Malloc Lives In Userspace

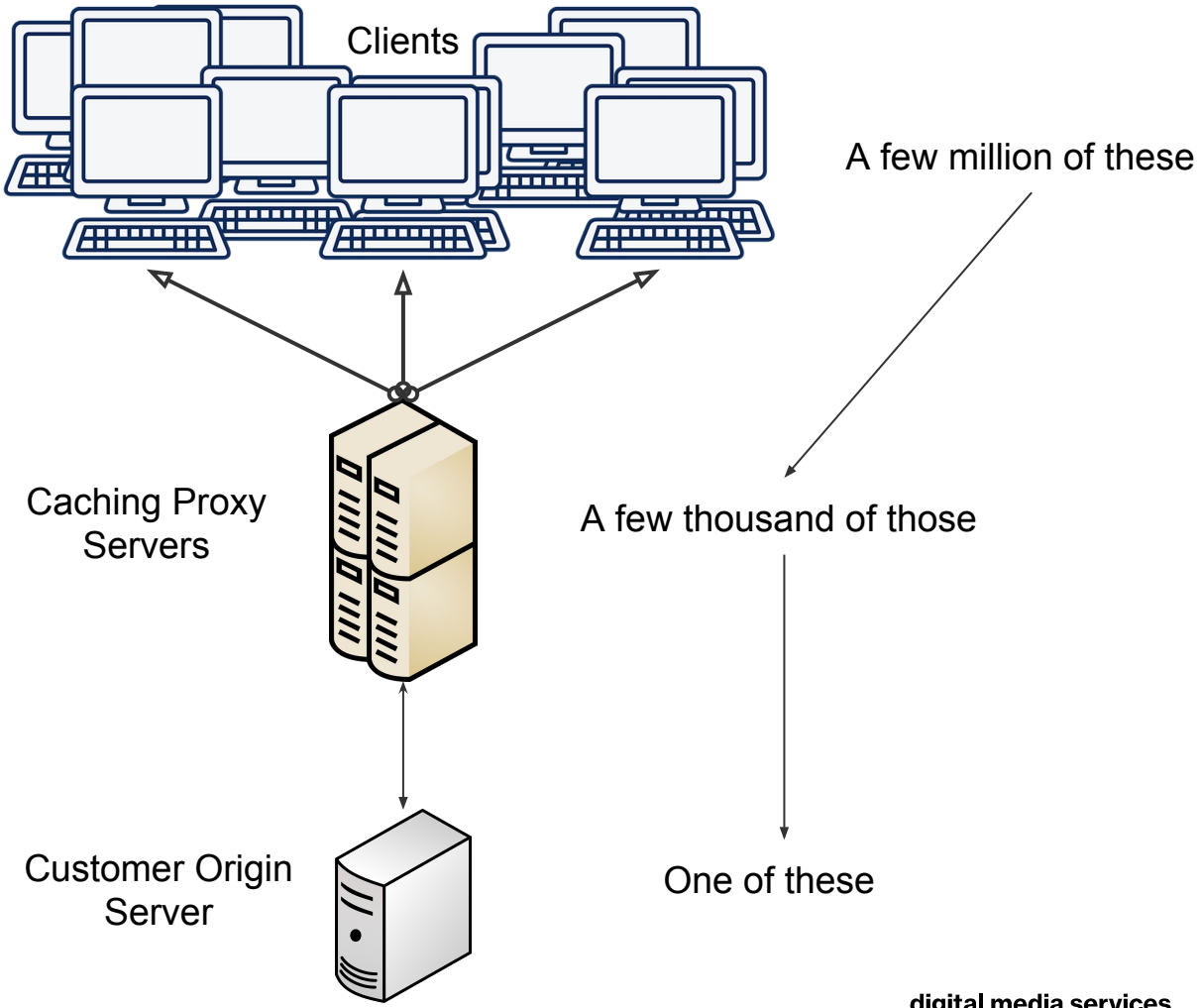
(And why you care)

March 03, 2017

Will Rosecrans

will.rosecrans@verizondigitalmedia.com

I work on a CDN. We serve a few Terabits of traffic to millions of clients on a typical day.



verizon

digital media services

Malloc is a function in C.

