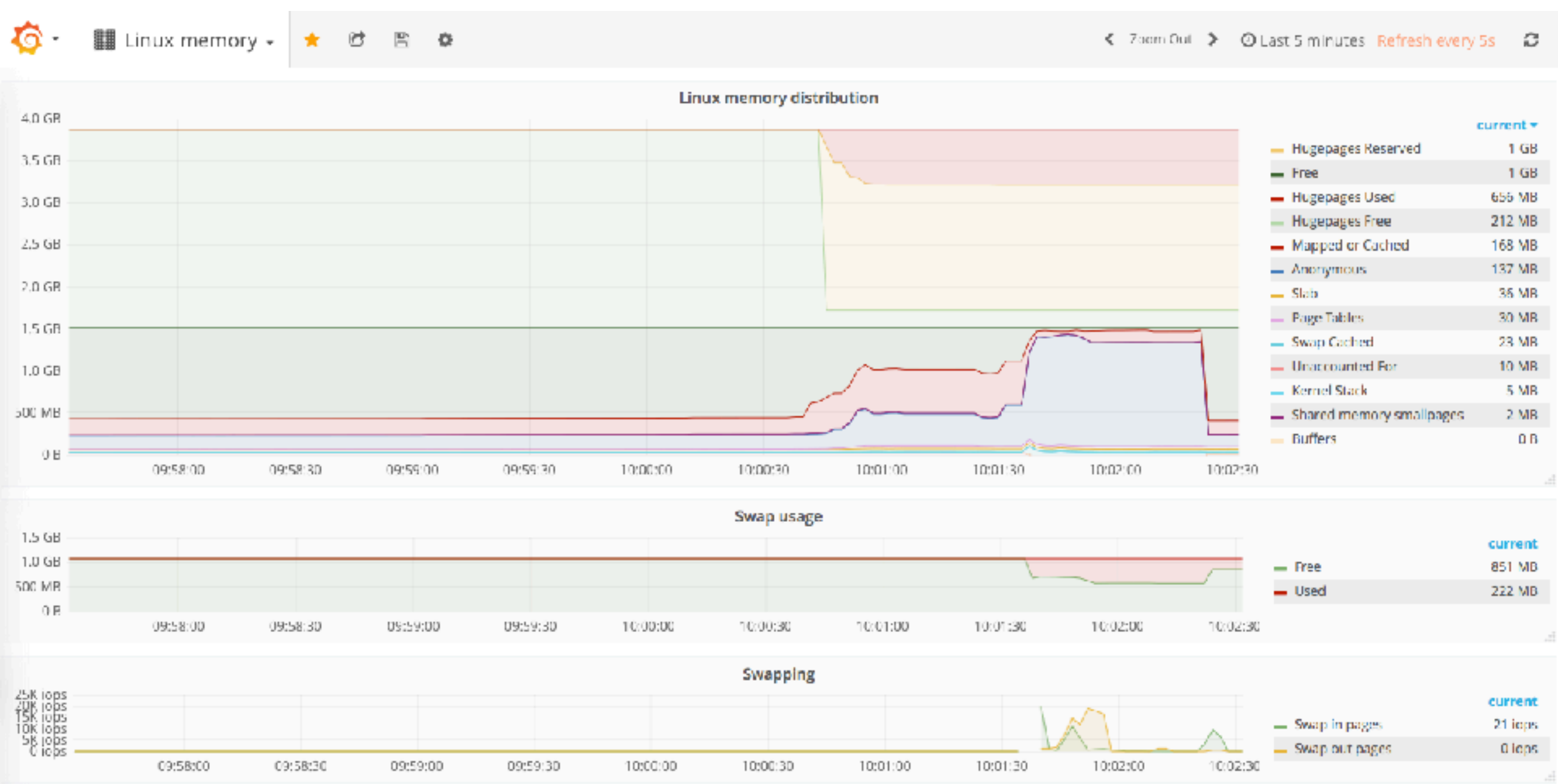
accentureoperations



accentureoperations

 |



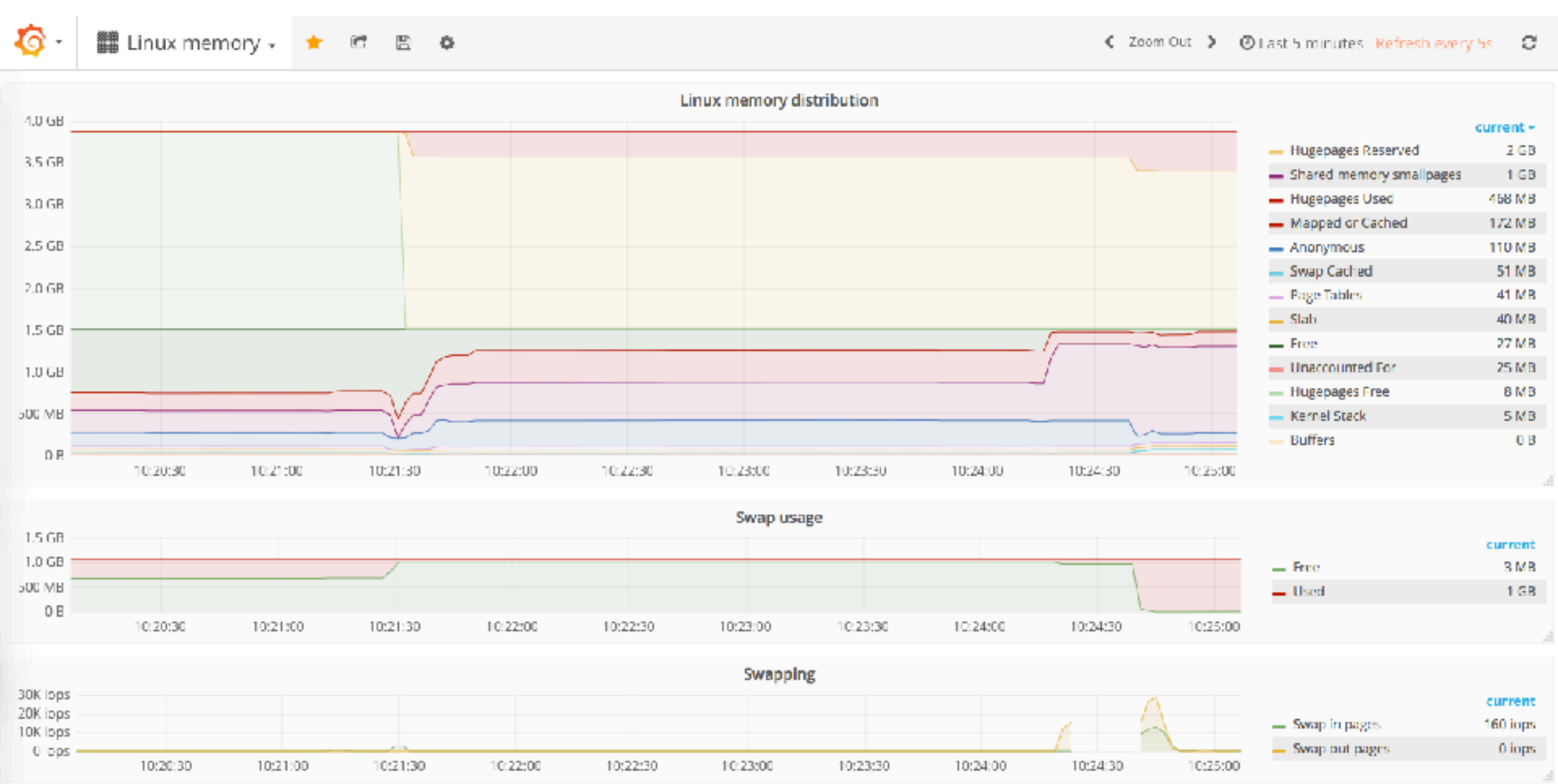


accentureoperations

 |

accentureoperations

 |



accentureoperations

 |

CONCLUSION

A computer system can use a limited amount of memory because of memory manager limitations and physical slots.

The linux kernel needs some memory to run, which means MemTotal in /proc/meminfo will not give you the total physical amount of memory.

An executable can be ‘dynamically linked’, which means it uses libraries for additional functionality. The oracle database executable is an example of that.

CONCLUSION

The full size of an allocation is called virtual set size, which is not a useful statistic for determining memory usage.

The amount of pages for any allocation in an address space that is truly directly available is called the resident set size.

Any process shares its memory allocations with its child processes. Only once a page is modified, it is made unique to a process.

The page sharing can be seen by looking at the resident set size and the proportional set size in /proc/PID/smaps.

CONCLUSION

Huge pages memory can only be used for huge pages allocations. Even if there is a shortage of small pages.

Shortage of (smallpages) memory results in pages being moved to the swap device.

Cached/Mapped and Anonymous pages are the first to be swapped.

Shared memory pages will be swapped when memory pressure persists.

Other/kernel related pages can be swapped too, but do not seem to get swapped too much.

PageTables: if these take a significant amount, consider hugepages.

CONCLUSION

The lazy loading can make memory shortage/swapping diagnosis hard.

Any memory allocation is done lazy.

The Oracle SGA is not fully physically allocated at startup (depending on settings). This means when sized too big (especially in a consolidated environment) it might take some time before memory is truly allocated and memory pressure becomes visible.

The Oracle PGA is not allocated at startup, but allocated per process on demand, and can free memory too. PGA_AGGREGATE_TARGET is a target value for instance work area's. PGA memory always is in Anonymous memory allocations on linux.

PGA can easily allocate huge amounts of memory, for example with 'collections'.

Having lots of sessions and lots of cursors can lead to a lot of Anonymous memory locked and wasted.

CONCLUSION

Because any memory operation is done lazy, moving swapped pages back into memory is also done lazy. This means a past memory shortage can lead to swap being in use currently, with no actual memory problem.

There only is a memory pressure/swapping problem if there is a persistent swapping out and swapping in happening, or single way at a high rate, leading to processes getting stuck. Not because swap is filled for X%