It gives you the address of some memory that you can use.

```
int how_much_memory = 1

int main(int argc, char** argv) {
    char *myAwesomeData =
        malloc(how_much_memory);
    *myAwesomeData = 42;
    free(myAwesomeData);
    return 0;
}
```

Most of your favorite languages use it under the hood.

Python is written in C

```
$ grep malloc `which python`
Binary file /usr/bin/python matches
```

Ruby is written in C

```
$ grep malloc /usr/lib/libruby-1.9.1.so.1.9
Binary file /usr/lib/libruby-1.9.1.so.1.9 matches
```

Google's Grumpy is implemented with Go, Go binaries link to libc and call malloc

```
/usr/local/go/bin$ ldd go
     libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
/usr/local/go/bin$ grep malloc go
Binary file go matches
```

JRuby is in Java, JVM is written in C, and will call malloc

```
/usr/lib/jvm/java-7-openjdk-amd64/jre/lib/amd64$ grep malloc
libjava.so
Binary file libjava.so matches
```

Malloc is guaranteed to get memory from...

Somewhere.*

*Unless it doesn't get it from anywhere. Linux will usually let a process allocate more memory than exists, but malloc is allowed to return NULL. Then you get nothing. Nothing except disappointment.

- **Malloc gets memory from "the system"**
  - But syscalls are slow
- **So it grabs a big chunk of memory from the OS**
  - and doles it out in small chunks to you
  - Malloc mainly uses sbrk or mmap
    i. sbrk grows from the bottom up
    ii. mmap grows from the top down
- **Your process is blamed for memory that malloc gets from the kernel**
  - Nobody cares about what you get from malloc
  - Except you
  - And the address space it got from kernel may not be in RAM yet because the chunk of memory doesn't have to be *in-memory*.

Dust to Dust.

Ashes to Ashes.

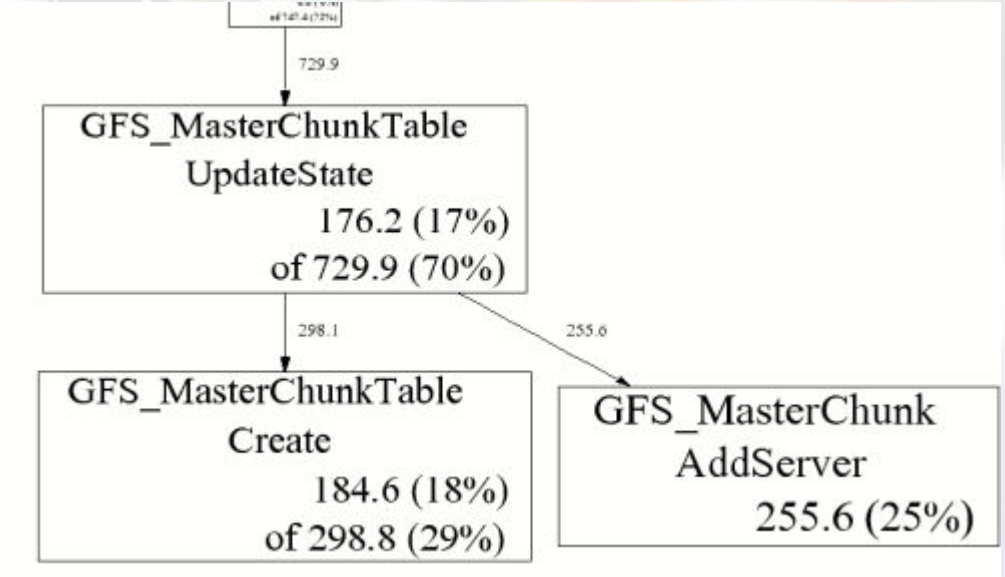
Malloc to Free.

- **When you are done with something you free it.**
- **You don't give it back to the kernel**
- **Malloc might give it back to the kernel**
  - But maybe not. Who knows?
  - Whatever. Get off my back.
- **As long as any part of a map/page is in use, malloc can't give it back to the kernel**
- **The system can't defrag RAM**
  - No universal way to move objects arbitrarily in application's virtual address space.
  - The extra indirection would be slower.

You can replace malloc.

It comes in assorted styles and sizes.

We use tcmalloc which lets us make cool graphs



In a longer talk this is probably where I'd demonstrate using tcmalloc with LD_PRELOAD, how to make a graph like the above, and make detailed comparisons to other mallocs like jemalloc.

The kernel has a page cache.

Not a file cache. *

*It might also have a file cache. But don't worry about that here. Filesystems like ZFS and SquashFS really don't conform to this model. But hey, it's a lighting talk, not the ultimate deep dive into how filesystems work. If you can read all this, you probably downloaded the slides after the talk, so you can look up the details if you are interested. It's a deep rabbit hole.

```
f = open('/tmp/importantfile.txt')
important_data = f.read()
```

- **The kernel reads sectors or blocks* on the device, and stores them in 4KB pages* in page cache memory owned by the kernel.**
- **Then the filesystem figures out files from that data in memory, and copies it into the application's address space.**
- **Your application never reads directly from disk.***
- **Sendfile() skips copying the file into your address space. Storage and Network performance is unchanged - sendfile is a memory optimization.**

* Blocks are typically 512 bytes on older drives, or 4 KB on newer ones. Run "fdisk -l" to see yours. There's also the filesystem block size, which may be larger or smaller.
* Run "getconf PAGE_SIZE" to see the value on your system. 4 KB is common. Anyhow, the size of reads will be influenced in some way by the block size, page size, filesystem parameters, maybe phase of the moon. But it won't be guaranteed to relate to the file size. Files don't matter yet. Files aren't guaranteed to be aligned to pages or blocks. They aren't guaranteed to be contiguous, or not to have holes, can be compressed or share pages because of dedup.
* This is true even with the "O_DIRECT" flag, which tries to minimize impact on the page cache. Your application still isn't directly reading form the disk, and you aren't guaranteed to save the extra copy or bypass the page cache.

MAP_32BIT

LuaJIT x64

So much pain for us.

*You have to pronounce it “X Sixty Four” rather than “X Six Four” for the syllables to work as a haiku.

- **LuaJIT uses tagged pointers**
  - 31 bit address space
  - All data must be in a specific address range
  - Near (but not at) the bottom of memory
- **It allocated using mmap’s MAP_32BIT mode**
  - But not until after the host app was already using a bunch of memory to load configs and do initialization
- **tcmalloc has some tuning features**
  - TCMALLOC_SKIP_MMAP
  - TCMALLOC_SKIP_SBRK
  - Skip sbrk to only allocate from top down

Fork does Copy on Write

COW saves memory

COW saves time making copies

- **Fork is a low level function from the C API**
- **Python multiprocessing module uses fork**
  - Load a bunch of data once
  - fork()
  - process it faster in parallel
- **Both sides share data that stays the same**

Fork does Copy on Write

Penny wise

Pound foolish

Then you try to run it on a dual socket server...

- **Process A running on socket #1**
  - Process B running on socket #2
- **Kernel ping pongs data between sockets**
  - Kernel can't decide who owns it.
- **You save one copy by paying for millions.**

Having two copies can be much faster than having one. Having more than Num_Sockets copies usually not faster than 2, wastes a lot of space.

Userspace can't see where memory physically lives, and this can and does get shifted around constantly.

Thank you.

Get these slides at the VDMS engineering blog:
http://eng.verizondigitalmedia.com
It's also got other fun and exciting topics like core dumps!

You can finally read all the fine print if you download the slides to read on your own time.